Proceedings of the
3rd International Workshop on
Software Engineering and Middleware
SEM 2002

Orlando, Florida (U.S.A.), 20-21 May 2002

(Edited by Alberto Coen-Porisini)
3rd International Workshop on
Software Engineering and Middleware - SEM 2002
Orlando, Florida (U.S.A.), 20-21 May 2002

General Chair
  Wolfgang Emmerich (University College London, UK)

Programme Committee Chair
  Alberto Coen-Porisini (Università dell'Insubria, Italy)

Programme Committee Co-Chair
  Andre van der Hoek (University of California Irvine, USA)

Local Chair
  Sebastian Uchitel (Imperial College, UK)

Programme Committee
  Gordon Blair (Lancaster University, UK)
  Alfred Bröckers (Adesso, Germany)
  Peter Croll (University of Wollongong, Australia)
  Flavio De Paoli (Università di Milano Bicocca, Italy)
  Premkumar Devanbu (University of California - Davis, USA)
  Naranker Dulay (Imperial College, UK)
  Wolfgang Emmerich (University College London, UK)
  Rachid Guerraoui (EPFL, Switzerland)
  Arno Jacobson (University of Toronto, Canada)
  Mehdi Jazayeri (TU Vienna, Austria)
  Wojtek Kozacynski (Rational, USA)
  Peter Löhr (Free University of Berlin, Germany)
  Dino Mandrioli (Politecnico di Milano, Italy)
  Julien Maisonneuve (Alcatel, France)
  Koichiro Ochimizu (Japan Institute of Technology, Japan)
  David Rosenblum (University of California Irvine, USA)
  Gustavo Rossi (Universidad Nacional de La Plata, Argentina)
  Isabelle Rouvellou (IBM Watson Research, USA)
  Dirk Slama (Shinka Technologies, Germany)
  Stefan Tai (IBM Watson Research, USA)
  Andre van der Hoek (University of California Irvine, USA)
# Table of Contents

## SESSION I - ARCHITECTURES

**Flexible Distributed Process Topologies for Enterprise Applications**  
Christoph Hartwich  
3

**An Architecture for the UniFrame Resource Discovery Service**  
Nanditha N. Siram, Rajeev R. Raje, Andrew M. Olson, Barrett R. Bryant, Carol C. Burt, Mikhail Auguston  
22

**An Architecture of a Quality of Service Resource Manager Middleware for Flexible Multimedia Embedded Systems**  
Marisol Garcia Valls, Alejandro Alonso, José F. Ruiz, Angel Groba  
39

## SESSION II - SPECIFICATION

**An Event/Rule Framework for Specifying the Behavior of Distributed Systems**  
Javier A. Arroyo-Figueroa, José A. Borges, Néstor Rodríguez, Amarilis Cuaresma-Zevallos, Edwin Moulier-Santiago, Miguel Rivas-Avilés, Jaime Yeckle –Sánchez  
59

**Development and Specification of Interfaces for Standard-driven Distributed Software Architectures in the E-learning Domain**  
Luis Anido, Manuel Caeiro, Juan Santos, Judith Rodríguez  
72
SESSON III - COMPONENTS AND ADAPTATIONS

A Component-Based Architecture for Collaborative Applications in Internet
Flavio De Paoli 94

Composing Distributed Components with the Component Workbench
Johann Oberleitner, Thomas Gschwind 106

FORMAware: Framework of Reflective Components for Managing Architecture Adaptation
Rui S. Moreira, Gordon S. Blair, Eurico M. Carrapatoso 118

Type Based Adaptation An Adaptation Approach for Dynamic Distributed Systems
Thomas Gschwind 132

SESSON IV - TECHNOLOGIES

An Evaluation of the Use of Enterprise Java Beans 2.0 Local Interfaces
Hans Albrecht Schmid 146

Dynamic Instrumentation for Jini Applications
Denis Reilly, A. Taleb-Bendiab 160

SESSON V - SERVICES

Message Queuing Patterns for Middleware-Mediated Transactions
Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, Alexander Totok 179

Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Middleware
Gianpaolo Cugola, Gian Pietro Picco, Amy L. Murphy 191

Active replication of software components
Juan Pavón, Luis M. Peña 206

Building Test Constraint for Distributed System Testing with Middleware
Jessica Chen 219
Session I – Architectures
Flexible Distributed Process Topologies for Enterprise Applications

Christoph Hartwich*
Freie Universitaet Berlin
Takustrasse 9
D-14195 Berlin, Germany
hartwich@inf.fu-berlin.de

Abstract
Enterprise applications can be viewed as topologies of distributed processes that access business data objects stored in one or more transactional datastores. There are several well-known topology patterns that help to integrate different subsystems or to improve nonfunctional properties like scalability, fault tolerance, or response time. Combinations of multiple patterns lead to custom topologies with the shape of a directed acyclic graph (DAG). These topologies are hard to build on top of existing middleware and even harder to adapt to changing requirements. In this paper we present the principles of an enterprise application architecture that supports a wide range of custom topologies. The architecture decouples application code, process topology, and data distribution scheme and thus allows for an easy adaptation of existing topologies. We introduce RI-trees for specifying a data distribution scheme and present rules for RI-tree-based object routing in DAG topologies.

1. Introduction
Enterprise applications are transactional, distributed multi-user applications that are employed by organizations to control, support, and execute business processes. Traditionally, data-intensive enterprise applications have been built on top of centralized transaction processing monitors. Nowadays, these TP monitors are replaced by object-oriented multi-tier architectures where entities of the business domain are typically represented as business objects. To differentiate between process-centric and data-centric business objects we use the terms business process object and business data object, respectively. In this paper we take a data-centric view and thus focus on business data objects, which constitute the application’s object-oriented data model. A business data object represents an entity of persistent data that is to be accessed transactionally by processes of the enterprise application, e.g., a Customer or Account object. In typical enterprise applications business data objects (or short: data objects) reside in and are managed by the second last tier while their persistent state is stored in the last tier. The last tier consists of one or more transactional datastores, for example, relational database management systems.

Current platforms for object-oriented enterprise applications (like CORBA [5] or Sun’s J2EE [9]) provide excellent support for two-tier architectures and three-tier architectures

* This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).
with ultra-thin clients, e.g., web browsers. However, many large-scale applications need more advanced structures that differ from these simple architectures, for example, to meet specific scalability or fault tolerance requirements.

In Section 2 we introduce process topologies, which provide an appropriate view on the distributed structure of an enterprise application. In addition, we present several well-known patterns that are used in many topologies and motivate the need for flexible DAG topologies. Section 3 discusses types of connections for building process topologies and difficulties developers typically face when custom topologies are required. An enterprise application architecture for flexible topologies that decouples application code, process topology, and data distribution scheme is outlined in Section 4. Two key aspects of the architecture are discussed in the following two sections: In Section 5 we present RI-trees for specifying a data distribution scheme. Section 6 proposes rules for RI-tree-based object routing in DAG topologies. We discuss related work in Section 7 and finally give a brief summary in Section 8.

2. Process Topologies

Enterprise applications can be viewed as topologies of distributed processes. Formally, a process topology is a directed acyclic graph (DAG). Nodes of the DAG represent distributed, heavyweight, operating system level processes (address spaces) which are either transactional datastores (leaf nodes) or application processes (inner nodes). Two processes may but are not required to be located on the same machine. Each edge in the DAG represents a (potential) client/server communication relationship. The concrete communication mechanism, e.g., RPC-style or message-oriented, is not important at this level of abstraction. Figure 1 shows an example of such a process topology.

![Figure 1. Example of a topology of distributed processes.](image)

Nodes that are not connected with an edge can never communicate directly. In enterprise applications there are many reasons for restricting communication and not allowing processes to directly communicate with arbitrary other processes, for instance:

- **Security** – A sub-system (nodes of a sub-graph) is shielded by a firewall which allows access to processes of the subsystem only via one or more dedicated processes.
Scalability – Highly scalable systems often require a sophisticated topology to employ services for caching, load balancing, replication, or concentration of connections. For example, allowing client processes to directly connect to datastores reduces communication overhead, but then the overall system cannot scale better than two-tier architectures, which are well-known for their restricted scalability.

Decoupling – The concrete structure and complexity of a sub-system is to be hidden by one or more processes that act as a facade to the subsystem. For instance, in a three-tier structure tier two can shield the client tier from the complexities of tier three.

2.1. Process Topology Patterns

There are a number of patterns that can be found in many enterprise applications and that are directly related to process topology. These topology patterns can be viewed as high-level design patterns [2], [4], where distributed processes represent coarse-grained objects. In Figure 2 six topology patterns are depicted:

1. Process replication – An application process is replicated and the load produced by its clients is horizontally distributed among the replicated processes.

2. Distributed data – Instead of using a single datastore, data objects are distributed (and possibly replicated) among multiple datastores. This pattern facilitates load distribution and basic fault tolerance.

3. Proxy process – A proxy process is placed between the proxified process and its clients. Tasks are shifted from the proxified process to the proxy to achieve vertical load distribution. For instance, the proxy can cache data and process a subset of client requests without having to contact the proxified process. In addition, this pattern allows to add functionality to the services provided by the proxified process without having to modify the proxified process.

4. Group of proxy processes – This is a common combination of Pattern 1 and Pattern 3. By introducing a new tier of proxy processes, load is first shifted vertically and then distributed horizontally among the replicated proxy processes. This pattern is typically employed for a concentration of connections when handling too many client connections would overload a server. The replicated proxies receive client requests, possibly perform some pre-processing (e.g., pre-evaluation) and forward the requests to the server using only a view “concentrated” connections.

5. Integration of subsystems – Often existing data and/or applications have to be integrated. This pattern has two variants: a) Equal integration by introducing a federation layer/tier on top of the existing subsystems. b) Integration of subsystems into another (dominant) system.

6. Mesh – Redundant connections are added to create alternative paths of communication relationships between processes. Usually, this pattern is employed in conjunction with Pattern 1 and facilitates fault tolerance and horizontal load distribution.
2.2. Construction of Custom Process Topologies

The patterns presented above are used to describe parts of a given process topology at a high level of abstraction. In addition, a pattern can be applied to a process topology and hence define a transformation step. Ideally, a concrete process topology is constructed by starting with a standard topology (e.g., simple three-tier) and successively applying combinations of patterns to parts of the topology as required by the enterprise application.

What an adequate process topology looks like is highly application-specific and depends on many factors, including organizational and legal requirements, existing hardware and operating systems, underlying network structure, existing systems to be integrated, number and location of clients, typical usage patterns, and performance/scalability/fault tolerance requirements.

Many of these factors tend to change over time. For example, a successful enterprise application might have to handle a rapidly growing number of clients, or new functionality is introduced that requires a higher level of fault tolerance. To adapt a custom process topology to changing requirements, developers have to transform it by applying, removing,
and modifying topology patterns. Thus, it is desirable to have a flexible process topology that can easily be adapted, for example by changing configuration data. With application code designed for a specific topology, this is extremely difficult. Therefore, a flexible process topology requires that application and underlying process topology are decoupled as much as possible.

3. Connection Types in Process Topologies

Communication relationships in a process topology are implemented by connections. In this section we examine connections offered by current middleware and discuss problems that occur when custom topologies are built on top of them.

In general, connections can be categorized along various dimensions, for example, the degree of coupling, available bandwidth, or underlying communication protocols. For our problem analysis it is useful to define categories with regard to how business data objects are remotely accessed and represented. Based on that dimension, most connections used in current enterprise applications can be categorized into four types:

3.1. Type D – Datastore Connections

A Type D connection connects an application process (client, inner node) with a datastore (server, leaf node). The client accesses data objects persistently stored in the datastore. Often generic access is supported via a query interface. Type D connections are typically provided by datastore vendors, examples are ODBC, JDBC, and SQL/J. For many non-object-oriented datastores, there are adapters that provide (local) object-oriented access to Type D connections, for instance, object-relational mapping frameworks.

3.2. Type P – Presentation-oriented

A Type P connection connects two application processes. The client process is specialized on presentation and has no direct access to business data. Instead, the server transforms data objects into a presentation format before it sends data to the client. The transformation effectively removes object structure and identity. The client can update predefined fields of the presentation and post their values to the server, which has to associate the presentation-oriented changes with data objects again. Examples are HTML(-forms) over HTTP and terminal protocols.

3.3. Type R – Remote Data Objects

The server process of a Type R connection represents data objects as remote objects. Remote objects have a fixed location and clients can access their attribute values via remote invocations. Examples are (pure) CORBA and RMI access to EJB entity beans [8], [7].

3.4. Type A – Application-specific Facades for Data-shipping

A Type A connection connects two application processes. The server exposes an application-specific facade to the client. Clients cannot directly access data objects on the server (and thus the original object-oriented data model), instead they talk to the facade which implements application-specific data-shipping operations: The object states are extracted, transformed into a format for shipping (often proprietary or XML), and then
copied to the client. The following listing shows an example interface (RMI) for such an application-specific facade.

```java
// for shipping values of a Customer instance
public class CustomerRecord implements java.io.Serializable {
    ...
}

public interface OrderSystemFacade extends Remote {
    CustomerRecord[] getCustomersByName(String pattern, int maxHits)
        throws RemoteException, DatastoreEx;
    OrderRecord[] getOrdersByCustomer(String customerId)
        throws RemoteException, DatastoreEx;
    OrderRecord[] getOrdersByDate(Date from, Date to, int page, int ordersPerPage)
        throws RemoteException, DatastoreEx;
    void updateCustomer(CustomerRecord c)
        throws RemoteException, NotFoundEx, UpdateConflictEx, DatastoreEx;
    void insertCustomer(CustomerRecord c)
        throws RemoteException, NotFoundEx, AlreadyExistsEx, DatastoreEx;
    void deleteCustomer(CustomerRecord c) ...
    ... (other methods) ...
}
```

In addition to data-shipping operations the facade usually implements application-specific operations for inserting, updating, and deleting data objects. Examples of Type A connections are CORBA facades to local data objects, RMI + EJB session beans, application-specific messages sent via messaging systems, and low-level, application-specific communication via sockets.

There is a subtle but important difference between types R and A. Both can be implemented on top of distributed object middleware. However, while Type R exposes data objects as remote objects, Type A treats them as local objects and copies their state by value.

### 3.5. Limitations of Existing Connections

In general, connections of Type D and P are well-understood and have mature implementations. In fact, three-tier architectures based on P/D combinations are the first choice in many projects because of the simplicity of the approach. Unfortunately, Type P connections and their thin clients (web browsers, terminals) are not an option for many applications that need complex, sophisticated, user-friendly, and highly interactive GUIs. Also, P/D combinations are not a solution for applications that employ topology patterns and require custom topologies.

Type R connections are comfortable – but object-oriented, navigational client access to remote data objects typically leads to communication that is too fine-grained. For instance, a fat client that displays a set of data objects as a table can easily perform hundreds of
remote invocations just for displaying a single view to a single user. This results in high network traffic, bad response time, high server load, and thus limited scalability [10].

Often, this leaves only Type A for connections that cannot be covered by D or P. Unfortunately, efficient implementations of Type A connections tend to get very complex since application developers have to deal with many aspects that are generally regarded as infrastructure issues, for instance:

- Client side caching of objects,
- managing identity of objects on the client side: Objects copied by value from the server should be mapped to the same entity if they represent the same data object,
- integration of client data and client operations into the server side transaction management,
- mechanisms for handling large query results that avoid copying the complete result,
- synchronization of stale data.

Most application developers are not prepared to handle these infrastructure issues, which makes Type A connections a risk for many projects. An additional drawback is that developers have to maintain application-specific facades and keep them in sync with the server’s object-oriented data model.

In principle, Types R and A can be combined, which results in a data model that consists of first class remote objects and second class value objects, which depend on their first class objects. This approach is a compromise, but it combines both the advantages and disadvantages.

While the limitations of Type A connections make it hard to build custom topologies on top of them, it is even harder to adapt such topologies to changing requirements. The insufficient separation of application and infrastructure concerns usually makes it necessary to modify and redesign large parts of an enterprise application when topology patterns have to be applied, modified, or removed.

4. **Overview of the FPT Architecture**

In this section, we outline principles of an enterprise application architecture that specifically addresses the problems motivated above.

Our *flexible process topology (FPT) architecture* specifies a data-centric infrastructure for object-oriented enterprise applications. The architecture has the following two main goals:

(a) Support for arbitrary DAGs of distributed processes as underlying topologies. For each enterprise application a custom-made topology can be designed that exactly matches application-specific requirements.

(b) Flexible topologies – topologies are easy to adapt, because application code, process topology, and data distribution scheme are decoupled.

4.1. **Approach**

The basic idea behind the FPT architecture is to place a generic *object manager* component in each application process of the topology as depicted in Figure 3. An object manager
maintains and transparently manages connections to object managers of connected client and server processes (these connections are called object manager connections, or Type O connections),

- offers to local application code an interface for transactional, object-oriented access to business data objects, including object queries and navigational access,
- internally represents business data objects as value objects that are copied (not moved) across process boundaries,
- transparently loads, stores, and synchronizes business data objects from/with connected object managers, and
- acts as a cache for business data objects that services local application code and client object managers.

4.2. Decoupling

Application code is decoupled from process topology and data distribution scheme because it accesses data objects exclusively through the local object manager that transparently loads objects by value via Type O connections. Type O connections are a matter of object manager configuration and hidden from the application. Optimizations, like bulk transfer of object state, object caching [3], query caching, or pre-fetching [1], are also handled by object managers, to clearly separate application and infrastructure issues.
The DAG of processes of an enterprise application corresponds to a DAG of object managers, which cooperate to access data objects persistently stored in datastores. A distribution scheme for data objects in datastores, including replication, is defined by RI-trees, which are discussed in the Section 5. Object managers are responsible for routing data objects and queries via other object managers to the appropriate datastores. To decouple data distribution from topology, object managers use a routing mechanism that takes topology and distribution scheme as parameters and produces a correct routing for all combinations.

4.3. FPT Transactions

Application code can set boundaries of FPT transactions and transactionally access data objects, i.e. perform insert, update, delete, and query operations. The local object manager fetches all objects accessed (and not present in the local cache) from connected server processes, which in turn can fetch them from their servers. Internally, each object has a version number, which is incremented whenever a new version is made persistent. Cached data objects may become stall, i.e. their version number is smaller than the version number of the corresponding entry in the datastore. Object managers keep track of each object’s version number and, in addition, of one or more home datastores that store the (possibly replicated) object. For replicated objects, which are accessed according to a “read-one-write-all” (ROWA) scheme, it is possible that only a subset of home datastores is known to an object manager, unless it explicitly queries other datastores that may store the object. For efficient navigational access, one, a subset, or all of the home datastores can be stored as part of object references.

When a transaction changes data objects, the local object manager transparently creates new private versions in the local cache. Transaction termination is initiated by a commit call and consists of two phases: (1) push-down and (2) distributed commit. In phase 1 all private versions are propagated “down” the DAG topology via Type O connections to application processes that can directly access the corresponding datastores. In phase 2, when all involved datastores have been determined, a (low-level) distributed database transaction is initiated, all propagated object versions are stored to their home datastores, and a distributed commit protocol is executed, for example two-phase commit.

To exploit caching and to relieve datastores and connections of fine-grained lock requests, an optimistic concurrency control scheme is used: In phase 2, before new versions are stored, the version number of each new version is checked against the version number stored with the corresponding datastore entry to detect conflicts with other transactions.

Due to space restrictions, we can only outline the FPT architecture – many aspects that deserve and require a detailed discussion, like isolation properties, vertical distribution of business logic, cache synchronization, “hot spot” data objects, fault tolerance, and various optimization techniques cannot be addressed in this paper. Instead, we focus on our approach to decouple process topology and data distribution scheme. The following section introduces the RI-tree, which is the basis for separating these two concerns.
5. RI-Trees

We use RI-trees for specifying a replication and distribution scheme for data objects. We assume that the set of all possible data objects is partitioned into one or more disjoint domains, for instance

- **Domain 1**: All data objects of type *Customer*
- **Domain 2**: All Orders with date < 1/1/2002
- **Domain 3**: All Orders with date ≥ 1/1/2002.

For each domain a separate RI-tree is specified that describes where data objects of that domain can be stored. For simplicity we will focus on a single domain and the corresponding tree only.

An RI-tree is a tree whose leaf nodes represent different transactional datastores and whose inner nodes are either R-nodes ("replication") or I-nodes ("integration"). Intuitively and informally, an R-node means that an object is replicated and placed in all sub-trees of the node. An I-node means that an object is placed in exactly one sub-tree of the node. Please note that R-nodes and I-nodes do not correspond to processes in a topology. In fact, RI-trees and process topology are orthogonal concepts.

![RI-tree example](image)

Figure 4. Example of an RI-tree.

Figure 4 shows an example of an RI-tree that specifies replication and distribution among six datastores labeled *ds*₂, *ds*₃, *ds*₅, *ds*₆, *ds*₈, and *ds*₉. The given tree specifies that there are three options for storing an object, each option defines a possible set of home datastores:

- **Option 1**: home datastores 2, 3, and 5
- **Option 2**: home datastores 2, 3, and 6
- **Option 3**: home datastores 8 and 9

For each newly created object one option, which defines the object’s home datastores, has to be selected. Then the object has to be stored in all these home datastores (replication). Once an option has been selected for an object, its home datastores cannot be changed. Note that partitioning of objects can be achieved either by creating domains or by using I-nodes (or both). While domains need a pre-defined criterion based on object type and/or attribute values, I-nodes are more flexible and allow new objects to be inserted, e.g., following a load balancing, round robin, random, or fill ratio based scheme.
Formally, an RI-tree \( T \) is a rooted tree \((V, E)\), \( V \) is the vertex set and \( E \) the edge set. \( V = \text{RNODES} \cup \text{INODES} \cup \text{DATASTORES} \). The three sets \text{RNODES}, \text{INODES}, and \text{DATASTORES} are pairwise disjoint. \text{DATASTORES} is a non-empty subset of the set of all datastores \( DS_{all} = \{ds_1, ds_2, ..., ds_{\text{maxds}}\} \). \( T \)'s inner nodes are \text{RNODES} \cup \text{INODES}, its leaf nodes are \text{DATASTORES}. Now we introduce a function \( \text{options} \) that maps each element of \( V \) to a subset of \( 2^{DS_{all}} \). For each RI-tree \( T \) its options are the result of a function \( \text{options(root}(T)) \), or short: \( \text{options}(T) \). \( \text{options} \) is recursively defined as follows:

\[
\text{options}(u) = \left\{ \begin{array}{ll}
\{M_1 \cup M_2 \cup ... \cup M_m | (M_1, M_2, ..., M_m) \\
\in \text{options}(v_1) \times \text{options}(v_2) \times ... \times \text{options}(v_m), \\
\text{where } \{v_1, v_2, ..., v_m\} = \{v | (u, v) \in E\} \}
\end{array} \right.
\]

for \( u \in \text{RNODES} \)

\[
\bigcup_{w \in \{v | (u, v) \in E\}} \text{options}(w)
\]

for \( u \in \text{INODES} \)

\[
\{\{u\}\}
\]

for \( u \in \text{DATASTORES} \)

The following example shows the formal representation of and options for the RI-tree from Figure 4:

\[ T = (V, E) \]
\[ V = \{u_1, u_2, u_3, u_4, ds_2, ds_3, ds_5, ds_6, ds_8, ds_9\} \]
\[ E = \{(u_1, u_2), (u_2, u_4), (u_1, u_3), (u_2, ds_2), (u_2, ds_3), (u_4, ds_5), (u_4, ds_6), (u_3, ds_8), (u_3, ds_9)\} \]
\[ \text{DATASTORES} = \{ds_2, ds_3, ds_5, ds_6, ds_8, ds_9\} \]
\[ \text{RNODES} = \{u_2, u_3\} \]
\[ \text{INODES} = \{u_1, u_4\} \]
\[ \text{root}(T) = u_1 \]
\[ \text{options}(u_4) = \{\{ds_5\}, \{ds_6\}\} \]
\[ \text{options}(u_2) = \{\{ds_2, ds_3, ds_5\}, \{ds_2, ds_3, ds_6\}\} \]
\[ \text{options}(u_3) = \{\{ds_8, ds_9\}\} \]
\[ \text{options}(T) := \text{options}(\text{root}(T)) = \{\{ds_2, ds_3, ds_5\}, \{ds_2, ds_3, ds_6\}, \{ds_8, ds_9\}\} \]

6. **Object Routing**

Having introduced the RI-tree formalism in the previous section, we can now discuss how RI-trees and process topology are related and how objects can be routed.

6.1. **Imports and Exports**

For each data domain \( \text{dom}_i \) a separate RI-tree \( T_i \) defines which datastores are potential home datastores for objects of the given domain. We propose a simple import/export scheme for process topologies that helps us to determine which domains and datastores can be accessed by which processes.

Each process in the DAG is assigned an attribute labeled \( \text{exports} \) and each communication relationship (edge) is assigned an attribute labeled \( \text{imports} \). Both attributes contain a set of tuples of the form \( (\text{domain}, \text{datastore}) \) as values. Each tuple represents the ability to access objects of a \( \text{domain} \), which are stored in a specific \( \text{datastore} \). Access can be either direct or indirect via other processes. Figure 5 illustrates an example of an import/export scheme for
three domains in a process topology with two datastores. In an FPT enterprise application import/export rules for a process and its connections to server processes are part of the configuration data of that process.

For each application process \( p \) and domain \( \text{dom}_i \) we define \( \text{canAccess}(p, \text{dom}_i) := \{ ds_k | (ds_k, \text{dom}_i) \text{ is element of the union of all imports of } p's \text{ outgoing edges} \} \). \( \text{canAccess} \) is the set of datastores that can be accessed (directly or indirectly) by \( p \) for a given domain \( \text{dom}_i \).

### 6.2. Formal Definition of Object Routing

When an application process \( \text{root} \) transactionally accesses a set of objects, i.e. performs insert, update, delete, and query operations, all objects in the set are copied by value to \( \text{root}'s \) cache and new versions are created locally for changed objects. On commit all new versions have to be propagated “down” the DAG topology via client/server connections and applied to datastores. A changed object \( o \) is either

- \emph{new}, i.e. it has been inserted by the application but not yet persistently stored, or
- \emph{existing}, i.e. an option \( \text{home}(o) \) has already been selected and a previous version of the object has been stored in the corresponding home datastores.

New objects have to be assigned an option \( \text{home}(o) \) and have to be stored in all home datastores defined by that option. For existing objects (updated or deleted) all home datastores have to be identified and accessed. The process of propagating changed objects through the topology is called \emph{object routing}.

Formally, an object routing \( R \) for an object \( o \) of domain \( \text{dom}_i \) is defined by an application process \( \text{root} \), a set of datastores \( \text{rtargets} \) and a set of paths \( \text{rpaths} \). For each datastore \( ds \in \text{rtargets} \) there is a path in \( \text{rpaths} \) that starts at \( \text{root} \) and ends at \( ds \). Each path in \( \text{rpaths} \) ends at a datastore \( ds \in \text{rtargets} \) and all edges in the path must contain \( (\text{dom}_i, ds) \) in their \emph{imports}.

![Figure 5. Example of an import/export scheme for process topologies.](image-url)
For an existing object \( o \) we call an object routing \textit{correct} iff \( rtargets = \text{home}(o) \), i.e. the object is routed to all its home datastores. For a new object \( o \) we call an object routing \textit{correct} iff \( rtargets \in \text{options}(T_i) \), i.e. a valid option is selected and the object is routed to all home datastores of that option.

### 6.3. A Conceptual Framework for Object Routing

A simple approach for object routing would let the root process locally pre-calculate an object routing \( R \) on commit. This simple approach has two important disadvantages:

- Instead of knowing only processes that are directly connected, the root process would have to be aware of the structure of the complete underlying topology.
- An object routing can be optimized with regard to various parameters, for example, current and maximum load of processes and datastores, clustering of objects in datastores, bandwidth of connections, or fill ratio of datastores. Each potential root process would have to obtain and track these parameters of other processes, which does not scale for many client processes.

Instead of suggesting one of countless possible optimization techniques for object routing we present a conceptual framework that is independent of specific optimization parameters and decisions. To preserve the autonomy of subsystems we only require each process to know its direct server processes and all RI-trees. We propose that an object routing is calculated incrementally during the propagation process and decisions are deferred for as long as possible/necessary. Each process recursively delegates routing decisions to its server process(es) unless RI-tree, import/export schema, and topology require a local decision.

Each application process \( p \) involved in an object routing performs a \textit{local routing} (see Figure 6) as follows:

1. With a \textit{routing message} a client \( c \) propagates an object \( o \) of domain \( \text{dom}_i \) together with a set of datastores (represented by ids) \( \text{homeCandidatesIn} \) to \( p \).
2. Let \( p \) have \( s \) servers \( \text{child}_1 \ldots \text{child}_s \). \( p \) uses a \textit{local routing function} to calculate \( s \) sets of datastores \( \text{homeCandidatesOut}_1 \ldots \text{homeCandidatesOut}_s \), one for each server of \( p \). Each set \( \text{homeCandidatesOut}_k \) must be a subset of \( \text{homeCandidatesIn} \). In addition, for each datastore \( ds \) in \( \text{homeCandidatesOut}_k \) there has to be an element \((\text{dom}_i, ds)\) in the \textit{imports} of edge \((p, \text{child}_k)\).
3. For each non-empty set \( \text{homeCandidatesOut}_k \) the following is done:
   - if \( \text{child}_k \) is an application process then \( p \) propagates \( o \) and \( \text{homeCandidatesOut}_k \) to its server \( \text{child}_k \) with a routing message. Each \( \text{child}_k \) in turn performs a local routing for \( o \) and takes \( \text{homeCandidatesOut}_k \) as its \( \text{homeCandidatesIn} \) input parameter.
   - if \( \text{child}_k \) is a datastore then \( p \) temporarily stores \( o \) and \( \text{child}_k \) for phase 2 (in which \( o \)’s version number is checked and \( o \) is stored to \( \text{child}_k \) – see Subsection 4.3).
4. Replies for routing messages sent in 3. are collected and a reply message is sent to client \( c \).
The local routing function can be application-specific and even process-specific as it is independent of routing functions of other processes. A root process skips step 1: \( o \) is determined by the current transaction and \( \text{homeCandidatesIn} \) is set to \( \{ ds \mid \text{there is an } e \in \text{options}(T_i) \text{ with } ds \in e \land e \subseteq \text{canAccess}(p, \text{dom}_i) \} \). Constraints for root processes and rules for local routing functions depend on whether \( o \) is new or existing:

**Routing a New Object**

A root process requires that \( \text{options}(T_i) \) contains an element \( e \) with \( e \subseteq \text{canAccess}(p, \text{dom}_i) \). A correct routing can be guaranteed when the local routing functions of all involved processes observe the following rules:

(a) \( \text{homeCandidatesOut}_1 \ldots \text{homeCandidatesOut}_s \) are pairwise disjoint.

(b) For each pair \( (x, y) \) of datastores \( x \in \text{homeCandidatesOut}_m, y \in \text{homeCandidatesOut}_n \), and \( m \neq n \) their lowest (deepest) common ancestor in the RI-tree \( T_i \) is an R-node.

(c) Let \( \text{del} := \{ ds \mid ds \in \text{homeCandidatesIn}, \text{but } ds \text{ is not included in the union of all } \text{homeCandidatesOut}_1 \ldots \text{homeCandidatesOut}_s \} \). For each \( ds \) in \( \text{del} \) there must be a node \( x \) in the RI-tree \( T_i \) so that

- \( x \) is ancestor of \( ds \) or \( x = ds \),
- \( x \) is child of an I-node \( y \),
- all descendants of \( x \) (including \( x \) itself) that are included in \( \text{homeCandidatesIn} \) are also included in \( \text{del} \),
- and there is at least one element in \( \text{homeCandidatesIn} \) that is not included in \( \text{del} \) and is a descendant of \( y \) in \( T_i \).
The example in Figure 7 shows an RI-tree and two different correct routings (both with process A as root) in the same topology. For simplicity we assume a maximum import/export scheme, i.e. all datastores export dom, everything an application process can export is exported, and everything an application process can import is imported. Solid arrows between application processes indicate the path of routing messages in transaction phase 1. A solid arrow from an application process to a datastore indicates access to that datastore in phase 2. Dotted arrows represent edges not used in the routing. The value of the parameter homeCandidatesIn for each involved application process is shown above the rectangle that represents the corresponding process.

![RI-tree T_i](image)

Figure 7. Two different correct routings for the same newly created data object.

Please note that the examples in Figure 7 and Figure 8 are not real world examples. Their RI-trees are complex and intentionally do not match the topology so that they clearly demonstrate how object routing works. In realistic scenarios we expect typical RI-trees to have only one or two levels of inner nodes.

**Routing an Existing Object**

A root process requires that home(o) ∈ canAccess(p, dom), although home(o) itself may not be known to the root process and other processes. We suppose that p knows a set homeConfirmed ⊆ homeCandidatesIn, which contains a subset of o’s home datastores (see Subsection 4.3). homeConfirmed may but is not required to be included as a
A correct routing can be guaranteed when the local routing functions observe the following rule – in addition to rules (a) and (b) given for new objects:

(c) Let \( \text{del} := \{ ds \mid ds \in \text{homeCandidatesIn}, \text{but} ds \text{ is not included in the union of all } \text{homeCandidatesOut}_1, \ldots, \text{homeCandidatesOut}_s \} \). For each \( ds \in \text{del} \) there is no element \( e \in \text{options}(T_i) \) so that \( ds \in e \land \text{homeConfirmed} \subseteq e \).

In some cases, especially when \( o \) is a replicated object and \( T_i \) contains I-nodes, an application process may face the situation that it cannot immediately produce a correct routing. In that case, one or more probe queries are sent to server processes (sequentially or in parallel) to confirm or rule out that datastores from \( \text{homeCandidatesIn} \setminus \text{homeConfirmed} \) are home datastores of \( o \):

- Each confirmed datastore \( ds_{conf} \) is added to \( \text{homeConfirmed} \).
- For each datastore \( ds_{notfound} \) that is reported not to be a home datastore, \( ds_{notfound} \) and all elements of \( \{ ds \mid ds \in \text{homeCandidatesIn}, ds \neq ds_{notfound}, \text{the lowest (deepest) common ancestor } x \text{ of } ds_{notfound} \text{ and } ds \text{ in } T_i \text{ is an R-node, and all nodes between } ds_{notfound} \text{ and } x \text{ are either I-nodes with only one child or R-nodes} \} \) are removed from \( \text{homeCandidatesIn} \).

Eventually, a correct routing can be produced – at the latest when \( \text{homeCandidatesIn} = \text{homeConfirmed} \). Probe queries can be expensive, especially when \( \text{options}(T_i) \) contains many options and each of these options contains many datastores.

Figure 8 illustrates a scenario for routing an existing object where \( A \), the root process, has to perform a probe query first to produce a correct routing. The existing object’s home datastores are \( ds_2, ds_3, ds_5, ds_6 \), but only \( ds_3 \) is initially known. Again, we assume a maximum import/export scheme.

So far, only routing of a single object has been discussed. When a transaction accesses multiple objects then complete sets of objects are to be routed. Often it is possible to route
them together using only a few coarse-grained messages for inter-process communication. On the way “down” the DAG topology sets may be split into smaller subsets that are treated differently and routed individually. Ideally, all objects can be routed as one set to a single datastore – in that case no distributed commit protocol is required.

When Pattern 6 (mesh, see Subsection 2.1) has been applied then a process may receive two or more routing messages for the same data object under certain circumstances. This is not a problem, since all these messages can be processed independently. To integrate two systems with Pattern 5, the data distribution schemes of both systems have to be merged: The new scheme simply consists of all domains and RI-trees of both data distribution schemes (union). Special treatment is needed for domains that exist in both systems (for example, each system stores Customer objects and both sets have to be integrated). Provided there are no duplicates or these are removed first, a new RI-tree is constructed by placing an I-node as a new root on top of both previous RI-trees for the domain.

7. Related Work

To our knowledge, neither data-centric custom process topologies nor the flexibility aspect of these topologies have been discussed in the context of enterprise applications before.

The distributed objects paradigm, especially CORBA [5], offers the powerful concept of transparencies, including access and location. But it does not sufficiently address the fact that in many topologies inter-process access to data objects is restricted and most processes are only allowed to directly communicate with a small subset of other processes in the topology (see Section 2).

TopLink [11], an object-relational mapping framework, supports so called remote sessions that are similar to Type O connections (see Subsection 4.1) but can only be employed to connect TopLink clients to TopLink servers. Since client and server roles are fixed and only one server per client is supported, remote sessions are rather limited and cannot be used to build arbitrary process topologies.

Research in the context of peer-to-peer networks focuses on flexible topologies and routing (for example [6]), but usually objects are coarse-grained and access is both read-only and non-transactional.

8. Summary and Conclusion

We view enterprise applications as topologies of distributed processes that access business data objects persistently stored in transactional datastores. We explained why many large-scale applications need custom topologies to address their application-specific requirements, e.g., regarding scalability and fault tolerance. There are several well-known topology patterns for custom topologies, which, when combined, lead to arbitrary DAG topologies. We categorized connections offered by current middleware and explained why it is difficult to build DAG topologies on top of them.

Then we outlined principles of our FPT architecture for object-oriented, data-centric enterprise applications with arbitrary DAGs as underlying process topologies. The
architecture is based on a network of object manager components which cooperate to access data objects. In contrast to existing middleware topologies are flexible, i.e. easy to adapt to changing requirements, because application, topology, and data distribution scheme are decoupled and can be specified independently. We introduced RI-trees for specifying a data distribution scheme (including replication) and a conceptual framework for RI-tree–based object routing in DAG topologies.

The framework does not define a specific routing strategy, instead only the general approach and constraints for correct object routing are given. A local routing function can be specified separately for each node in the topology. These functions typically have a large set of routing possibilities from which they can select one according to their (private) optimization criteria. This allows a broad range of application-specific optimization techniques to be integrated. For all topologies and RI-trees a correct routing can be produced, provided that the corresponding domains/datastores can be reached by a root process. The enterprise application can even tolerate the loss of redundant processes and connections at runtime when the corresponding tuples are removed from the import/export scheme.

The fact that arbitrary topologies and RI-trees are supported does not mean that all combinations necessarily lead to efficient systems. For example, extensive use of replication always has an impact on performance. Selecting efficient topologies and distribution schemes for an application is still a challenging task and up to software architects. But once these decisions are made, our architecture significantly simplifies the development of an enterprise application and improves maintainability by factoring out the topology aspect.

We view the FPT architecture as a set of concepts that can either be used for extending existing enterprise application frameworks or be used as a basis for new frameworks. Currently, we follow the latter approach and are working on a framework based on Java, RMI, relational databases as datastores, and xa transactions. A first prototype has already been completed. Future work includes a detailed performance and scalability analysis, case studies, and a broad range of optimizations for queries, routing, and synchronization in flexible DAG topologies.

References


[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995


An Architecture for the UniFrame Resource Discovery Service

Nanditha N. Siram, Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University Purdue University Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202-5132, USA
Email: {nnayani, rrale, aolson}@cs.iupui.edu

Barrett R. Bryant, Carol C. Burt
Department of Computer and Information Sciences
The University of Alabama at Birmingham
115A Campbell Hall, 1300 University Boulevard
Birmingham, AL 35294-1170, USA
Email: {bryant, cburt}@cis.uab.edu

Mikhail Auguston
Department of Computer Science
New Mexico State University
PO Box 30001, MCS CS
Las Cruces, NM 88003
Email: mikau@cs.nmsu.edu

Abstract

Frequently, the software development for large-scale distributed systems requires combining components that adhere to different object models. One solution for the integration of distributed and heterogeneous software components is the UniFrame approach. It provides a comprehensive framework unifying existing and emerging distributed component models under a common meta-model that enables the discovery, interoperability, and collaboration of components via generative software techniques. This paper presents the architecture for the resource discovery aspect of this framework, called the UniFrame Resource Discovery Service (URDS). The proposed architecture addresses the following issues: a) dynamic discovery of heterogeneous components, and b) selection of components meeting the necessary requirements, including desired levels of QoS (Quality of Service). This paper also compares the URDS architecture with other Resource Discovery Protocols, outlining the gaps that URDS is trying to bridge.

1. Introduction

Software realizations of distributed-computing systems (DCS) are currently being based on the notions of independently created and deployed components, with public interfaces and...
private implementations, loosely integrating with one another to form a coalition of distributed software components. Assembling such systems requires either automatic or semi-automatic integration of software components, taking into account the quality of service (QoS) constraints advertised by each component and the collection of components. The UniFrame Approach (UA) [12][13] provides a framework that allows an interoperation of heterogeneous and distributed software components and incorporates the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [11]), b) an integration of QoS at the individual component and distributed system levels, c) the validation and assurance of QoS, based on the concept of event grammars, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available choices. The UniFrame approach depends on the discovery of independently deployed software components in a networked environment. This paper describes an architecture, URDS (UniFrame Resource Discovery Service), for the resource discovery aspect of UniFrame. The URDS architecture provides services for an automated discovery and selection of components meeting the necessary QoS requirements. URDS is designed as a Discovery Service wherein new services are dynamically discovered while providing clients with a Directory style access to services. The result of using URDS, the UA and its associated tools is a semi-automatic construction of a distributed system.

The rest of the paper is organized as follows. Section 2 discusses related resource discovery protocols. Section 3 discusses the UniFrame approach and the URDS architecture. An example is presented in section 4. A brief comparison of URDS and other approaches is presented in section 5. Details of an initial prototype and experimentations are indicated in section 6 and the paper concludes in section 7.

2. Related Work

The protocols for resource discovery can be broadly categorized into: a) **Lookup (Directory) Services and Static Registries** and b) **Discovery Services**. A few prominent approaches are briefly discussed below.

**Universal Description, Discovery and Integration (UDDI) Registry:** UDDI [16] specifications provide for distributed Web-based information registries wherein Web services can be published and discovered. Web Services in UDDI are described using Web Services Description Language (WSDL) [4] -- an XML grammar for describing the capabilities and technical details of Simple Object Access Protocol (SOAP) [1] based web services.

**CORBA Trader Services:** The CORBA Trader Service [10] facilitates ‘matchmaking’ between service providers (Exporters) and service consumers (Importers). The exporters register their services with the trader and the importers query the trader. The trader will find a match for the client based on the search criteria. Traders can be linked to form a *federation of traders*, thus making the offer spaces of other traders implicitly available to its own clients.

**Service Location Protocol (SLP):** SLP [6] architecture comprises of *User Agents (UA), Service Agents (SA), and Directory Agents (DA)*. UAs perform service discovery on behalf
of clients, SAs advertise the location and characteristics of services and DAs act as directories which aggregate service information received from SAs in their database and respond to service requests from UAs. Service requests may match according to service type or by attributes.

**JINI:** JINI [15] is a Java-based framework for spontaneous discovery. The main components of a JINI system are Services, Clients and Lookup Services. A service registers a “service proxy” with the Lookup Service and clients requesting services get a handle to the “service proxy” from the Lookup Service.

**Ninja Secure Service Discovery Service (SSDS):** The main components of the SSDS [5], [9] are: Service Discovery Servers (SDS), Services and Clients. SSDS shares similarities with other discovery protocols, with significant improvements in reliability, scalability, and security.

### 3. UniFrame and UniFrame Resource Discovery Service (URDS)

The Directory and Discovery Services, described earlier, mostly do not take advantage of the heterogeneity, local autonomy and the open architecture that are characteristics of DCS. Also, a majority of these systems operate in one-model environment (e.g., CORBA Trader service assumes only the presence of CORBA components). In contrast, a software realization of a DCS will most certainly require a combination of heterogeneous components – i.e., components developed under different models. In such a scenario, there is a need for a discovery system that exploits the open nature, heterogeneity and local autonomy inherent in DCS. The URDS architecture is one such solution for the discovery of heterogeneous and distributed software components.

#### 3.1. UniFrame Approach

##### 3.1.1. Components, Services and QoS

**Components** in UniFrame are autonomous entities, whose implementations are non-uniform. Each component has a state, an identity, a behavior, well-defined public interfaces and private implementation. In addition, each component has three aspects: a) **Computational Aspect:** it reflects the task(s) carried out by each component, b) **Cooperative Aspect:** it indicates the interaction with other components, and c) **Auxiliary Aspect:** this addresses other important features of a component such as security and fault tolerance.

**Services,** offered by a component in UniFrame, could be an intensive computational effort or an access to underlying resources. The **QoS** is an indication given by a software component about its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures.

##### 3.1.2. Service Types

Components in UniFrame are specified informally in XML using a standard format. XML [3] is selected as it is general enough to express the required concepts, it is rigorously specified, and it is universally accepted and deployed. The UniFrame service type, which represents the information needed to describe a service, comprises of:
ID: A unique identifier comprising of the host name on which the component is running and the name with which this component binds itself to a registry will identify each service.

ComponentName: The name with which the service component identifies itself.

Description: A brief description of the purpose of this service component.

Function Descriptions: A brief description of each of the functions supported by the service component.

Syntactic Contracts: A definition of the computational signature of the service interface.

Function: Overall function of the service component.

Algorithm: The algorithms implemented by this component.

Complexity: The overall order of complexity of the algorithms implemented by this component.

Technology: The technology used to implement this component (e.g., CORBA, Java RMI, etc.).

<UniFrame>
  <ComponentName>AccountServer</ComponentName>
  <Description>Provides an Account Management System</Description>
  <FunctionDescription>
    <Function>javaDeposit</Function>
    <Function>javaWithdraw</Function>
    <Function>javaBalance</Function>
  </FunctionDescription>
  <ComputationalAttributes>
    <InherentAttributes>
      <ID>intrepid.cs.iupui.edu/AccountServer</ID>
    </InherentAttributes>
  </ComputationalAttributes>
  <FunctionalAttributes>
    <Function>Acts as Account Server</Function>
    <Algorithm>Simple Addition/Subtraction</Algorithm>
    <Complexity>O(1)</Complexity>
    <SyntacticContract>
      <Contract>void javaDeposit(float ip)</Contract>
      <Contract>void javaWithdraw throws OverDrawException</Contract>
      <Contract>float javaBalance()</Contract>
    </SyntacticContract>
    <Technology>Java-RMI</Technology>
  </FunctionalAttributes>
  <CooperatingAttributes>
    <PreprocessingCollaborators>AccountClient</PreprocessingCollaborators>
  </CooperatingAttributes>
  <AuxiliaryAttributes>
    <Mobility>No</Mobility>
  </AuxiliaryAttributes>
  <QOSMetrics>
    <Availability measure="99">90</Availability>
    <End2EndDelay measure="ms">10</End2EndDelay>
  </QOSMetrics>
</UniFrame>

Figure 1: Sample UniFrame Specification in XML
QoS Metrics: Zero or more Quality Of Service (QoS) types. The QoS type defines the QoS value type. Associated with a QoS type is the triple of \(<QoS\text{-type-name}, \text{measure, value}>\) where QoS\text{-type-name} specifies the QoS metric, for example, throughput, capacity, end-to-end delay, etc. Measure indicates the quantification parameter for this type-name like methods completed/sec, number of concurrent requests handled, time, etc. Value indicates a numeric/string/boolean value for this parameter. We have established a catalog of Quality of Service metrics that are used in UniFrame specifications [2].

Figure 1 illustrates a sample UniFrame specification. This example is for a bank account management system with services for deposit, withdraw, and check balance. This example assumes the presence of a Java RMI server program and a CORBA server program, which are available to interact with the client requesting their services. We will return to this example in detail when we describe the resource discovery service.

3.2 URDS

The main components of the URDS architecture (illustrated in Figure 2) are: i) Internet Component Broker (ICB), ii) Headhunters (HHs), iii) Meta-Repositories, iv) Active-Registries, v) Services, and vi) Clients. Other details in the figure will be explained in the following sections. The numbers indicate the flow of activities in the URDS. These are explained, in detail, in the context of an example in section 3.2.7. The URDS architecture is organized as a federation in order to achieve scalability. Figure 3 illustrates the federation aspect of URDS.

Every ICB has zero or more headhunters attached to it. The ICBs in turn are linked together with unidirectional links to form a directed graph. The URDS discovery process is “administratively scoped”, i.e., it locates services within an administratively defined logical domain. ‘Domain’ in UniFrame refers to industry specific markets such as Financial Services, Health Care Services, Manufacturing Services, etc.

Figure 2: URDS Architecture
3.2.1 Internet Component Broker (ICB)

The ICB acts as an all-pervasive component broker in the interconnected environment providing a platform for the discovery and seamless integration of disparate components. The ICB is not a single component but is a collection of services comprising of the Query Manager (QM), the Domain Security Manager (DSM), Adapter Manager (AM), and the Link Manager (LM). It is envisioned that there will be a fixed number of ICBs deployed at well-known locations hosted by corporations or organizations supporting this initiative. The functionality of the ICB is similar to that of an Object Request Broker. However, the ICB has certain key features that are unique. It provides component mappings and component model adapters. The ICB, in conjunction with headhunters, provides the infrastructure necessary for scalable, reliable, and secure collaborative business using the interconnected infrastructure. The functionalities of the ICB are:

- Authenticate the users (Headhunters and Active Registries) in the system and enforce access control over the multicast address resources for a domain with the help of the Domain Security Manager (DSM).
- Attempt at matchmaking between service producers and consumers with the help of the Headhunters and Query Manager. ICBs may cooperate with each other in order to increase the search space for matchmaking. The cooperation techniques of ICBs are facilitated through the Link Manager (LM).
- Act as a mediator between two components adhering to different component models. The mediation capabilities of the ICB are facilitated through the Adapter Manager (AM).

**Domain Security Manager (DSM)**

The DSM handles secret key generation and distribution and enforces the group membership and access control to multicast resources through authentication and use of access control lists (ACL). The resources being guarded are the multicast addresses allocated to a particular *domain*. The DSM serves as an authorized third party, which maintains an inclusion list of Principals (headhunters or registries), corresponding to a
domain. DSM has an associated repository (database) of valid principals, passwords, multicast address resources and domains. Every Headhunter or Active Registry is associated with a domain. The Active Registries associated with a domain have components registered with them, which belong to that domain. The Headhunter in turn detects Registries, which belong to the same domain as itself, and hence the service components detected by the headhunter will belong to a particular domain. The Principal (authenticated user), is allowed access only to the multicast address mapped to the domain with which it is associated. A Principal that wishes to participate in the discovery process contacts the DSM with its credentials (id, password, domain). The DSM authenticates the principal and checks its authorizations against the domain ACL. The DSM returns a secret key and a multicast address mapped to the corresponding domain to a valid principal. In case the principal is a Headhunter the DSM registers the contact information of the headhunter with itself. The QM to propagate queries uses this information.

**Query Manager (QM)**

The QM uses a natural language parser [7] to translate a service consumer’s natural language-like query into an XML based query. The QM parses the XML based query to generate a structured query language statement and dispatches this query to the ‘appropriate’ Headhunters. The QM obtains the list of registered Headhunters from the DSM. The HH returns the list of matching service providers. The QM in conjunction with the LM is also responsible for propagating the queries to other linked ICBs. The functions performed by the QM are:

- Parse a service consumer’s natural language-like query and extract the keywords and phrases pertaining to various attributes of the components UniFrame specification.
- Extract the consumer-specified constraints, preferences and policies to be applied to the various attributes.
- Compose the extracted information into an XML based query.
- Translate the XML based query to a structured query language statement.
- Dispatch this structured query to all the headhunters associated with the domain on which the search is being performed and also forward the query to the Link Manager, which will propagate the query to other ICBs.
- The headhunters will query the Meta-Repository and return a list of components matching the search criteria to the QM.
- QM will wait for a specified time period for results to be returned from the headhunters/other ICBs before timing out.
- The client has the option to specify search-scoping policies to affect the time spent on the search process.
Link Manager (LM)

ICBs are linked to form a *Federation of Brokers* (see Figure 3) in order to allow for an effective utilization of the *distributed offer space*. ICBs propagate the search query issued by the Clients to other ICBs to which they are linked apart from the headhunters with which they are associated. The LM performs the functions of the ICB associated with establishing links and propagating the queries. *Links* represent paths for propagation of queries from a source ICB to a target ICB. The LM supports the following operations:

- **Register**: LMs register with each other to create unidirectional links from the Source LM to the Target LM. The registration information comprises of the location information of the LM.

- **Query**: The query operation is responsible for propagating the query from the source LM to the list of Target LMs with which the Source LM is registered.

- **Failure Detection**: This involves keeping track of LMs that may no longer be active due to failures. Periodically each LM sends a unicast message to all other LMs that are registered with it. LMs receiving the message maintain a cache of the pairs `<Sender LM address, Time-stamp of receipt>`. At regular time intervals the receiving LMs note the ‘freshness’ of the information they hold and purge the Sender’s information, which they deem to be ‘stale’. Staleness is determined by the time elapsed between the receipt of the LM address through the unicast communication and the current time.

- **Link Traversal Control**: The Link Traversal Control mechanism used in the LM is similar to that of CORBA Trader Services. The necessity for Link Traversal Control arises due to the nature of LM linkage, which allows arbitrary, directed graphs of LMs to be produced. This can introduce two problems: i) a single LM can be visited more than once, and ii) loops can occur. To ensure that a search does not enter into an infinite loop, a **hop count** is used to limit the depth of links to propagate a search. The hop count is decremented by one before propagating a query to other LMs. The search propagation terminates at the LM when the hop count reaches zero.

Adapter Manager (AM)

The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization (i.e., which heterogeneous component models they can bridge efficiently). Clients contact the AM to search for adapter components matching their needs. The AM utilizes adapter technology, each adapter component providing translation capabilities for specific component architectures. The adapter components achieve interoperability using the principles of wrap and glue technology [8].

3.2.2 Headhunters

Another critical component of URDS is a headhunter. The headhunters perform the following tasks: a) Service Discovery: detect the presence of service providers (Exporters),
b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements of the consumer (Importers) requests forwarded by the QM.

The service discovery process utilizes a search technique based on multicasting. Once deployed in the system, the headhunters periodically multicast their presence to a multicast group. The multicast group address is obtained from the DSM. The active registries, that also obtain a multicast group address from the DSM, listen for these multicast messages. The active registries maintain a cache of the pairs \(<\text{headhunter address, time-stamp of receipt}>\) and periodically send response messages to all the headhunters in their cache. The headhunter in turn maintains a cache of the pairs \(<\text{registry address, time-stamp of receipt}>\).

The Headhunter intermittently queries the Registries for the component information of service providers they contain. During the registration, the headhunter stores into the meta-repository all the details of the service providers, including the UniFrame specifications. The headhunter uses this information during matching. A component may be registered with multiple headhunters. The functionality of headhunters makes it necessary for them to communicate with Active Registries belonging to any model, implying that the cooperative aspect of headhunters be universal. The headhunters need to also address the issues of failures and security.

- **Failure Detection:** Failure detection involves keeping track of service exporters that may no longer be active in the system for various reasons. Headhunters achieve failure detection at the level of detecting failures of the active registries, which hold the service exporters. The headhunter keeps track of the time at which it obtains registry location information from various active registries. At regular time intervals the headhunter notes the ‘freshness’ of the information it holds and purges the registry information, which it deems to be ‘stale’. ‘Fresh’ or ‘Stale’ are determined based on the time elapsed between the receipt of the registry address through unicast communication and the current time. This process is based on the principle that if a registry is still active in the system, it will respond to the headhunter with its location information and thus have a recent timestamp. A registry which for whatever reason is unable to contact the headhunter with its information will hold a ‘stale’ timestamp and it will be assumed that all service exporter components held by this registry are no longer available for rendering their services.

- **Multicast Security:** This involves securing the multicast data transmission mechanism from security threats such as eavesdropping, and masquerading. The headhunter uses Secret Key Encryption to ensure security of transmitted data. The secret key used is a symmetric key wherein the sender and receiver use the same key for purposes of encryption and decryption.

### 3.2.3 Meta-Repository

The Meta-Repository is a data store that serves to hold service information of exporters adhering to different models. The service information stored by the Meta-repository consists of: a) Service type name, b) Details of its informal specification, and c) Zero or more QoS values for that service for each of the components. The implementation of a
Meta-Repository is database oriented. A Meta-Repository is a passive component, i.e., a headhunter brings information to the meta-repository.

3.2.4 Active Registry

The native registries (e.g., RMI Registry or CORBA registry) are extended to have the following features:

- **Activeness:** The registries are modified to be able to listen to multicast messages from the headhunter and respond with their host IP Address.

- **Introspection Capabilities:** The registries are extended to not only keep a list of component URLs of those components registered with them but also their detailed UniFrame specifications. This is achieved by querying the components (using principles of introspection) to obtain the URL of their XML based specifications. The registries parse the specification and maintain the details in a memory resident table, which is returned to the headhunter upon request.

- **Failure Detection Of Headhunters:** Failure detection involves keeping track of headhunters, which may no longer be active in the system for reasons such as network or node failure. The active registries keep track of the time at which it obtains headhunter location information from various headhunters through multicast. At regular intervals the active registries note the ‘freshness’ of the headhunter information they hold and purge the headhunter information, which they deem to be ‘stale’. ‘Fresh’ or ‘stale’ are determined based on the time elapsed between the receipt of the headhunter address through multicast communication and the current time.

3.2.5 Service Exporter Components

Service Exporter Components are implemented in different models, e.g., Java RMI, CORBA, EJB, etc. The components are identified by their Service Offers comprising of a) service type name, b) informal UniFrame specification, and c) zero or more QoS values for that service. The component registers its interfaces with its local registry. The component interface contains a method, which returns the URL of its informal specification. The informal specification is stored as a XML file adhering to certain syntactic contracts to facilitate parsing. These service exporter components will be tailored for specific domains, such as Financial Services, and will adhere to the relevant standards in those domains.

3.2.6 Clients

Clients are Service Requesters searching for services matching certain functional and non-functional requirements.

4. An Example

Table 1 outlines the interactions between the URDS components in servicing a client query for assembling an account management system. The rows of the table are numbered corresponding to the flow of control shown in Figure 2. The result of this interaction will be
an ensemble of components, which may be assembled into a complete system as described in [12].

Table 1: Interactions between URDS components

| 1 | This indicates the interactions between the principals (Headhunters/Active registries) and the DSM.  
  |   | • The principals contact the DSM with their authentication credentials in order to obtain the secret key and multicast address for group communication (many to one interaction).  
  |   | <name="headhunter1", password="secret1", domain="financial">  
  |   | <name="registry2", password="secret2", domain="financial">  
  |   | • The DSM authenticates the principals and returns a secret key and multicast address to a valid principal (one to many interaction).  
  |   | <secretkey = key.dat, multicast_address="224.2.2.2"> |

| 2 | This indicates the interactions between Service Exporter Components and active registries.  
  |   | • Service exporter components register with their respective registries (many to one interaction) -- <id="intrepid.cs.iupui.edu/AccountServer">  
  |   | • These registries in turn query these components for their UniFrame Specification (one to many interaction).  
  |   | <introspect property = “uniFrameSpecURL”>  
  |   | • The components respond with the URL at which the specification is located (any to one interaction).  
  |   | <url="C:\Account System\AccountServerSpec.xml"> |

| 3 | This indicates the interactions between Headhunters and Active Registries.  
  |   | • Headhunters periodically multicast their presence to a multicast group addresses (one to many interaction).  
  |   | <headhunterlocation = phoenix.cs.iupui.edu/headhunter1>  
  |   | • Active Registries, which are listening at this group address, respond to Headhunters’ messages by passing their information to Headhunters (many to many interaction).  
  |   | <registrylocation = magellan.cs.iupui.edu/registry2>  
  |   | • Headhunters intermittently query the active registries to which they hold a reference for the information of all the components registered with them (one to many interaction). The active registries respond by passing the list of
components registered with them and the detailed UniFrame specification of these components (many to many interaction).

4 This indicates the interactions between a Headhunter and a Meta-Repository.
   - Headhunters persist the component information obtained from the active registries onto the Meta-Repository (one to one interaction).
   - Headhunters query Meta-Repository to retrieve component information (one to one interaction).

<query="SELECT * FROM componentTable A, functionTable B WHERE (A.ID = B.ID) AND ((description LIKE%account%) OR (description LIKE %system%)) AND (end2endDelay<10) AND (availability > 90)"/>

   - Meta-Repository returns search results to headhunter (one to one interaction).

5 This indicates the interactions between the QM and clients.
   - Clients contact the QM and specify the functional and non-functional search criteria (many to one interaction).
   - The natural language-like client query is as follows:
     “Create an account management system that has end-to-end delay < 10 ms and availability > 90% preference maximum availability”.
   - Figure 4 shows the translated XML based query.

<Query>
  <Description> Account System </Description>
  <Domain> Financial </Domain>
  <End2EndDelay constraint="\"\" max Delay > 10 </End2EndDelay>
  <Availability constraint = "\"\" preference = "\"\" max Availability > 90 </Availability>
</Query>

Figure 4: Processed XML query

   - The QM returns the search results to the clients (one to many interaction).

< component 1: id="..", description="...", availability="...", ...
component 2: id="..", description="...", availability="...", ...
component 3: id="..", description="...", availability="...", ...>

6 This indicates the interaction between the QM and DSM.
   - QM contacts DSM for contact information of registered headhunters
belonging to the domain of client query (one to one interaction).

- DSM responds with list of registered headhunters (one to one interaction).
  
  <phoenix.cs.iupui.edu/headhunter1
  magellan.cs.iupui.edu/headhunter2>

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>This indicates the interactions between the QM and headhunters.</td>
</tr>
<tr>
<td>• The QM propagates Client’s query to all headhunters registered with it, which fall in the domain of the Client’s search request (one to many interaction).</td>
</tr>
<tr>
<td>• The headhunters respond to the QM query with search results matching the criteria (many to one interaction).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>This indicates the interactions between adapter components and AM.</td>
</tr>
<tr>
<td>• Adapter components register with the AM, which is running at a well-known location (many to one interaction).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>This shows the interactions between the clients and the AM.</td>
</tr>
<tr>
<td>• Clients contact the AM at the well-known location at which it is running with requests for specific adapter components (many to one interaction).</td>
</tr>
<tr>
<td>• The AM checks against its repository for matches and returns the results to the clients (one to many interaction).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>This shows the interactions between QM and LM.</td>
</tr>
<tr>
<td>• The QM propagates the query to the LM (one to one interaction).</td>
</tr>
<tr>
<td>• LM returns search results to QM (one to one interaction).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>This shows the interactions between the LM of one ICB and target LMs of other ICBs with which this LM is registered.</td>
</tr>
<tr>
<td>• The LM propagates the search query issued by the QM to the target LMs (one to many interaction).</td>
</tr>
<tr>
<td>• The source LM receives the result responses from these target LMs (many to one interaction).</td>
</tr>
</tbody>
</table>
5. **Comparison between URDS and Other Resource Discovery Protocols**

A brief comparison between URDS and other approaches is provided below.

- **Interoperability:** The other resource discovery protocols provide services for specific models and interoperations can be achieved only through proxies. URDS addresses the issue of non-uniformity by providing for discovery and coordination between components implemented using diverse models.

- **Network Usage:** Unlike other protocols, URDS clients and services do not participate in active discovery thus cutting down on the periodic communication required for the process of discovery. Instead, the active nature of the extended native registries allows the discovery process and removes the additional burden of developing ‘active’ components.

- **Query Processing and Matchmaking:** Unlike other approaches, which rely on Java-based or XML-based matching, the URDS supports a natural language-like query mechanism. This provides flexibility in formatting queries and during the matchmaking process.

- **Domain of Discovery:** In URDS the contextualization of the search space is logical and dictated by the industry specific markets. In other discovery protocols the notion of “administrative scope” is associated with the topology of the network domain.

- **Security:** The URDS security model addresses many of the common threats, which may occur during the discovery process. SSDS is another service notable for its robust security model.

- **QoS:** UniFrame incorporates of the notion of QoS as applied to software components and integrates this aspect into the service specification and the matchmaking process.

6. **Prototype and Experimentation**

A preliminary prototype [14] of the URDS has been implemented using the J2EE version 1.4 software environment. The core architectural components (domain security manager, query manager, link manager, headhunters and active registries) have been implemented as Java-RMI based services.

The repositories (domain security manager’s repository and meta repository) have been implemented using Oracle version 8.0. The Web-based components (JSPs), which service client interactions, are placed in a Tomcat 4.0 Servlet/JSP container.

The unicast communication between the core architectural components is achieved through JRMP (Java Remote Method Protocol) and the multicast communication between the headhunters and the active registries is achieved through Multicast sockets based on UDP/IP. The database connections are established using the JDBC (Java Database...
Connectivity) APIs and the user interaction is achieved through a browser front-end using the HTTP protocol. The security infrastructure, of URDS, is implemented by the security and cryptography APIs that form a part of Java Cryptography Architecture and Java Cryptographic Extension frameworks.

Preliminary experiments were carried out on this prototype to observe the performance of URDS. The experimental setup consisted of Sun SPARC machines connected by an Ethernet. The experiments contained one ICB, one headhunter, and one active registry (enhanced version of Java RMI registry). A single client was used to issue query requests, which consisted of different QoS constraints. The measurements were averaged over one hundred trials. The following times were measured:

- Average Authentication Time: It is the average time taken by the domain security manager to authenticate a principal (i.e., headhunter and active registry).
- Average Query Service Time: It is the average time taken to service a query.
- Average Registry Discovery Time: It is the average time taken by a headhunter to discover an active registry.
- Average Component Information Retrieval Time: It is the average time taken by the headhunter to retrieve component information from an active registry.

These initial experiments showed a value of 690 ms for the average authentication time. The average query time and the registry discovery time showed a marginal increase with an increasing number of registered components; while the average component retrieval information time increased linearly with the number of components (as expected).

The current prototype is able to discover only Java-RMI components, thus making it homogeneous. Efforts are underway to make it heterogeneous, i.e., able to discover components created using other models (such as CORBA, .NET, etc.) also. The current prototype also does not include the federation aspect.

7. Conclusion

The paper has presented an architecture that facilitates the semi-automatic construction of a distributed system by providing for the dynamic discovery of heterogeneous components and selection of components meeting the necessary requirements, including desired levels of QoS. The URDS architecture addresses issues such as interoperability, QoS of software components, scalability, fault tolerance, security and network usage. Interoperability is achieved by discovering components developed in several different component models. The discovery mechanism uses multicasting to detect native registries/lookup services of various component models that are extended to possess ‘active’ and ‘introspective’ capabilities. The component specification captures their computational, functional, co-operative, auxiliary attributes and QoS metrics. Flexibility in query formatting is achieved by providing support for natural language-like client requests. As a scalability mechanism URDS is organized in a federated hierarchical structure. Failure tolerance is handled through periodic announcements by entities and through information caching. Security is provided through authentication of the principals involved, access control to
multicast address resources, and encryption of data transmitted. URDS provides a directory based discovery service which is scalable secure and fault tolerant. Although, the current prototype does not address all the features of the URDS architecture, it has created a basis for validating the concepts behind URDS. Efforts are underway to extend the current prototype that will enable a validation of all the features presented in this paper.

References


37
Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2002.


An Architecture of a Quality of Service Resource Manager Middleware for Flexible Multimedia Embedded Systems

Marisol García-Valls
Department of Telematics Engineering
Universidad Carlos III de Madrid
Avda. Universidad 30, E-28911 Leganés (Madrid), Spain
email: mvalls@it.uc3m.es

Alejandro Alonso, José F. Ruiz, and Angel Groba
Department of Telematics Systems Engineering
Universidad Politécnica de Madrid
Ciudad Universitaria s/n, E-28040 Madrid, Spain
email: {aalonso,jfruiz,amgroba}@dit.upm.es

Abstract

There is a growing interest on multimedia applications and, in particular, in Consumer Electronics Embedded Multimedia Systems (CEEMS), such as set-top boxes and VTRs. At present, functionality changes and enhancement in CEEMSs are frequent. Manufacturers must adapt to such changes to keep in the market. Therefore, they must reduce time to market for their products. One of the clues to better improve and enhance system functionality is to develop easily upgradable (flexible) systems, so a modification in some function of a CEEMS will not imply redesigning the whole system. Current trend is to include programmable components in these devices to enhance their flexibility. In order to make this approach feasible, it is required a high and efficient use of resources. This paper presents HOLA-QoS, an architecture of a QoS resource manager that gives support to building flexible and open multimedia embedded systems. It is composed of a set of homogenous layers, where each of them manages one of the main system entities: applications, quality levels, and resource budgets. HOLA-QoS is flexible and composable in order to facilitate experimentation with different types of multimedia applications for CEEMSs and management policies. A prototype of this architecture has been built to test the main concepts in the design.

1. Introduction

Nowadays, the consumer electronics environment is very dynamic and changes rapidly. The appearance of new functionality for products and new standards for media processing and communications is fairly frequent. As a result, consumer electronic devices undergo constant upgrading. Such upgrading is derived from the need of manufacturers to keep their market share by offering new and fast solutions to market demands. Therefore, upgradability and time to market are very important parameters for fabricants. How can these parameters be kept within acceptable value ranges? A key to achieve this is to build flexible and open products —such products are systems for which upgradability is relatively easy. Instead of rebuilding the whole system, most of the existing technology may be reused.
To pursue this desired flexibility in consumer electronics, current trend is to replace dedicated hardware by software. Traditionally, hardware components dedicated to a single media processing function (MPEG decoding, etc.) have been used. This resulted in lack of flexibility; a change in a function would imply complete circuit redesign. At present, processing functions are starting to be coded in software and run concurrently in the same processor. Therefore, (programmable) resources will be used more intensively, and resource arbitration will require special attention compared to the dedicated solution.

1.1. Consumer Electronics Embedded Systems (CEEMS)

The focus of this work is on CEEMSs. These are embedded multimedia systems used as receivers in a broadcast environment. They process high quality digital video and audio [1], such as set-top boxes and VTRs. These devices are characterised by receiving an input signal from different possible channels (mainly by means of an antenna), performing digital processing of such data and generating an appropriate result in time.

Unlike the traditional multimedia systems that usually execute in powerful workstations, CEEMSs are mass-produced embedded multimedia systems with a limited amount of resources. The primary goal in CEEMSs is to efficiently and cost-effectively use the system resources. Though important, the network is not the main problem to consider in these systems.

Therefore, the main features of CEEMSs are the following.

- **Robustness.** They are commercial products. The user expects the device to show a correct function during the whole of its operation.
- **No artifacts.** Their output will have to be free from noise, distortion, and other perception disturbances.
- **Cost-effective resource usage.** Resources in CEEMSs are very scarce and also expensive, if compared to dedicated hardware solutions. To achieve a cost-effective resource utilisation, resources will be kept as busy as possible. This is also argued in [2].

1.2. Scalable media processing for family products

Consumer electronic devices usually come as family products. A family of products is composed of a set of devices, some of which offer better output results (this are high-end products) and others offer lower output results (low-end products). By software coding, it is possible to develop *scalable* media processing functions. By scalable, it is meant that a given function may be programmed so that it may give different output results. Which output result it gives, will depend on the (amount and type of) resources the function is allowed to use. This way, it is said that such functions may run in different modes, each of which will require different processing power. Also, each execution mode will be related to a different output quality and/or functionality.

Also, scalable media functions allow the provision of high-end functionality on low-end products, at the cost of a lower output quality. For instance, it is what happens in a TV set showing two picture windows simultaneously. One of it may be a main window playing a
movie from a digital input signal, whereas the other one may be a smaller window displaying an analog input signal. Them both may be shown at the same time on a high-end product. However, it may also be possible for a low-end product to have such functionality if the quality of the main window is reduced. This paper refers to these scalable media processing functions as multimedia applications or simply applications.

1.3. Architectural approach

In the literature, it is unusual to find a software engineering approach to QoS management in multimedia environments. Such an approach is presented in this paper. In this work, it has been developed a modular architecture (HOLA-QoS) based on components that aims at achieving flexible CEEMSs. Such architectural approach to QoS management is given in UML from the context diagram (which specifies a functional description of our system) to the task model (which gives a real-time basis of our approach).

The architecture HOLA-QoS (Homogeneous and Open Layered Architecture for Quality of Service management) gives support for building a flexible QoS resource manager for arbitration of concurrent execution of multimedia applications. This component architecture is the main goal of the work. In addition, robustness of CEEMSs is also a key point in this work. Robustness is addressed by relying on real time techniques and designing appropriate management protocols.

2. Background and motivation

There are a number of approaches to QoS management in multimedia systems, the vast majority of which are aimed at distributed multimedia systems and networking protocols. A complete survey in that direction is presented in [13]. In relation to the end system (which this paper addresses), focus of researchers is mainly at the following different levels: operating system scheduling algorithms for resource arbitration, middleware, and quality of service management architectures.

Firstly, low-level scheduling algorithms of system resources are of great importance. Operating system based approaches are mostly done in junction with the real time systems domain. In this area, it is found the work done in RT-Mach [9] for giving resource guarantees to tasks. Other operating systems such as Rialto [10] allowed resource assignments not only for individual tasks but also for task groups. On top of these extensions, other enhancements have been given to integrate QoS into them. For instance, interfaces are presented for the operating system to determine resource needs of applications. Also, for Rialto have appeared enhancements to raise the abstraction level it offers, such as Vassal [11] that gives the possibility for applications to load their own schedulers dynamically in the kernel. This might cause interference problems in case those incompatible schedulers are loaded at the same time. Operating system solutions fall at a very low level abstraction.

Other approaches to QoS management in multimedia systems are focused on middleware solutions. This is the case of DQM [5] where applications cooperate with the operating system for using the system resources. In DQM, the system does not enforce resource utilization. Another middleware solution is [6]. It consists of a set of user level schedulers
for integration of different types of applications. Research found in [7] for providing basic QoS guarantees to applications in end systems is also on this intermediate software side. Usually, middleware approaches do not rely on a RTOS but on a general purpose one. This way, it is not possible to enforce resource utilisation preventing predictable system behaviour on overload situations.

Lastly, some researchers also address the development of architectures for QoS management, which mainly result from the union of a set of algorithms for scheduling individual resources. As it has been said above, such architectures are aimed at distributed environments [3,4]. Also, high abstraction negotiation schemes are derived, such as in [8].

Most approaches based on the operating system focus on the end system. However, they provide too low abstraction management level. Middleware solutions and architectures are aimed at distributed multimedia systems, which are very much network oriented.

Our focus is not on distributed multimedia environments but on CEEMSSs, which are consumer terminals executing multimedia scalable applications, and having special features (i.e., need for flexibility to enable development of product families as one of the most important characteristics). This raises the need for further investigation in the field of QoS resource management, and applying architectural solutions that give support to such QoS resource management for CEEMSSs.

CEEMSSs are usually receivers in a broadcast environment providing high-quality digital audio and video, showing the following particular features with respect to the mainstream multimedia domain (which is the subject of work of the majority of research done up to present) [1]:

- High-quality video has an output frame rate of 50-120 Hz, and no tolerance for frame-rate fluctuations. On the contrary, in the mainstream multimedia domain frame rates are low (with a maximum of 30 Hz) and tolerance for frame rate fluctuations is high, also accepting frequent frame skips. This work relies on real time techniques for fulfilling time constraints.

- Programmable components (of CEEMSS) are more expensive in cost and power consumption than single function dedicated components. Therefore, it is very important for manufacturers to limit the amount of resources present in such devices. This constraint raises the need for cost effective utilization of system resources while still preserving the over-all system quality. One of the main issues of this work is the support for the appropriate protocols for application execution management. Such protocols range from low-level resource scheduling algorithms for individual application tasks to policies and strategies to govern application execution at a higher level.

- Although there are examples of scalable video algorithms for research done in the mainstream multimedia domain, industrial scalable multimedia applications for CEEMSSs are starting to be developed as prototypes. Their structure and behaviour is still not fully defined, though some scalable functions (i.e., scalable video algorithms) have already been implemented. Moreover, the range of applications
which may be run on these devices (3D graphics, video, audio, etc.) will result in the concurrent execution of applications of various natures. For this purpose, it is needed to develop QoS characterizations of applications with the least assumptions about application semantics. This work presents a QoS characterization of multimedia applications for CEEMSs that has been effectively used in our simulations and prototype implementations of HOLA-QoS.

These three points are the main motivation for the work presented in this paper that describes architecture for QoS management in CEEMSs. Architectural approaches for QoS have mostly been bottom-up, i.e., resulting from an elaborated union of several scheduling algorithms for individual resources. However, for achieving flexibility, this work identifies the need to create a top-down approach. For this purpose, the architecture HOLA-QoS has been created to meet the requirements stated above that motivate this work:

- It gives support for building flexible systems: it defines the placement of resource scheduling algorithms and higher-level policies. Those algorithms and policies are needed for developing flexible and open QoS resource managers in CEEMSs. Replacement of such algorithms and policies is as simple as replacing code modules, where interfaces are kept.

- It gives support for resource guarantees to multimedia applications of CEEMS. The architecture relies on real time techniques for task execution. Beneath it is a real time operating system that gives support for pre-emptive priority based execution and resource accounting of tasks and applications. These are the enabling facilities for performing resource enforcement.

Our solution also includes an additional feature: it gives support to QoS management at different abstraction levels. On one hand, it can be used to implement task execution management at lower levels. On the other hand, application execution (and quality level) management at higher abstraction levels is also supported. With abstraction levels, HOLA-QoS builds an integral QoS management for CEEMSs with hierarchical control across layers. Higher-level management is performed less frequently than lower level; however, higher-level management operations and decisions have more influence on system operation than lower level ones.

HOLA-QoS is a homogeneous layered architecture; it contains the same set of components for all layers. A QoS resource manager built following HOLA-QoS does not require the whole of the architecture to be used. It is possible to implement just low-level QoS management without using high-level strategies/policies and application level reasoning. This way, task level (or task group level) reasoning and control would be performed.

3. Functional description

The ultimate goal is to manage execution of a set of scalable multimedia applications in flexible CEEMS in order to maximize the global output quality. For this purpose, a Quality of Service Resource Manager (QoSRM) is built. It is a middleware program that dynamically manages system resources and assigns them to applications. Assignment is based on a contract model between the QoSRM and applications. The QoSRM agrees to
provide a given amount of resources to applications, which in turn should provide results with a certain quality level. The QoSRM guarantees applications the use of the assigned resources and prevents any other applications from using more resources than assigned. The interaction of the QoSRM with external actors is shown in figure 1.

Objects that use and receive shares of resources are called Resource Consuming Entities (RCE). They can be either tasks or clusters, which are a group of tasks that belong to an application. For simplicity, in the rest of this article, it is referred to RCEs simply as tasks. Resource shares are assigned to tasks in the form of budgets. A budget represents the number of time units that a task can use a resource. When the budget is consumed, it may be refilled, based on a refill period associated to each task. Agents of figure 1 are described below.

Applications. They are scalable, multimedia, and concurrent. They should provide stable output Quality Levels (QL) when the required resources are available. Applications are characterised by a set of QLs they can provide with a given input. As a general rule, the higher the QL, the larger the amount of resources that are needed. Each QL is characterised by the output quality it provides, the required resources, its time requirements, and its internal structure that defines the number and configuration of concurrent tasks. Moreover, applications should communicate to the QoSRM the available QLs and their characteristics. The QLs provided can change dynamically (due to the characteristics of the incoming media which is being processed). Applications should include procedures for changing from one QL to another, in a suitable way.

Real-Time Operating System (RTOS). A RTOS is needed for achieving time response predictability, dynamic resource management, and resource guarantees. The use of an appropriate scheduling policy and mechanisms built on top of the RTOS makes it possible to apply the analytical admission test. Such test is the basis for application admission.

QoSRM. The QoSRM is described by detailing the main functions that it performs.

Interaction with the user. The QoSRM provides an interface for letting the user decide what the system should do and know how the system behaves. Applications may provide specific interfaces and, in such a case, should communicate to the QoSRM the relevant user
requests. The major user requests are to launch and stop applications, change QLs and importance of applications.

System configuration. A system configuration defines a system execution scenario. It includes information about which applications are going to be executed, which are their QLs, which tasks are to be executed, and which are the required resources for each task, (and hence applications) to provide the selected QL. The QoSRM has to decide on the feasibility of a system configuration, that has might have been entered by the user or it may have been proposed during system optimisation. A configuration is admissible if there are enough resources to satisfy the needs of applications.

Negotiation with applications. A system configuration should be negotiated with applications. In the context of this work, this means that both parts agree about the QL to provide and about the required resources. In this approach, the negotiation is based on the knowledge that the QoSRM has about the QLs provided by applications and their characteristics.

QL configuration setting. Once a system configuration has been validated, it must be set, i.e. all applications in the system should execute according to it. This usually implies that a number of the applications should change their current execution QL. This process is called mode change. It has to be done in a way that keeps system output acceptable while the mode change is happening.

Resource usage accounting and enforcing. The QoSRM has to account for the resource usage time by tasks and applications. This way, it is possible to enforce the assigned budgets, to detect and handle applications overrun. Statistical information on applications resource usage and system resource availability is collected to let the QoSRM improve system behaviour. Overruns are notified to applications that will handle them.

System monitoring. The QoSRM is in charge of monitoring the behaviour of applications in order to ensure that the negotiated QLs are provided and to know the precise resource usage. It is possible to adapt system behaviour. If there are enough free resources, the QL of some applications could be raised, as a way of improving the global system quality. Alternatively, if there are applications that are not providing the appropriate QL, the system can reduce the QL of some of the applications, in order to reassign resources and ensure that all applications provide it. Reassignment of resources during adaptation is done based on the importance of applications.

4. QoS characterization

Application QoS characterization of HOLA-QoS (HOLA-QoS characterization) abstracts from application semantics as much as possible. For this reason, applications are modelled by identifying the parameters that are relevant to the QoS resource manager (system relevant parameters). HOLA-QoS characterization is presented as a structural view in UML, which identifies the main entities (parts) of a multimedia application. This characterization aims at being general enough to be applied to different types of multimedia applications. Such applicability is achieved through instantiation of the model with the
parameters that a given application needs. Instantiation is done easily by adding particular attributes to the classes of the UML model.

HOLA-QoS characterization has been derived from two main characteristics of the CEEMSs environment. The collaborative model is in the first place. In the proposed model, application experts collaborate with system experts in building a complete CEEMS. Application experts know well the structure of media processing functions (multimedia applications), and their resource needs in the average case. All such data is stored in the QoS characterization. Secondly, there are a number of different types of multimedia applications (video, audio, 3D graphics, etc.). HOLA-QoS characterization draws a general model to be flexible enough to support different types of applications. Figure 2 shows the structural model, which is given in UML notation, where classes are shown as a rectangle with the name inside it, and relations among classes are presented as solid lines.

In this model, an application (multimedia application) is composed of a set of tasks, which cooperate to perform a certain global function. Applications may have different operation modes called quality levels. Quality levels are directly related to the output presentation that the application may offer. Each quality level is related to a set of quality level configurations for it. This means that it can happen that an application may, for instance, have different combinations of tasks to reach the same quality level. However, at a given moment in time, one quality level has only one active quality level configuration for it. Application tasks are assigned a budget to execute on a system resource. A budget is an amount of resource that is granted for use. A budget is assigned for a single resource; therefore, a task may be assigned a budget for each resource it uses (CPU and memory for instance). As a general rule, only some budget assignments are meaningful for a task. This is due to the fact that application experts are able to obtain reference (not exact) resource usage values for the media processing algorithms that tasks execute. These values might depend on different issues, such as the type of the input data. Therefore, a new class, named TaskConfiguration, appears which models this type of relation. This model supports the existence of task groups or clusters. Using clusters has some benefits. For instance, it allows the assignment of a single resource budget or a single time deadline value to a group of tasks or cluster. This may have advantageous effects in terms of compensation of resource usage and response times of individual tasks of the cluster, respectively. The motivation of class ClusterConfiguration (which is made of a set of task configurations) is the same as for task configurations. Also, a given quality level configuration is made of one
or more cluster configurations. The model considers the existence of multiple resources, which applications may compete for. The budget concept/entity may be instantiated with attributes such as computation time, memory size or bandwidth to suit different choices of QoS parameters.

5. Overview of HOLA-QoS

HOLA-QoS contains four layers, as shown in figure 3. Each of these layers manages application execution at a different abstraction level, with the lower layers controlling individual task (and/or task group) execution and the upper ones managing quality levels and application execution.

The analysis of the QoSRM reveals that there are three main conceptual entities in the system that are subject of dynamic management and monitoring decisions: applications, quality levels, and budgets. When the QoSRM has to make a decision to improve the system behaviour, it has to modify the configuration of the system. This may be done by changing the settings with respect to any of the above-mentioned entities. This is the motivation for developing a layered architecture, where each layer performs management operations with respect to a different entity, and therefore, at different abstraction levels (i.e., at application level, at tasks level, etc.). In HOLA-QoS, the three upper layers deal with a different entity. The QoS Management layer (QSM) handles applications, the Quality Level Control layer (QLC) deals with quality level management, and the Budget Control layer (BDC) manages budgets. The Run Time Control layer (RTC) is in charge of accounting and enforcing system resources. It interacts with the RTOS and hardware for this purpose. The database contains system information, i.e. information of the set of applications that the system may run. This way, each application is modelled according to the QoS characterization that was explained in the previous section, which is the information that is relevant for the operation of the QoSRM.

Composability is a key feature of HOLA-QoS, which contributes to achieving flexibility for CEEMS. Therefore, the architecture layers are composed of a set of components for performing all management activities. The layers are homogeneous in the sense that they all contain the same set of components, as shown in figure 4. Components with the same name perform similar operations, but at the abstraction level of the layer in which they are located. In turn, each one of the layers performs the following operations:

- Decide on the setting of the entity that the layer handles.
- Set minimum requirements on the entity handled by the layer beneath it.
- If it cannot meet the requirements set on its entity, it notifies this event to the upper layer that will be responsible for solving the situation.
- If through monitoring, it is detected that there are available resources, the layer tries to improve the overall quality by changing the current setting of its entity.
To clarify the approach, let us consider the QLC layer, which handles the following entities: selected QLs for applications. Its upper layer communicates the minimum QLs that the QLC must satisfy. After a while, system monitoring may reveal that for some applications, it is not possible to satisfy the required QL. The reason may be that the application requires more resources than estimated. In addition, the detection of this event by the QLC layer means that the BDC layer has not been able to increase the budget assignment of the applications, because there are not enough free resources. In this case, the QLC has to free some resources by reducing the QL of some of the applications and assigning additional resources to the greedy application. If this is not possible because all applications are running at their minimum required QL at that time, the QLC has to notify it to the QSM layer, to take corrective actions. Such actions might be to stop some application or to notify the user of this. On the other hand, if the QLC detects that there are enough free resources, it could improve the QL of some of the applications.

5.1. Homogeneous structure of layers

The component structure of HOLA-QoS is modelled in UML and shown below in the component diagram of figure 4. Layers of HOLA-QoS follow a homogeneous pattern, except for the Run-Time Control layer, which is the lowest abstraction layer. For this, it has a special structure. The remaining layers contain the same components, each performing similar operations on the concept associated to that level. The components are the following ones:

Admission. Admission components of each layer are in charge of performing admission control to determine if a request can be satisfied by the system. Requests may come either from the user or from the QoSRM itself after monitoring. Also, they may be either for launching a new application or modifying the quality level of some application that is already running.

Settings. Components for setting the configuration of the system perform the necessary operations to change from the current situation (system configuration) to a new one. The system configuration is the set of tuples of the type \( (A_x, Q_y) \), where \( A_x \) is one application of the system and \( Q_y \) is the \( y^{th} \) quality level for application \( A_x \). The settings protocol provides a smooth and controlled transition to the new configuration. In HOLA-QoS, admission
protocol is separated from configuration setting, which introduces more flexibility for the system programmer.

**Monitoring.** These components contain the necessary functionality to perform monitoring of the system in terms of the amount of resources that applications (for upper layers) and tasks (lower layers) use. Adaptation capabilities in order to keep a cost-effective overall resource utilisation are also integrated inside monitoring components.

**Alarm Handler.** Abnormal task and application behaviour is treated by alarm handler components. Alarms have to be attended firstly by the layer where they have been generated. Such layer tries to solve the situation which caused the alarm; however, if it cannot solve such situation, it passes the alarm to its immediate upper layer that will try to fix the situation, and so on. If the alarm reaches the upmost layer (QSM Layer) and still cannot be handled, it means that the alarm is an unrecoverable error, and the system will exit.

**Interface with External Actor.** Each HOLA-QoS layer contains an interface to interact with actors that are external to the architecture itself (applications, user, and RTOS). This way, the interface of the QSM Layer enables the QoSRM to communicate with the user for capturing information on user requests and global strategies to be applied. Also, the QLC Layer has an interface to interact with applications for carrying out the configuration establishment protocol and capturing application notifications (temporal disabling of some quality levels that may be due to changes in the incoming media, etc.). In the BDC Layer, this component contains the necessary functionality for applications to register creation/deletion/modification of their individual tasks/ task groups.

As it has been introduced above, the homogeneous structure of HOLA-QoS applies to all layers except for the Run-Time Control layer. This is the lowest level one; its main purpose is to enhance the functionality given by the underlying RTOS. For this purpose, the RTOS enhancement component gives a higher abstraction level to the RTOS primitives adjusting better to what multimedia applications need. For this purpose, it contains: wrappers to the RTOS primitives, resource usage accounting functionality, resource budget assignment for tasks and task groups, and resource budget enforcement functionality, to give guarantees on resource budget assignments.

### 6. Hierarchical control

HOLA-QoS makes a distinction among strategies and mechanisms. This, together with its layered structure and various QoS management abstractions, enables a hierarchical control structure. In this approach, strategies are high-level decision rules or algorithms that govern the system operation. On the other hand, mechanisms correspond to the lower level operation algorithms. Mechanisms are related to automatic activities that are performed in a more frequent manner. Whereas strategic decisions are made less frequently, and their purpose is to change system operation according to the occurrence of special events (mainly, user requests and input data changes).

The control structure of HOLA-QoS is hierarchical. Upper layers of the architecture contain the strategies or policies that govern (parameterise) the mechanisms that in HOLA-
QoS belong to the lower layers. For instance, let us imagine the following global strategy of the system: "give absolute priority to all applications having user focus". In the lower layers, this will be mapped to applying a suitable priority assignment mechanism to tasks. Therefore, tasks that belong to applications having the user focus, will be assigned higher priorities in the system. However, if a global strategy is to "keep the overall output quality", then a more fair priority assignment scheme will be used by scheduling mechanisms in the lower levels of the architecture.

Upper layers of the architecture perform their management tasks less frequently while lower layers perform their operations more frequently. This way, the frequency at which the system has to modify resource budgets assigned to tasks is higher than the frequency of QL changes or requests for applications to be launched in the system.

The architecture gives the possibility to perform QoS management at different abstraction levels. Therefore, it is not necessary to use all levels of the architecture if only low-level management for tasks and working with resource budgets is needed. Below, the main management protocols are described with their complete operation-sequence across all architecture layers.

**Admission control.** The admission control protocol determines if the system is capable to switch to a certain system configuration (due to a request to launch an application or to switch some application quality level) given the current available resources. Each layer performs admission at its abstraction level, and passes the results of their admission operation to the neighbouring layers.

To explain this idea, the QSM layer passes a configuration of the form \((A,Q)\) to the QLC layer. The quality level \(Q\) of application \(A\) may be obtained from a specific user request or, if the user does not specify an initial QL, QSM layer picks a default one. After, it passes this information (tuples of type \((A,Q)\)) to QLC layer where the admission component maps this QL to a set of resource budgets obtained from the system info table (where all such mappings are fixed from the beginning). Then, QLC layer generates a set of tuples (task, resource budget) for all tasks in the system and it passes this information to the BDC layer.

The admission component of the BDC layer contains the schedulability algorithm that applies to the received system configuration determining whether it is schedulable or not.

The admission control protocol is based on a schedulability algorithm contained in the BDC layer. This algorithm is based on the resource budgets assigned to tasks, and other relevant task parameters: priority, activation/budget-refill period, and deadline. The BDC layer manages resource budgets. Therefore, the analytical algorithm is contained in this layer. Also, the admission protocol determines which is the best low-level configuration for it, according to the current strategy (i.e., optimise the usage of some resource, etc.).

**Configuration setting.** Configuration switches are due to changes in application quality levels, user requests to launch/stop applications, etc. This section describes the configuration setting protocol of HOLA-QoS, which determines the way in which the current system configuration will be replaced by a new one.

In order for HOLA-QoS to be as general as possible to support different kinds of scalable multimedia applications, the configuration setting protocol has been developed to be
independent from application semantics. High-level configuration settings protocol is included in the settings component of the QLC Layer. It uses the interface provided by the settings component of the BDC Layer, which gives very much flexibility to building high level configuration setting protocols in HOLA-QoS. HOLA-QoS gives support for immediate and progressive mode changes or configuration settings protocols. Immediate mode changes are typical of media processing functions which do no tolerate data loss (for instance, when processing of one data unit is done on an individual basis, i.e., it does not depend on former or subsequent data units). Progressive mode changes are required by applications with very small tolerance to data loss. For the immediate configuration setting protocol, the QoSRM may send the reconfiguration requests to applications in two ways. First, requests are sent according to the load introduced in the new mode. Applications that lower their resource demands in the new mode change first. Second is application importance, i.e., applications with higher importance are requested to change first. Which mode change type is used will depend on the system strategy that is being applied, according to the applications which are active in the system.

**Monitoring.** HOLA-QoS allows monitoring system behaviour at all levels through the monitoring components of each layer. Differences in the monitoring operations performed by the various layers are mainly the following: the frequency at which each layer performs monitoring operations and the information that each layer receives and manages.

The essential monitoring information (basically, resource usage of tasks) is obtained by the Run-Time Control layer. This information is passed to the upper layers. Each layer receives monitoring information from the layer beneath it, which comes in the format structure defined by the RTC later. Monitoring information is then processed and encapsulated to create higher abstraction information (of the appropriate level for the receiving layer).

If during monitoring, a high overrun percentage of tasks or high percentage of deadline misses is detected, the QoSrm informs applications of this. Applications may try then to make a request to change their QL, if needed. The monitoring protocol in HOLA-QoS is also adaptive. This way, in case of over- or under utilization of resources, the QoSrm performs adaptation operations to balance the system load. Adaptation decisions are based on the current global strategy. Strategies for adaptation may be: maximize resource usage in the system, prioritise applications which have the user focus, and applications according to their importance.

The high level monitoring protocol addresses the cost-effective use of resources, which is one of the main requirements of multimedia embedded systems. For this purpose, it introduces an absorption band, to absorb load peaks. This band determines a value range for the total system load. If the total system load falls inside this band, then it is considered that the resources are being used cost-effectively. Otherwise, the system initiates the adaptation process.

**7. Task model**

HOLA-QoS is flexible enough to be associated to various task models, depending on the characteristics that are desired for the system which will be built. This section presents a
particular task model for the architecture which has been chosen for three main reasons. The first one relates to the simple design for keeping system overhead low. This is achieved by reaching the suitable number of tasks for the QoSRM, so that it does not introduce unacceptable overhead and tasks interference. Secondly, this work pursues suitability for centralized systems but to ease distribution. This work aims at obtaining a task model for a centralized environment but with an open structure for easily adding new functionality, and obtaining a distributed version of the architecture. Lastly, adjustment to the various abstraction levels of the architecture and its hierarchical control. There are different tasks that perform similar operations, but on distinct entities of the system and at different frequency. This depends on the abstraction level of the task.

![Diagram of Task Model](image)

**Figure 5: Task Model**

<table>
<thead>
<tr>
<th>Task name</th>
<th>Main Component</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Interface (QSM Layer)</td>
<td>Driving admission control protocol. Selecting best system configuration.</td>
</tr>
<tr>
<td>Application Interface of QLC Layer</td>
<td>Interface (QLC Layer)</td>
<td>Manages communication with applications for exchanging requests (during mode change protocol or spontaneously from applications).</td>
</tr>
<tr>
<td>Application Interface of BDC Layer</td>
<td>Interface (BDC Layer)</td>
<td>Managing registry of operations that applications perform with tasks.</td>
</tr>
<tr>
<td>Mode Change</td>
<td>Settings (QLC Layer)</td>
<td>Controlling switching from current system configuration to new one.</td>
</tr>
<tr>
<td>Monitorization of QSM Layer</td>
<td>Monitorization (QSM Layer)</td>
<td>Performing highest level monitorization operations (on application entities). Making decisions on the overall application set.</td>
</tr>
<tr>
<td>Monitorization of QLC Layer</td>
<td>Monitorization (QLC Layer)</td>
<td>Performing monitorization operations and deciding on QLs of applications and their implementation.</td>
</tr>
<tr>
<td>Monitorization of BDC Layer</td>
<td>Monitorization (BDC Layer)</td>
<td>Monitoring actual resource budget usage and deciding on adjustments of budgets to be assigned.</td>
</tr>
<tr>
<td>Alarm handlers</td>
<td>Alarm Handlers of all layers</td>
<td>Managing alarm/error occurrences in the system.</td>
</tr>
</tbody>
</table>

**Figure 6: Description of the Task Model**

Figure 5 shows a UML task model for HOLA-QoS. Invocations among tasks are represented with connecting lines. For each task, table of figure 6 shows a column for the component from which the task executes most of its functionality, and another column for describing the main activities that it performs. Alarm Handler components have not been implemented; this is a subject of future work. For this reason, Alarm Handler tasks are presented as a whole. In this task model, it is proposed that upon occurrence of an alarm, the task of the layer in which it happens, first tries to handle it. As explained in section 3.1,
if this cannot be achieved, it should inform the Alarm Handler task of its upper layer, so that it tries to solve the situation.

8. Measurements and results

The presented approach differs significantly from the ones which there are currently in the field of QoS management for multimedia systems. This is mainly due to the following reasons:

- Architectures are aimed at a distributed multimedia environment for the mainstream domain where the network usually plays one of the most significant roles. Our aim focuses on the multimedia embedded systems, that present special features compared to the mainstream multimedia domain.

- Architectures usually are the result of the union of algorithms for managing specific resources which have been previously developed. Here, obtaining an architecture which gives support for building flexible and open multimedia embedded systems is not the primary focus.

- HOLA-QoS does contain its own management protocols (for admission, configuration setting, and monitoring) that aim at being simple and general enough for embedded multimedia systems. However, it is not the main focus of this work to develop such management protocols, but to give architectural support for flexible CEEMSs. In HOLA-QoS, updating a management protocol is as simple as replacing the appropriate component keeping its interface.

Therefore, there is no existing prototype implementation which is clearly related to the approach of HOLA-QoS. In this sense, no comparison with other work would give meaningful results.

This section presents the results which have been obtained from the implementation of a QoS resource manager with the HOLA-QoS architecture. The motivation for this section is to show the validity of HOLA-QoS by means of presenting two types of experiments: functional (which show the feasibility of the HOLA-QoS architecture and the soundness of the concepts behind it) and performance experiments (to show the efficiency of this architecture through the measurements obtained in the operation of synthetic and real multimedia load in the system).

The implementation of HOLA-QoS contains the four layers and protocols for admission, configuration settings, and monitoring. All layers provide their complete interface except for alarm handler components.

At present, HOLA-QoS considers only CPU resource. Therefore, quality levels are mapped, in the end, to CPU budgets for particular application tasks (i.e. quality levels are mapped to certain sets of task configurations). Figure 7 shows the QoS characterization of synthetic applications which have been used in our experiments.
The target system for all experiments is a real one: a multimedia microprocessor TM1000 [14] of Philips Semiconductors with a software development environment [15] for TriMedia. The underlying operating system is a real-time one, pSOSystem [16].

Figure 8 shows the benefits of using our prototype of a QoSRM implemented with the HOLA-QoS architecture. Figure 8 shows two executions of the synthetic application A of figure 7. In the first situation, application A runs on top of the RTC layer of HOLA-QoS, therefore, with no quality level management. The second experiment runs application A with a QoS resource manager built according to HOLA-QoS specifications.

For both experiments, noise was progressively introduced by running independent tasks that are intensive processor consumers. In absence of QoS resource manager, application A cannot keep its output packet rate, as the CPU becomes busier. However, it can be seen that this does not happen in case that the QoS resource manager implemented with HOLA-QoS is used in the system. In this case, application A keeps its output packet rate around a stable value. The overload introduced by the QoS resource manager is slightly noticed only for low CPU load.

Following, figure 9 shows one of the functional experiments which have been carried out for the execution of three synthetic applications described in figure 7. The purpose of experiments of this type is to observe that execution is normal, the system behaves as expected, and user requests are fulfilled within the acceptable time bounds. Changes in the load that applications introduce correspond, as expected, to user requests to launch, stop, or change quality levels of applications and system adaptation decisions. The absorption band for this experiment is [65,85] (percentage of CPU load). So the adaptation protocol keeps, when possible, system load within this value rage.
For the same experiment, figure 10 shows the variation in the output time of packets generated by application A. The three quality levels of this application are related to the values presented in the X coordinates. This way, when application A executes at its lowest quality level, it requires 16% of CPU, whereas for the highest quality level, it needs 36% of CPU. Figure 10 shows that for all quality levels that application A switches to, the system is able to maintain the contract made with the application. This way, the packet output time is kept almost constant for its different quality levels.

Figure 11 describes one of the experiments done with real load. On one side, there is a synthetic application with three quality levels. On the other side, there is a video application that processes raw video with two quality levels that correspond to: big screen with high resolution and small screen with low resolution.

Figure 12 shows the processing time for frames of the real application. It can be seen that, as expected, frame processing time is kept stable around a value of 4 ms for the low quality level and 9 ms for the high quality level. Interference from the synthetic load introduced by application A and its mode changes does not cause disturbance to the processing of the real load.

From all the experiments made with the prototype implementation of a QoSRM with HOLA-QoS, it has been concluded that results and measurements have met the expectations.
9. Conclusions

The design of future CEEMSs is a challenging goal, due to their heavy requirements and high user expectations. In order to make them profitable, they should be robust and provide high quality outputs, using limited computational resources. The dynamic management of these resources is basic for meeting these goals. Moreover, CEEMSs must be flexible and easily upgradeable for enterprise benefit. For this purpose, suitable architectural support is also necessary.

A software engineering approach for QoS resource management is unusual, and so is to address multimedia embedded systems. The main result of this approach has been the development of an architecture for a QoSRM in an up-down process. In this paper, the architecture HOLA-QoS has been introduced as a software engineering approach to QoS management for multimedia systems. Its goal is that system resources be used in such a way that the output quality of applications is maximised. At the same time, budget enforcement and support to detection of missed time requirements provides a solid basis for developing robust applications. The QoSRM uses real-time techniques for meeting these goals.

HOLA-QoS is intended to be flexible and composable, in order to ease experimentation with different types of algorithms, protocols, and applications. It is composed of a number of layers, which handle the main system entities and are designed in a homogeneous way. The main functions of these layers are to select and set feasible system configurations, to monitor and adapt system behaviour, to handle faulty situations and to interface with external agents.

A prototype of the proposed QoSRM has been developed. A number of initial tests have been executed with synthetic and real multimedia load to check its suitability. Results were positive and the main ideas in the design of the QoSRM seem to be useful for the development of CEEMSs.

Acknowledgements

The research described in this paper has been done in cooperation with the Information Processing Architectures Group at Philips Research Eindhoven, which has hosted some of the authors to develop this work. Specially, the authors would like to thank Liesbeth Steffens, Sjir Van Loo, and Reinder Bril for their contributions to this work, and their comments and review.

References


Session II – Specification
An Event/Rule Framework for Specifying the Behavior of Distributed Systems

Javier A. Arroyo-Figueroa, José A. Borges, Néstor Rodríguez,
Amarilis Cuaresma-Zevallos, Edwin Moulier-Santiago,
Miguel Rivas-Avilés, Jaime Yeckle-Sánchez

Center for Computing Research and Development
Department of Electrical and Computer Engineering
University of Puerto Rico - Mayagüez Campus
Mayagüez, Puerto Rico 00680
Contact: jarroyo@ece.uprm.edu

Abstract

Although a number of standards for developing distributed systems (DS) already exist (e.g., RPC, CORBA, DCE, DCOM, Java RMI, Jini), they still lack of abstractions, services and tools for specifying the behavior of a DS. The specification in these environments is limited to the interface, i.e., which are the attributes and methods of distributed objects. We believe that behavioral abstractions should be part of the specification of a DS. This paper presents ERF, an Event/Rule Framework that provides a set of abstractions for specifying the behavior of DS in terms of events and rules. Rules in ERF are used to specify how distributed objects react to event occurrences. ERF has an open architecture which can be integrated to support different environments like CORBA, RMI and Jini.

1. Introduction

Although a number of standards for supporting distributed systems (DS) already exist (e.g., RPC, CORBA, DCE, DCOM, Java RMI, Jini), they still lack of abstractions, services and tools for specifying, designing, implementing, monitoring, debugging and maintaining a DS. One possible abstraction to support of these activities is a conceptual view of the system. We mean by conceptual view the specification of the system as seen by the community of developers, in terms of structure and behavior. It should be noted that existing "middleware" do well in specifying structure, i.e., attributes and structural relationships among system components. However, in terms of behavior, the specification is limited to the definition of function- or method-signatures. The semantics of behavior are hidden or "buried" inside the application code (implementation). Therefore, anyone interested in knowing the behavioral semantics of the system either has to look into the application code, or into a specification or design document (which probably will be "out of sync" from the implementation). Furthermore, existing environments do not allow incorporating changes in behavior dynamically; any change in the behavior will involve changes in the implementation of functions or methods, which requires recompilation.

It is our belief that behavioral abstractions should be part of the specification of a distributed system. Such specifications may be specified at a high level, maintained and executed in an environment which provides (i) high-level abstractions (e.g., rules) for
specifying behavior in terms of events, conditions and actions; (ii) ability to incorporate changes dynamically; (iii) an engine for specifying and processing rules that react to events in real time; and (iv) tools for specifying, designing, implementing, debugging, monitoring and maintaining the system.

This paper presents ERF, an Event/Rule Framework that provides a set of abstractions for specifying the behavior of DS in terms of events and rules. ERF was designed with the following objectives:

**Object-oriented model.** Similar to CORBA, DCOM and Java RMI, ERF has an object-oriented model in which system components are treated as objects. ERF does an extension to existing models by treating events and rules as objects.

**Support of multiple standards.** ERF is designed to support the multiple standards for distributed system environments (CORBA, Java RMI, DCE and DCOM). An open architecture allows extensions for supporting new standards as well.

**Events.** ERF uses events as an abstraction for specifying system behavior. The specification of behavior is done in terms of events that trigger rules. A formal object-oriented event model allows the systematic definition of events.

**Rules.** In ERF, ECAA (Event-Condition-Action-AlternativeAction) rules are used to specify system behavior. Rules are defined in terms of trigger events, conditions that need to be satisfied to apply the rule, and a set of actions and alternative actions to perform when the events occur and the conditions are satisfied. A formal object-oriented rule model allows the design and implementation of a rule support system independent of any particular rule language syntax.

**Intelligent event service.** The heart of ERF is a Rule-Based Intelligent Event Service (RUBIES). Following the object-oriented paradigm of existing standards, RUBIES comprises a set of classes whose interfaces include methods for rule definition, event notification (or posting), registration of event producers and consumers, and rule management.

The rest of this paper is organized as follows. In the next section, we present a survey or related works. In Section 3, the architecture of ERF is presented. The functionality, model and languages of a Rule-Based Intelligent Event Service (RUBIES) is presented in Section 4. In Section 5, we present the features of the Integrated Development Environment (IDE). The development of a Real-Time Flood Alert System (RTFAS) using ERF is discussed in Section 6. Current implementation status is presented in Section 7. Conclusions are presented in Section 8.

2. **Related work**

Many commercial DS support environments are already available, supporting standards such as CORBA [1], DCE [2], Java RMI [3] and DCOM [4]. All of them have in common a “programmatic” view, which lacks abstractions, tools and services for specifying, monitoring, debugging and maintaining DS under an integrated development environment.
Bates’ work on EBBA (Event-Based Behavioral Abstraction) lead to the development of a toolset for specifying and debugging distributed systems [5]. An event description language (EDL) is used to define simple and composite events, and behavioral models of different scenarios of a distributed system. It does not have an object-oriented model. However, the concepts of this work have inspired part of the current research.

CORBA (Common Object Request Broker Architecture) [6] has incorporated events and event services in its object model [7]. The services CosEvent and CosNotification have a set of services that support event management. Event handling is performed at the object level, i.e., only localized event handling is supported. Composite events and delayed event handling are not supported.

In the work by Bacon et. al [8], CORBA’s Interface Definition Language (IDL) is extended to incorporate events. Like in ERF, events are uniformly treated as objects using an event class hierarchy. Simple registration and notification services are provided, relying on client-supplied event handlers. Only a limited set of composite event types is supported by means of a language based on finite state machines.

DCE (Distributed Computing Environment) [2] has not yet incorporated events or event services in its object model. However, Cohen and Wilson [9] proposed an event management service based on the CORBA event specification. Only event-attribute-based event filtering is proposed. Composite events are not supported.

Mansouri and Sloman present a configurable event service for the Darwin distributed environment [10]. Their approach is similar to ours in the use of rules as an abstraction for the specification of event handling. A language called GEM is used to define events and rules for event handling. Many of different types of composite events are supported by a set of predefined operators. Temporal constraints are also supported. Contrary to ERF’s approach, events, rules and the event service are not uniformly treated as objects.

EVEREST [11] is a system tailored for the study of various approaches to event recognition and notification. It allows sophisticated users to define, by means of a script file, primitive and higher-level (i.e., composite) events, and assign event handling to predefined processes. A set of operators is defined to capture the semantics of composite events. Contrary ERF’s approach, event-handling responsibility is relied to predefined processes called monitors. The computational model of this system is not object-oriented.

ISIS [12] is a CORBA-based toolkit for supporting distributed messaging in form of events. It provides a C library for support services such as channels, multicast, membership management, failure detection, fault-tolerance, and group monitoring. The main purpose of ISIS is to provide an environment for group messaging. Event processing is done at the client side (client-supplied event handlers).

Most of existing event-rule approaches have been applied to active databases. Examples of such approaches are Ariel [13], Sentinel [14], REACH [15], OSAM*.KBMS [16]. They all have in common the use of a rule language for specifying database operations to perform upon the occurrence of events, which are other database operations. Sentinel and REACH, although applicable to active databases, are more general extensible object-oriented systems. Although they were not designed specifically for supporting DS, they are similar.
to the ERF in the treatment of events and rules as objects in a rich object-oriented model. However, rules are compiled and tightly coupled with the database system.

Su et al. proposed the NCL language [17] and an Knowledge Base Management System (OSAM*.KBMS) [16], which use rules as abstractions for specifying interoperability in DS. NCL is a combination of the EXPRESS and IDL languages proposed to be used in the NIIIP project, using CORBA as the underlying support environment. OSAM*.KBMS does the rule processing that is not supported in CORBA. NCL and OSAM*.KBMS do not treat events or rules as objects, and only simple events are supported.

3. Architecture

ERF has an open architecture, which allows extensions to support multiple standards. Currently, ERF is implemented in Java RMI and CORBA. The architecture of ERF is presented in the UML diagram of Figure 1. The architecture consists of two main structures: (i) a static structure, which holds the objects that represent definitions (e.g., event types and rules); and (ii) a dynamic structure, which holds the objects that encapsulate the behavior of a distributed at run time (e.g. all distributed objects).

The static structure of ERF defines packages, types and rules. Similar to Java, ERF provides the package abstraction. Packages in ERF are collections of distributed object classes, rules, and other packages. The classes Package, DistributedObjectClass, EventType and Rule represent packages, distributed object classes, event types and rules, respectively. The class ERFException type represents the event types that represent system exceptions.

The dynamic structure defines the distributed objects that encapsulate run-time behavior. The class DistributedObject is the base class of all distributed objects. The following distributed objects are part of ERF: (i) RUBIES, which is a Rule-Based Intelligent Event Service that executes rules based on the events that occur in the system; and (ii) EventChannelInterface, which is represented by an abstract class with different implementations for different environments (e.g., RMI, CORBA, etc.). Depending on the middleware environment, there are different implementations of the EventChannelInterface. Since RMI does not provide an event channel, an ERFEventInterface is provided to interface it with ERF’s built-in event service. For CORBA, a CORBAEventInterface interfaces ERF with CORBA’s COSNotification service.
RUBIES manipulates two sets: a rule set, consisting of all the rules that are active and which can be triggered by events; and an event set, consisting of all events that are alive and which can trigger rules. The class Event is an abstraction of an event; all event types are defined by deriving a class from this class. A subclass of Event, ERFException defines a special type of event which occurs upon a system exception.

4. A Rule-based Intelligent Event Service (RUBIES)

A Rule-Based Intelligent Event Service (RUBIES) is the engine with the functionalities required to support the event-rule framework. The design of RUBIES is driven by the following rationale:

Object-Oriented. We follow the object-oriented paradigm supported in environments such as CORBA, DCE, DCOM and Java RMI. Events, rules and the RUBIES itself are treated as objects.

Robust event/rule model. The event and rule models of RUBIES support simple and composite user-defined events, relative timing and relative presence/absence.

Rule-based event handling. Instead of being buried inside application code, event handling is performed by means of rules. ECAA rules (Events-Condition-Action-
AlternativeAction) are defined in terms of event filtering specification (triggering events, priority, conditions), actions and alternative actions.

**Immediate and delayed event handling.** Both immediate and delayed event handling is supported by keeping events in an event set during a specific period of time ("time-to-live") defined for each event. Delayed processing is carried out by inactive rules, which, upon activation, are processed against events existing in the set.

**Distributable and replicatable architecture.** The architecture of RUBIES allows for the distribution of load among different instances of the service (for scalability and performance). Similarly, many replicated instances of RUBIES may co-exist in a DS (for fault tolerance).

### 4.1. Event model

In RUBIES, events are uniformly treated as objects. An event type defines the structure and behavior that is common to a set of like events, and is represented by an event class. The class Event is the base class of all event types. It defines the structure and behavior that is common to all event types. Figure 2 presents a Java language definition of the class Event.

The Event class defines common attributes such as “eid” (unique event identifier) "daytime" (value of the day and time when it was produced), “ttl” (time-to-live), and “producer” (the distributed object that produced the event). Also, a set of common methods are defined in this class: (i) “t()”, which returns the day and time of the event production; (ii) “ts()”, which returns the amount of time of life of the event; (iii) “producerClassName()”, which returns the name of the class of the object that produced the event; (iv) “getProducer()”, which returns a reference to the distributed object that produced the event; and (v) “isDead()”, which returns “true” if the event has past its own time-to-live.
In Figure 3, an example of an event type specification is presented. The example is taken from the RTFAS system (see Section 6). In this example, the event type GageLevelReport is defined by means of an event class of the same name. Notice that in addition of the inherited attributed, this event type defines new attributes such as “level” and “loc”.

```java
// Class GageLevelReport definition
public class GageLevelReport extends Event {
    double level;  // gage water level
    int loc;  // gage location
}
```

Figure 3. Example of an event type specification

4.2. **Rule model and specification language**

Rules are abstractions that allow to declaratively specify the behavior of a DS. The rule model defines the structure of each rule having the following components:

**Trigger events.** The trigger event specification is a set of events that will trigger the execution of the rule. A simple event triggers the rule upon its occurrence. Composite
events trigger the rule depending on the relationships specified among events (e.g., one event along with others, one event but not others, time-related).

**Usage.** Sometimes rules need to make use of services of an existing DS component. The usage clause of a rule specifies which services are to be used in evaluating a condition or performing an action.

**Priority.** The execution paradigm of rules needs to include rule priority, such that semantics are captured correctly by executing rules in the appropriate order.

```
package aaa;

rule eip1
  on GageLevelReport glr1 >> GageLevelReport glr2
  if (glr2.t() - glr1.t() <= 15:00) && (glr2.level - glr1.level >= 0.75)
    && (glr2.loc == glr1.loc)
  then
    post EventInProcess { loc = glr1.loc };
end;

rule eip2
  on GageLevelReport glr1 && {GageLevelReport [ (ts() <= 15:00)
    && (level >= 12.0) && (level <= 15.0) && loc=glr1.loc] } S1 
    && {GageLevelReport [ ts() <= 15:00] && loc = glr.loc } S2
  if (S1.size() / S2.size() > 0.5)
  then
    post EventInProcess { loc = glr1.loc }
end;

package aee;
import aaa;
rule aeenotify
```

**Condition.** A condition may be specified for a rule, which must be satisfied ("true") to execute the rule upon the occurrence of events. Rule conditions may include time relationships among events (e.g., one event before another, one event 5 minutes after the other).
Action. The action specification of a rule states a list of operations to be performed if the trigger events occur and the condition is satisfied. This is the event handling part of the rule.

Alternative action. A list of operations can be specified to be performed when the rule is triggered but the rule condition is not satisfied.

RUBIES provides a syntax-independent interface for defining rules. This allows any client (e.g., a rule compiler) to define rules using any syntax. Such client is responsible for providing the rule definition in the common structures provided by ERF. A client that provides definitions in a format understandable by ERF is called ERF-compliant. We have implemented a rule compiler that uses a language called RDL (Rule Definition Language).

Figure 4 presents examples of rule definitions in RDL. These rules are part of a Real Time Flood Alert System (RTFAS), which is currently under development using ERF (see Section 6).

4.3. Interface and behavior of RUBIES

The interface of RUBIES provides the following functionalities:

Registration: The interface provides means of registering services that act as facilitators.

Rule management. Some methods allow authorized clients to manage rules. New rules can be added, existing ones can be modified, and a rule evaluation schedule can be defined dynamically, and such changes should take effect immediately. For each rule object, a set of attribute values is set, such as: trigger events, priority, conditions, actions and alternative actions.

Notification. Notification methods notify the occurrence of events. Upon notification, applicable rules are triggered and executed.

5. Integrated Development Environment

The integrated development environment (IDE) of ERF includes tools for the efficient specification, coding, debugging, monitoring and maintenance of DS applications. In other words, the IDE provides a framework that supports all the activities in the development process of DS. The IDE has the following design rationale:

Easy to learn and use. Human computer interaction and usability engineering issues and methods have been given high priority to make sure that this goal is accomplished. A unified model to define rules and events has been developed such that a consistent representation could used on most aspects of the DS life cycle.

Language independence. User should not be required to know a specific programming language (e.g., C++, Java) in order to specify the structure and behavior of a DS at a high level.

Independence of the underlying DS environment. The development of a DS using this environment should be independent of the underlying environments (CORBA, DCE,
DCOM, Java RMI, etc.). In fact, users should be able to specify or select any underlying environment.

The IDE requires tools to support the following processes:

**Specification.** The IDE also includes tools to specify application, event and rule objects. This is useful for supporting the requirement specification, design and coding processes. The framework should provide a tool to define wrapper classes, associate them with specific services, and specify their attributes and methods. This process generates code according to the a particular DS environment (e.g., CORBA). Similar to application objects, the event objects can be defined based on a standard protocol. An event category is selected from the Event hierarchy, or a new category is defined, then their attributes and methods are specified. The definition of composite events is based on the rule model. The tools should include facilities for specifying rules.

**Monitoring and debugging.** These two processes are highly coupled. Monitoring and debugging tools are needed to keep track of event occurrences, event flows, message flows, rule triggers, event forwarding and changes in the state of the system. These tools are needed for both simulation runs and during the execution of applications.

Figure 5. Run-time and build-time scenarios of the IDE

Figure 5 presents the run-time and build-time scenarios of usage of the IDE. It is convenient to present both scenarios together, since the IDE allows to define events and rules dynamically. Build-time flows are represented by dashed arrows; run-time flows are represented by solid arrows. During build time, developers may define wrappers, events
and rules, which go into the IDE repository. Notice that the IDE repository might be a
distributed system component accessed by the DS communication infrastructure (e.g.,
CORBA’s ORB). The middleware compiler converts the language-independent wrapper
specifications into language-specific wrappers, which are used to wrap distributed system
components during build time. During run time, the specification tool can be used to
specify new or modify existing events and rules. RUBIES instances access the IDE
repository to obtain event and rule specifications. System components generate data and
events. Events are sent to RUBIES instances, which process the applicable rules, which in
turn may generate other events or perform method invocations. The monitoring and
debugging tool is used by developers to generate events and data, which may be sent to
other system components or instances of RUBIES. Monitoring data generated by RUBIES
is used by this tool to display system status to the user.

6. Validation and Demonstration: a Real-time Flood Alert System (RTFAS)
The development of a Real-Time Flood Alert System (RTFAS) has been chosen to validate
the ERF because it has ideal characteristics for an ERF implementation. This system: (i)
comprises many applications which reside in dissimilar platforms, (ii) requires the
processing of many simultaneous events, (iii) requires real-time processing of events, (iv)
requires the processing of different sets of rules which may involve multiple distributed
event services, (v) needs continuous monitoring for debugging and validations purposes.

RTFAS receives the data provided by the gauging stations of the U.S. Geological Survey
(USGS) and the Civil Defense (CD), and the images of the Doppler Radar of the National
Weather Service (NWS). This data will be processed to detect events and report
hydrological data and alerts to the Aqueduct and Sewer Authority (ASA), the Puerto Rico Electric Power Authority (PREPA), the CD and the NWS (see Figure 6). This information will be used by the different agencies for forecasting flash floods, monitoring rainfall events, and reservoir management control.

It should be noticed that, although RTFAS processes events in real time, the rate of event occurrences (1 event every 5 minutes) is many orders of magnitude than event processing capacity of ERF (20 events per second).

7. Implementation status

ERF is implemented in the Java language. Currently, there are two versions of ERF: ERF/RMI and ERF/CORBA, supporting the RMI and CORBA architectures, respectively. The CORBA version was implemented using dCon [18] which is an implementation of the CosNotification service. VisiBroker v4.0 was used as the CORBA development environment which is fully compliant with OMG’s CORBA specification (version 2.3) (For more details refer to http://www.borland.com/bes/visibroker).

The current RDL compiler was implemented mostly in Java, with the parser being implemented in C with lex and yacc. Event classes are specified in Java and compiled with a compiler built on top of a Java compiler. Both versions of ERF are available for download (free of charge for non-commercial purposes) at www.ece.uprm.edu/~jarroyo/erf.

8. Conclusions

In this paper, we have presented the Event-Rule Framework (ERF) Project. ERF has the following major research and development activities: (i) the development of a Rule-Based Intelligent Event Service (RUBIES), which is the heart of the system and processes distributed events by means of rules; (ii) an Integrated Development Environment (IDE) which comprises a set of tools for specifying, monitoring and debugging distributed systems; and (iii) a Real-Time Flood Alert System (RTFAS), which will be used to validate and demonstrate the functionality of ERF, as will allow to specify modeling requirements of RUBIES and the IDE.

Acknowledgements

This research has been sponsored by the National Science Foundation Next-Generation Software Program, Grant # EIA-9974979, with matching funds from the University of Puerto Rico.

References


Development and Specification of Interfaces for Standard-driven Distributed Software Architectures in the E-learning Domain

Luis Anido, Manuel Caeiro, Juan Santos, Judith Rodríguez
Departamento de Enxeñería Telemática
Universidade de Vigo
ETSE Telecomunicacións, Vigo, Spain
email: {lanido, mcaeiro, jsgago, jestevez}@det.uvigo.es

Abstract

This paper contributes to the e-learning standardization process with the definition of a service architecture to build standard-driven distributed and interoperable learning systems. The proposal presented is based on the definition of open software interfaces for each subsystem in the architecture, avoiding any dependency from specific information models. The selected approach to solve this problem relies on a systematic methodology for software development, which will support the identification of the services offered by particular subsystems in the architecture, as defined by the requirements established by users in the e-learning domain. The proposed methodology is based on the application of the Unified Software Development Process together with proposals from other authors like Bass, Clements and Kazman.

1. Introduction

Progress in Information and Communication Technologies fostered the development of a new generation of e-learning systems. Due to the integration of multimedia, networking and software engineering, Internet is today the most suitable environment for distributed learning. As a consequence, both public and private academic institutions rush to get the most from these technologies to offer better learning services.

In this scenery, e-learning systems proliferate. This fact, intrinsically positive, raises the question of interoperability and software reuse, which at this moment are far from being solved. In most cases, software developers and institutions deploy proprietary, even incompatible systems.

As in other fields, this situation triggered a standardization process, which in the case of distributed e-learning is rapidly becoming one of the key research initiatives worldwide. This process will enable software reuse and interoperability among heterogeneous systems, although a previous consensus on architectures, services, data models and software interfaces is needed. This article is intended to contribute to this process. Specifically, we propose a methodology for the development of distributed e-learning software systems, driven by use cases, architecture-centered, iterative and incremental.

We also present the set of software services defined by CORBAlearn, a draft proposal of specifications for the development of distributed e-learning systems in CORBA [1]
environments. The eventual objective of our work is to contribute to the creation of a new CORBA Task Force in the e-learning domain. In this line, we are taking into account recommendations made by the institutions involved in the learning technology standardization, and reviewing other active domain works [2] to extract design patterns from their specifications.

Section 2 briefly introduces the most relevant initiatives in e-learning standardization area. Section 3 presents the methodology used to derive the CORBAlearn framework, which comprises the Reference Model (section 4), the Use Case Model (section 5), the Analysis Model (section 6), the Reference Architecture (section 7), and the Design Model and specifications (section 8).

2.  E-learning Standardization

The e-learning standardization process is an active, continuously evolving process that will last for years to come, until a clear, precise and generally accepted set of standards for educational-related systems is developed. Organizations like the IEEE LTSC [3], IMS Global Learning Consortium [4], Aviation Industry's AICC [5], US DoD's ADL initiative [6], CEN/ISSS/LT [7] and many others are contributing to it, being the IEEE LTSC the institution that is actually gathering recommendations from other standardization bodies and projects. ISO/IEC JTC1 created in November 1999 the Standards Committee for Learning Technologies SC36 [8], to develop ANSI or ISO standards. Next paragraphs introduce the main areas of concern.

A key aspect in networked educational systems is to characterize, as precisely as possible, the educational contents offered to potential users. The trend seems to describe this information using learning metadata. The most outstanding contribution so far is the Learning Object Metadata (LOM) specification [9], proposed by the IEEE LTSC, which is becoming a de-facto standard. LOM defines the attributes required to fully and adequately describe learning resources.

Learner profiles schemas allow to characterize learners and their knowledge, and are used to maintain records of learners. The IEEE LTSC Public and Private Information (PAPI) specification [10] describes implementation independent learner records. Another important specification is the IMS Enterprise Data Model, aimed at e-learning administration, including group and enrolment management.

Other basic aspect subject to standardization is educational content organization, that is, data models to describe static and dynamic course structure. Static course structure defines the a priori relations among course contents (lessons, sections, exercises, etc.), whereas course dynamics determines the particular vision that users have depending on their attributes and previous interactions. In a similar way, learning environments need to understand the course structure to schedule the next student activity. The AICC guidelines for interoperability of Computer-Managed Instruction (CMI) systems, and the ADL's SCORM, based on the AICC specification, deal with this problem.

Other standards address content packaging to facilitate course transfer among institutions (IMS Content Packaging Specification), question and test interoperability (IMS QTI
specification), student tracking, competency definitions, and many others that are still in their definition process.

The more mature results are centered on the definition of information models to exchange learning resources. In most cases, XML [11] is used to define supporting information models enabling easy interoperability in WWW settings. No open reference architecture or service definitions have been proposed so far. The work that we present in this paper is a proposal towards such open architecture.

3. Notes on Methodology

The proposed methodology to identify a service architecture for e-learning has been guided by the Unified Software Development Process [12] and modelled using the Unified Modelling Language (UML) [13]. We combined the Unified Software Development Process with the recommendations by Bass et al. [14] to derive our software architecture. The overall process is summarized in figures 1 and 2.

![Figure 1: Methodology overall process I](image)

First, we identified a business model for the e-learning domain (step 1). This model corresponds to the Reference Model presented by [14] and describes the key agents participating in the e-learning process.

Once we have obtained a stable Reference Model, we proceed to functional requirements capture (step 2). This task is performed from use cases, considering the most common functional aspects in e-learning systems. Note that the eventually defined software services should be as general as possible, and therefore suitable for most learning tools.

From the domain analysis, we concluded that different e-learning systems define different sets of actors, although some actors appear in practically all relevant systems (e.g. learner, tutor). Additionally, some contributions to the e-learning standardization process address the identification of relevant actors and their responsibilities within e-learning systems. After a thorough analysis, we decided to use for functional requirements capture the actors
defined by the ERILE [15] specification. In other words, for each of the elements in the Reference Model we extracted the relevant requirements assuming the actors in ERILE were the target users.

Use case diagrams are the starting point to define an Analysis Model including all relevant classes needed to realize the identified use cases (step 3). Classes included in this model are implementation independent and purely conceptual. The analysis model will serve as the basis to define a Reference Architecture [14]. For this, analysis classes are grouped into service packages, which will correspond to subsystems in the Reference Architecture (step 4). In order to organize the analysis work, a set of analysis packages [12] are firstly identified. Classes not assigned to any service package form the \textit{wrapper}. The \textit{wrapper} provides value-added system-specific services.

![Figure 2: Methodology overall process II](image)

The Reference Architecture is a decomposition of the Reference Model into a set of components that provide the functionality identified along previous stages (steps 2 and 3). Note that in a general sense, our objective is the definition of a set of services to facilitate and speed up the development of standardized, distributed and interoperable e-learning systems. Thus, we will only include in the service packages determining our architecture those analysis classes offering basic, common services. Additionally, the selected classes should be reusable, independently from specific learning environments or specific value-added services. Specific subsystems in the architecture should be implemented in an integral manner. In fact, these subsystems model class groupings that typically evolve together.

Final applications, demanded and used by the actors in the use case model, will be composed by implementations of the defined service packages and the analysis classes external to them, that will constitute the \textit{wrapper}. As in other standardization fields, external components (i.e. external analysis classes) will provide specific features and value-
added services that characterize individual, maybe competing learning systems. In other words, this external wrapping will provide a differentiation among proposals from distinct developers and institutions.

The next step consists on the elaboration of a Design Model from the Reference Architecture (step 5). This model includes, as service subsystems, all corresponding service packages in the analysis model (i.e. components in the Reference Architecture). As our final objective is to develop a draft proposal for a domain CORBA facility for e-learning, we used the UML profile for CORBA to model artifacts compliant to the specifications in the facility (i.e. design subsystems). Each service subsystem is a monolithic unit that, when introduced in a system under development will enable new features or improve existing ones.

This proposal is materialized as a CORBA domain facility for e-learning: CORBAlearn (step 6). CORBAlearn is a specification where CORBA was chosen as the implementation/deployment environment. Services offered by CORBAlearn are used by developers of final e-learning platforms, who would add a wrapper that makes use of them and adds application-specific value-added services to differentiate systems from different vendors. Also, the core provided by CORBAlearn, whose specifications would be published in the framework offered by the OMG and its domain facilities [2], would support interoperability among heterogeneous platforms.

In the next sections we provide further insight on the results of this process.

4. Reference Model

The Reference Model is derived from the e-learning domain analysis [16],[17], the survey of the main proposals made by the institutions involved in the e-learning standardization process and the previous authors' experiences [18],[19],[20],[21]. This Reference Model would be clearly identified by experts in the e-learning domain.

As shown in Fig. 3, we identified three functional modules. Educational Content Providers (ECPs) are responsible for developing educational contents and offering them, maybe under payment, to the Educational Service Providers (ESPs). Authors, multimedia and pedagogical experts are the main actors who interact with the ECP. Standard course structure formats and metadata to describe contents must be used at the ECP.

Learners interact with ESP modules in their learning process, from course enrollment to graduation. These modules are responsible for providing: structured storage and management of learning objects; an on-line environment for the delivery of learning content; and administration facilities to handle the registration and course progress of educational sessions. Developers of ESP must use existing information models for course structures, learner records, tracking data, and ADL/AICC-like Runtime environments.

Brokers are responsible for helping both learners and ESPs to find suitable learning resources: learners will use broker services to find and locate those institutions (ESPs) that offer courses they are interested in; ESPs will use broker services to find learning resources
offered by the ECPs to enlarge their catalogues. In order to offer these services, brokers must maintain metadata information about courses offered by associated ESPs and ECPs.

![Figure 3: Reference Model](image)

5. **Use Case Model**

The Use Case Model was derived from a successive refinement process. As the use cases mature and are refined and specified in more detail, more of the functional requirements are discovered. This, in turn, leads to the maturation of more use cases and, even new actors may show up as generalization or specialization of others. Use cases are not selected in isolation, they drive the system architecture and the system architecture influences on the selection of the use cases. Therefore, both system architecture and actual use cases mature together.

In order to obtain our Use Case Model we carefully went over requirements for the ERILE specification [15] actors (learner, tutor, teacher, reviewer, author and administrator) and identified the elements of the Reference Model each actor interacts with. We also used the outcomes from the surveys on the e-learning domain and its standardization. For each pair actor-Reference Model element we looked for generic use case that encapsulated common functionality in standard-driven e-learning systems.

This process led to 215 use cases. As an example, Fig. 4 shows the use case diagram for the learner in his interaction with courses. There are use cases that capture requirements to navigate through the course structure and much other functionality that are required by learners in this scenario. Modelling was based upon UML diagrams, natural language descriptions and also a more formal description using a table-based schema [22].
6. Analysis Model

In analysis, we identified those classes that are able to cooperatively realize the functionality captured in use cases. From an initial analysis of the Use Case Model, we identified a set of analysis packages for each element of the Reference Model. Analysis packages group those use cases that are functionally related. Then, every package is analyzed separately in order to identify analysis classes for them. We have identified a total of 14 analysis packages: 2 for the Broker module, 7 for the ESP and 5 for the ECP.

Three types of classifiers are used at this step:

**Boundary classes**: Model the interaction between the system and the actor.

**Control classes**: Represent the needed coordination and sequencing between analysis classes in order to allow the realization of use cases.
Entity classes: Model the management of persistent information. Underlying information models are directly derived from the standards introduced in section 73.

Identification of analysis classes for every package is an iterative process. First, obvious entity classes were directly identified from the domain analysis (domain classes, e.g. Courses, Learner Records). Then, from the realization of every use case new boundary and control classes came up. In the latter step, we firstly tried to reuse existing analysis classes, sometimes adding new responsibilities. For those cases where this was not possible, new entity, control and boundary classes were identified.

6.1. Analysis classes for the Learning Environment

For the sake of illustration, Fig. 5 shows the identified analysis classes for the Learning Environment package of the ESP. Entity classes in this package encapsulate learner interaction data plus the preference configuration for the learning environment. Control classes here were based on the IEEE’s Learning Technology System Architecture [23], being able to deal with the main functionality required by learners while interacting with courses. Finally, there exist some boundary classes that represent learner user interfaces.

Figure 5: Analysis classes for the Learning Environment
This figure also shows relations among classes in this package and classes from other analysis packages (e.g. Coach class from the Learning Environment package needs to access entity class Course in the Courses package of the ESP).

6.2. Service Packages

The next step was to identify service packages. Classes in a service package have strong relationship among them, manage the same underlying standardized information models (c.f. section 73) and, therefore, tend to change together. They offer the basic functionality required to build final systems and allow those responsible for developing new e-learning systems to easily reuse service packages. Thus, final applications can be built incrementally and in a scalable way and the “time-to-market” factor is drastically reduced. All those analysis classes that do not belong to any service package are assigned to the wrapper, which provides value-added services that allow distinguishing between systems developed by different vendors.

As an example, Fig. 6 shows the identified service package for the Learning Environment: the Runtime Environment service package, which provides a set of services for creating customized learning environments. It offers the needed functionality to allow learners to navigate through the contents and to evaluate learner progress. We have included in service packages those classes whose assigned responsibilities are essential to build new systems.
Others that may be more system-dependent or represent added-value services were assigned to the wrapper.

6.3. Dependencies among Service Packages

A total of 13 service packages compose our Reference Architecture (see next section). There are several dependencies and relationships among the elements of the architecture. These dependencies are a consequence of use case realizations where classes from different service packages are involved. They are extremely important as they determine relationships among elements at the architectural level.

Fig. 7 shows the Runtime Environment dependency on the Course Repository service package, which is responsible for learning resources storage. For example the control class Coach needs to access the course structure encapsulated by the entity class Course. This dependency appears, for instance, in the Get Table of Contents use case realization (c.f. Fig. 8), where the Coach accesses to the Course Structure entity class from Course Repository service package. In any case, these dependencies do not suppose any drawback as they are always directed from “less” to “more” basic service packages (i.e. we can always rely on the existence of the service package that any other service package may depend on).

![Figure 7: Dependencies among service packages](image)

7. Reference Architecture

The Reference Architecture is a decomposition of the Reference Model, reflecting all the components supporting the model’s functionality, together with the corresponding data flows. It is obtained from a detailed analysis of the functional requirements, and includes further special requirements identified along this analysis process. Elements in the Reference Architecture are identified from loosely coupled service packages, which in turn are composed by tightly coupled components.
Our overall objective is the definition of services to speed up and simplify the development of standard-based, distributed and interoperable e-learning systems. The proposed Reference Architecture [14] is a service architecture enabling component reuse. This is the main reason to adopt as the Reference Architecture components the service packages identified along the Unified Software Development Process [12].

On the other side, the Reference Architecture is implementation independent and purely conceptual. We only identify basic responsibilities for software components and their interaction to comply with the requirements established in the Use Case Model. Decisions related to the eventual implementation environment are left to implementation-oriented design stages. Fig. 9 outlines the proposed Reference Architecture, including the Reference Model decomposition into Reference Architecture elements and the analysis package each service package belongs to.

The basic properties of this Reference Architecture are:

- **Service architecture based on reusable subsystems.** Architecture subsystems correspond to reusable service packages. The additional elements for final e-learning systems are boundary classes representing user interfaces and logic control for advanced functionality.

- **Standard-driven architecture.** Subsystems in the architecture correspond to components implementing business logic and information models identified by standardization bodies and institutions. However, no dependency to specific proposals has been established. Defined services include introspection mechanisms to find out the supported models.
Most modifications and updates will affect to one component. Dependencies in the existing information models and user requirements have been thoroughly analyzed prior to component decomposition.

Scalability. Construction of final systems can be done as an incremental process through the successive introduction of new elements from the architecture. Architecture subsystems are loosely coupled, and dependencies are directed from more complex to more basic components.

Adaptability. The clear identification of each subsystem's role through use case realizations, decomposition and dependency analysis supports the replacement of specific subsystems by functionally equivalent ones.

Interoperability among heterogeneous systems. Open software interfaces and a common Reference Architecture will support interoperation, provided final systems conform to...
the architecture. Added-value features are then implemented through the elaboration and improvement of wrapper classes.

8. **CORBAlearn: Design Model and Specifications**

Up to this point, the main outcomes have been a Reference Model, which identifies the main stakeholders and business processes involved in learning systems, and a Reference Architecture, where the identified processes are decomposed into a set of more basic building blocks providing appropriate services. Real interoperability on a middleware-based environment needs concrete definition of the methods that provide the identified services. This is done in design, where we fixed our implementation/deployment environment: CORBA. The main reason to select CORBA as our choice is the presence of domain facilities as part of the CORBA architecture. Domain CORBA facilities identify high-level services for a particular domain. Already existing facilities [2] include: CORBA Healthcare, CORBA Telecom, CORBA Manufacturing, etc.

Therefore, the purpose of the Design Model is to define the expected behaviour for each object that belongs to the software architecture. To carry out this task it is necessary to define object interfaces, using an adequate interface definition language, in our case CORBA IDL. IDL specifications are grouped according to the elements identified for the Reference Architecture. The whole set of specifications forms our proposal for a new Domain CORBA facility: CORBAlearn.

At this stage we used common design patterns in CORBA environments (e.g. factory objects or interface navigation mechanisms) and the UML profile for CORBA [23]. The design process is driven by the Reference Architecture and available standardized information models.

CORBAlearn covers all those aspects of a distributed e-learning system identified by the Reference Architecture. Each of them is supported by a different specification of the CORBAlearn domain facility. For the sake of brevity, we just present here part of the software architecture and its object IDL interfaces [25]. We encourage the reader to request the whole set of specifications [26].

8.1. **Design model examples**

**Interface specifications.** In design, we defined service subsystems from the identified service packages in analysis. For each of them, we developed a separate design model. As an example of the 13 CORBAlearn service specifications, we present in Fig. 10 the UML class diagram to model part of the Runtime Environment subsystem. Responsibilities are divided into a set of separate interfaces, each of them devoted to a specific purpose:

- **CoursePreferencesRepository.** It provides standardized management and access to learners' course preferences settings.

- **CourseEvaluator.** This interface defines methods to evaluate learner interaction with courses as a whole.
• **RuntimeFactory.** This factory interface creates the different objects needed to manage learning sessions for a particular pair learner-course.

• **NavigationManager.** It gathers operations needed to control learners' navigation through the course and track their evolution.

---

Figure 10: Learning Environment's UML class diagram
• **SCOInteractionRepository.** It offers mechanisms to manage the tracking data generated by learners during their interaction with SCOs.

• **PresentationManager.** This interface defines standardized methods to apply learner's preferences to course elements.

• **SCOEvaluator.** It specifies operations for SCO evaluation.

Each one of these interfaces includes a set of related methods. For each method, the specification defines the whole signature: method name, return value, parameters (in, out and inout) and the exceptions the method may raise. The interested reader can find in [26] the specification for the 888 methods included in the 69 CORBAlearn interfaces. The next IDL section shows an example of a CORBAlearn method signature:

```idl
CL_CommonTypes::FileDownloader getNextLearningElement
(out CL_CommonTypes::NodeId node_id)
raises(NoEntryPointAvailable, AccessDenied, NoMoreSCOs,
      CL_CommonExceptions::NotAvailable,
      CL_CommonExceptions::InternalError);
```

**Data model specifications.** Apart from the software services that are provided by IDL methods we also designed all data structures needed to encapsulate the data models involved. For this, we took as a reference current standards and specifications but without any tie to a particular one. For instance, Fig. 11 shows the IDL structure for a **Sharable Content Object** (SCO), which is the minimum content that can be delivered to a learner. The `SCOProperties <<CORBA::Struct>>` models the following IDL code:

```idl
enum SCOType{
    LEARNING, EVALUATION
};
struct Property{
    PropertyName property_name;
    any property_value;
};
typedef sequence<Property> PropertySeq;
struct SCOProperties{
    SCOType SCO_type;
    CL_CommonTypes::PropertySeq attributes;
};
```

† Sharable Content Object: the minimum entity that can be delivered to a learner and tracked.
The only parameter explicitly fixed is \textit{SCOType}, which can be \textit{LEARNING} or \textit{EVALUATION}. The \textit{Property} sequence provides an extension mechanism to include additional information. This mechanism to extend fixed properties is used along the whole CORBAlearn specifications to allow for further extensions.

Under the skeleton provided by this type of definitions it is possible to exchange standardized information models among heterogeneous platforms as part of service invocations. The UML profile for CORBA has been used to model all data structures defined in the domain facility.

CORBAlearn specifications also define introspection mechanisms to discover the actual information models a given component is able to deal with (partial implementations are also possible). In any case, service interfaces are independent from the information model used: operation signatures do not refer to any particular model. In order to allow the implementation to identify concrete attributes, CORBAlearn specifications contain definitions of information models potentially used by different component implementations. Thus, CORBAlearn clients know how to invoke operations according to the information models actually implemented.

\textbf{Working Prototype}. Defined services include common functionality for building \textit{Learning Runtime} environments. Developers of particular Web-based learning systems benefit from
the offered services and their reuse. Thus, time to market is reduced. As an example of applicability, we developed a Web-based courseware, the Learning Management System, tool that conforms to the US Department of Defense ADL runtime model [27] (c.f. Fig. 12). This model is bound to be accepted by the learning technology standards community as the common way for launching and getting lesson information in a Web-based distance learning environment. For this, we just needed to develop a thin layer (the wrapper) between the Web browser and the CORBAlearn server objects. Interactions between them are presented in the UML interaction diagram included in Fig. 13. The API defined by the ADL model (on the left column) can be easily implemented using the CORBAlearn objects. Sequence diagrams like this help developers of both CORBAlearn implementations and wrappers to understand how software services have to be implemented and used respectively.

Figure 12: ADL’s SCORM runtime model prototype
9. Lessons Learned

Together with e-commerce and e-banking, e-learning is becoming one of the killer applications for the Internet. Nevertheless, design and implementation of distributed learning systems is not supported by a convenient framework. As a consequence, many functionally similar, independent systems are being developed again and again.

As a first approach, formats are needed to establish, as precisely as possible, the syntax and semantics of information models in the e-learning domain. It seems that XML is taking the leading role to define the syntax of these formats. Open architectures should be addressed after information models have been clearly established. Active work in the IMS consortium shows this trend: preliminary works on the architecture were left apart to devote all efforts to information model characterization.

The next step should be oriented towards the definition of educational services to support interoperation at runtime among heterogeneous systems. These services should also support the development of new systems through the composition of reusable elements providing such services. With this work, we make a proposal in this line through the definition of a service architecture whose subsystems offer an open and clearly defined interfaces. This
architecture is system independent and can be particularized for different implementation/deployment environments. In this sense, we followed a MDA-like [28] philosophy, where an implementation independent model is obtained. This model is supported by a well defined methodology to derive software services from eventual user requirements. As advised by the MDA, UML was used to model the whole process. The next step is to define specific services for a concrete implementation/deployment environment.

OMG's CORBA technology provides a suitable environment where it is possible to build frameworks for domain specific applications. CORBA domain task forces identify software services to support component-based software development in distributed settings. Domain services are defined using IDL interfaces that act as contracts between component developers and component users. These users are, at the same time, developers of their own domain software products. As common functionality is provided by the domain facilities, time to market is drastically reduced. Additionally, clearly identified services enable interoperability at runtime among components from different vendors. Domain specifications mean a step further toward the standardization of the working area. Particularization of previously defined services for CORBA is modelled using the UML Profile for CORBA, following the recommendations made by the MDA.

The eventual outcome of this work is a proposal for a new Domain CORBA Facility: CORBAlearn. We also discussed the application of the Unified Software Development Process [12] to derive a domain-specific development framework and we established the relationship between the Unified Process methodology and Bass' recommendations [14]. With respect to the proposed methodology, we detected that the Unified Software Development Process does not offer appropriate mechanisms to develop service architectures. We complemented this process with proposals from Bass et al. [14] to obtain service packages as defined by the Unified Process. These packages define the Reference Architecture, the responsibilities of each subsystem, and the corresponding interfaces. On the other side, avoiding dependencies from specific information models allows reusing some elements in other application domains, in most cases with no or minor modifications. As an example, Broker may be applied to other domains where brokerage is present (e.g. e-commerce).

As the standardization process in the e-learning domain is far from being stable, CORBAlearn cannot be based on specific information models. As a consequence, we have defined generic structures (e.g. Category, Element, CourseStructure) and extensibility mechanisms to isolate interfaces from particular information models. This approach is similar to those that try to encapsulate information models defined in XML [29],[30]. Compliant implementations will select concrete proposals. We have also defined introspection methods that provide reflective mechanisms to identify which models are supported through the corresponding interface. This approach is also appropriate in other domains where no mature proposals are available. On the other side, we have identified some converging trends. This information has been used to derive the draft proposal for a domain CORBA facility.
Acknowledgments

We want to thank “Xunta de Galicia” and “Ministerio de Ciencia y Tecnología” for their partial support to this work under grants “Arquitecturas distribuidas para Teleservicios” (PGIDT00TIC32203PR) and “CORBALearn: Interfaz de Dominio guiada por Estándares para Aprendizaje Electrónico” (TIC2001-3767) respectively.

References


Session III – Components and Adaptations
A Component-Based Architecture for Collaborative Applications in Internet

Flavio DePaoli

Dipartimento di Informatica, Sistemistica e Comunicazione
Università di Milano Bicocca
Via Bicocca degli Arcimboldi 8, 20126, Milano, Italy
email: depaoli@disco.unimib.it

Abstract

As Internet has become the de-facto standard for any modern application, software needs to be reconsidered to address collaboration issues. Aspects like awareness and knowledge management are becoming vital to effective interactive software products. In this scenario, middleware plays a key role as enabling technology toward the goal of development of integrated workspaces. This paper describes a novel collaborative software system and discusses its component-based architecture. The aim of the system is to integrate dispersed knowledge to deliver comprehensive workspaces. Component technology enables the development of open systems that can be easily modified to accommodate both new features and new interaction styles.

1. Introduction

Today, the framework to develop interactive applications is the Internet. This assumption implies that applications are distributed; therefore, middleware platforms have assumed a key role in current software development. Traditional software was based on the operating systems features, since it was conceived as a monolithic entity. In facts, aspects like modularization and separation of concerns – functional and non-functional aspects – were confined to the early phases of the development process, often the source code did not reflect such aspects since the provided constructs did not address these issues. Middleware environments, like CORBA and DCOM, suffer from similar limitations: they reproduce in a distributed environment the computational model of object-oriented languages like C++ or Java. In CORBA and DCOM, an application is composed of a set of freely designed objects that communicate each other by remote procedure calls. In other terms, CORBA and DCOM provides basic communication and naming services, and let the designer address issues like persistency, synchronization, and session.

To overcome these limitations, component models have been proposed. Such environments provide designers with a richer support to address modularization and composition through the definition of the internal component architecture with a set of built-in services. For example, Enterprise Java Beans (EJB) provide persistency, session and message services, over basic services that address common features like naming, access control, and security. Even if component models are not mature yet, the approach is promising. At one side, it supports modularization and separation of concern issues at any level of the development processes – at run time, it is still possible to clearly identify components. On the other side, services are properties of the type of components – they are automatically and
autonomously associated with components by the environment and are not included into the component code.

This paper aims to present and discuss the proposed architecture for a collaborative, knowledge management system, named SKOWE – Shared Knowledge Organizer and Workspace Environment, which is under development at the Università di Milano Bicocca. The SKOWE systems will provide users with an integrated environment to create and access both personal and corporate knowledge in a comprehensive way. The SKOWE project relies on previous experiences on the development of collaborative and knowledge management systems. In particular, it follows the Esprit project Klee&Co that has delivered an innovative prototype to create collaborative workspaces based on knowledge and awareness.

The SKOWE project deals with the development of a knowledge base system that provides groups of users with integrated workspaces to interact and share information. The system is web based and has the ambition of understanding the user activities and behaving accordingly to provide the best support in any situation. “Best support” means to provide users with all possibly needed information in suitable format [8]. User workspaces should adapt their aspect and their content on the base of the environment and the context of use. For example, when a user is working on her workstation, say writing a research paper, she may need information about related projects carried on either inside or outside her organization, and related materials -papers, software, pictures, movies. She may need to know who are the persons involved in similar research projects and who are experts in the fields, whether they are on-line and available for chatting or their e-mail address. When she is drawing a figure, she may need to be told about similar drawings, and so forth. In a word, this means that users need to be aware of what is going on around them to be able to get advantage of this knowledge.

The SKOWE architecture has been recently proposed as reference architecture for the Esprit project MILK. MILK aims to design a server-based system that is accessible from remote locations, including office rooms and open spaces in company sites, and employees’ houses, to provide users with remote workspaces. Moreover, users will carry on the KM processes through an interconnected solution even when they are outside institutional spaces, like when they travel. MILK will also develop and validate methods encouraging knowledge sharing both in a multi-functional intra-organization way and within inter-organizational knowledge chains of companies with customers and business partners. The architecture is web-based to ensure accessibility, takes advantage of Java/EJB and XML middleware technology for the application logic, and is independent from the information management systems at the back end.

The next section briefly describes the SKOWE project to identify the system requirements. Then the architecture is presented (section 3). The paper ends with a discussion (section 4) and conclusions (section 5).
2. The system requirements

Networked computers open the possibility to share information and collaborate. For example, a project is usually a complex task accomplished by a team of persons that need to collaborate to achieve a common goal. Collaboration may involve several activities: share a common knowledge, exchange information, close interaction to produce an artifact, establish a common plan, and so forth. Collaboration does not exclude personal working, which anyway should be shared somehow to be included in the common project knowledge. The ideal system should provide project members with private workspaces that are integrated to build up a common environment.

To address these issues, basic aspects that have to be considered are the possibility to collaborate in real time, the capability of capturing and maintaining the knowledge, and the capability of making this knowledge available to others.

Real time collaboration is the capability of supporting two or more people working at the same artifact at the same time. The point is that such shared workspace should be as much as possible similar to their private workspace. Moreover, real time collaboration requires awareness about what other users are doing and support for direct communications. For example, a user needs to know who is on-line at any time, and whether or not she is available for chatting. If requested, the system should be able to support communication among users within its framework.

A relevant aspect of knowledge capturing and maintenance deals with understanding and relating document contents. Document parsing is a way to get information on the content, such as keywords and summary, and enable for document indexing. Moreover, information like the profile of authors, the context in which the document has been produced, and so forth allows for the definition of a document profile. On the base of profiles, it is possible to define relationships among documents. Relationships may deal with several aspects, like temporal sequences (e.g., a doc may be a version of another or written before another), similar content (e.g., a doc deals with the same subject as another, or it is part of another or it details some element of another), same context (e.g., a document is a comment to another document, or it may be an e-mail message associated with it).

The concept of profiling can also be extended to represent and include other entities in addition to traditional documents. An example is the inclusion of people: user profiles allows for classification of users according to their interest, expertise, role and so forth. Profiles should be dynamic to follow the evolution of both users and environment. This approach supports the definition of relationships among any kind of entity. For example, the expertise relationship relates a document with a set of people, known to the system as experts at the content of that document.

Making knowledge available to users deals with knowledge diffusion and presentation. Tools that support document indexing based on keywords and automatic summary already exist, but they are either immature or focus on document searching. For example, MS Word processor supplies a service of automatic summary that is exploited for searching only.
The approach is to reverse the roles: the computer has to suggest solutions to users. This means that the system is expected to supply users of knowledge information related to their activity. The Klee&Co system introduced the concept of “view-with-context” to define a novel presentation style that consists in displaying documents surrounded by any related information. Therefore, users do not have to look for information that they may need. Figure 1 gives an example of a “view with context” interface.

The advantage is twofold: users save time and effort to look for useful information, and, most important, they may get information that they didn’t think it available. The latter point is very important. Searching activity assumes that searchers already know what to look for, but often users are not aware of what information may be searched simply because they cannot know what is available.

The “view-with-context” navigation style enhances the knowledge of users by making them aware of what exists, that is, by providing an in-depth knowledge about a topic. For example, the view-with-context may display a person profile surrounded by related information: documents she has written or edited, communities she belongs to, expertise, interests, and so forth. To avoid the risk of supplying useless information, knowledge information need to be filtered according to some criteria, like the profile of the user, the activity she is carrying on, and the context in which she is acting.

In conclusion, the SKOWE system requires the integration of several and different functionality and services in an open way to easily address maintenance and evolution aspects. In other terms, a goal of SKOWE is to provide a component-based architecture in which components can be added and replaced to improve and extend existing functionalities.

3. The system architecture

The SKOWE project defines an open architecture and a set of specialized components that can be composed to deliver tailored systems. The overall architecture is multi tiers as
sketched in Figure 2. To point out the distributed nature of the architecture two sites are represented with interconnection among them. Users access the system by an interaction manager that provides them with presentation and navigation facilities. Typically, it is integrated in a web server to enhance availability. A collaboration manager addresses those concerns related to the presence of other actors within the system. Therefore, they are in charge of exchanging information to ensure a global view to users, and controlling the access to shared resources. Such managers rely on Knowledge Management Engines that have the task of capturing and maintaining the knowledge by monitoring activities and parsing documents to collect information. Moreover, they have to combine such information to identify relationships. KM engines rely on a set of Knowledge specific services and Document Management Systems (DMS) to store information.

![Figure 2 The SKOWE architecture](image)

### 3.1. Interaction Managers

Presentation and interaction issues are crucial aspects for SKOWE. In fact, many systems are just unusable due to their poor interface. Interaction managers are devoted to those tasks. They should deliver comprehensive interfaces that include heterogeneous but correlated information. The approach is to present each item (a document, a drawing, a e-mail message) surrounded by its context, which means representations of the related information (related documents, people, comments, e-mail message) to create awareness. An example is the Klee&Co interface already presented in Figure 1.

Interaction managers are composed of a set of user components, which represent human users, and presentation components, which organize and present the information to users. A user component defines the interaction rules with other users and the selection rules for available knowledge. Based on personal profiling and context of use, the system provides users with personalized behavior.

Among others, personal profiling defines rights (e.g., security info, access rights and group memberships) and expertise (e.g., roles and fields of interests). According to profiling, available knowledge is filtered and selected to meet the user needs. The context of use defines what and how has to be presented. For example, browsing and working situations may require different system behavior: working means that the user may need to edit a
document and therefore access sources to make up the document, while browsing means that the user may need related information in read-only mode. Another example is given by the use of the system during a meeting: documents should be proposed in suitable formats for projection, e.g., a slide presentation or similar.

Besides PC-based interface, interaction managers could be designed to address different terminals, such as community walls, Personal Digital Assistant (PDA) and cellular phones. The development of interaction managers is part of the MILK project. The aim is to provide users with contextualized interfaces in different situations. Since interaction managers are independent components, they can be developed to address interaction styles and content presentation in accordance with user requirements, either technological or functional, without affecting the rest of the system. Figure 3 sketches the MILK architecture, which highlights the three interaction managers. MILK architecture is a centralized instance of the SKOWE architecture, without collaboration managers.

### 3.2. Collaboration Manager

Collaboration managers have the task of managing the knowledge sharing. The purpose is to deliver middleware systems that can support different configurations: private workspaces, centralized shared workspaces, and distributed collaborative workspaces. In the former cases, the collaboration tier might even be eliminated, as in Figure 3. In the latter case, the collaboration tier plays a key role having the tasks of spreading and collecting knowledge.

Let us examine the tasks in detail. Each user wishes to access a global view of the knowledge managed by the system. Therefore, the knowledge managed at a site has to be made available at other sites. A collaboration manager has the task of notifying other collaboration managers about the information managed on its location. This happens by exchanging messages including concise information about, for example, an available document. In such a way, each collaboration manager is aware of what is available from every other location. Using such concise information, collaboration managers can supply interaction managers to let them present comprehensive views. Moreover, this information can be cached locally (by the KM tier, see below) to maintain a global view of the system, without affecting the rest of the system.
even if full information might not be available. In fact, it is important to users be aware of the existence of something that can be accessed on request.

In a collaborative environment, users should also be aware of actions performed by others. Collaboration managers are in charge of notifying actions performed by users. For example, a collaboration manager sends information on connected users to another collaboration manager to let them communicate.

3.3. KM Engine and Services

The system knowledge is captured and maintained by the Knowledge Management (KM) Engine tier. Information flows from the other tiers to KM engine that use it to update the knowledge base, to discover relationships and so forth. Document and user profiling are example of activities belonging to KM engines. Documents are parsed to extract information like keywords and summary. Actions performed by users are parsed to deduce information like frequency of usage and relevance of documents or user expertise. For example, assume that a conversation between users A and B has occurred; and that the subject was section S of document D. The profile and the knowledge base associated with document D will be updated, as well as the profile of users A and B [9]. KM engine activities rely on a set of KM services that provides for utilities like document parsing and relationship computing. The use of services external to the engine has the advantages of making their development and deployment independent.

4. Discussion

To deliver a system like the one outlined so far, several issues have to be considered, ranging from application functionality (e.g., indexing, activity tracking, profiling) to system properties (e.g., flexibility, availability, scalability, security, performance), to social requirements (e.g., privacy, relationship among users, collaboration, knowledge sharing). Most of the existing systems address only some of these issues; the challenge of the present project is to define a comprehensive framework to develop applications that fulfill every requirement. A leading criterion for system development is to use standards to address openness and integrate, whenever possible, existing systems and applications.

In this discussion, two key issues are considered: knowledge management and knowledge diffusion. Over the functionality aspects, issues related to Internet have to be addressed. The reference scenario for SKOWE is enterprise-wide systems that include local installations – e.g., LAN connecting a set of users – interconnected via Internet. Users are registered and form communities that collaborate toward common goals.

There are examples of Knowledge Management tools, like Autonomy [3] and Verity [4], that provide effective support for knowledge extraction and indexing, while others, like Livelink [1], and DocuShare [2], are document management systems that provide support for different kind of navigation. Those systems have been designed with two-tier client-server architecture, whose clients are web browsers that interact with a HTTP server. Instead, we are willing to develop a multi-tier architecture with a clean distinction among presentation logic, business logic, and services. Each tier can be viewed as an independent application to be accessed in different ways and included in different architectures.
The KM Engine has been designed as a collection of EJB components that implements functionalities available to interaction and collaboration managers. The advantage is to support different kinds of deployments and different access modes. A typical scenario envisions deployments of KM engines within a LAN to support local users, and to provide access to local knowledge to remote users (i.e., users outside the LAN). The deployment within a LAN should be tailored to address local needs. For example, it would be possible to have a KM manager per workstation if users like better to keep their files locally. Another possible configuration is to deploy a centralized KM manager to support small communities of users. EJB components provide for logical names that make transparent to users the system configuration. Moreover, local access can be directly supported by Java/EJB interface to ensure good performance to clients within LANs. Moreover, since EJB are accessible through the standard IIOP protocol, clients can be written in Java or with any other technology supporting IIOP.

EJB access may not be suitable to support Internet access. Firewalls and different technology environment might pose restrictions that can be addressed by generic standards. XML technology enables content encoding and protocols as XML/RPC and SOAP provide suitable transport means to implement communication. Both protocols can take advantage of the popular HTTP protocol to facilitate user access. The major disadvantage of HTTP is poor performance that is often overtaken by advantages in availability and reduced security problems. KM engine has been designed to provide clients with a XML/RPC interface to facilitate Internet access.

The KM engine prototype, which is under development within the MILK project, takes advantage of the BSCW system to store, browse, and search of documents and associated information. The BSCW (Basic Support for Cooperative Work) shared workspace system [11] is concerned with the integration of collaboration services, which include features for uploading documents, version management, group administration and so forth. BSCW has been included as a component of the KM engine. EJB components mediate clients’ requests to address system requirements. For example, uploading a new document is not just storing it. The document profile needs to be constructed, hence the document has to be parsed to extract knowledge, the user may insert further information, context information need to be collected, and so forth.

Knowledge specific activities are carried on by KM external services. Such services can be deployed differently from KM engines. A typical scenario could be to install them on LAN servers. This solution has the double advantage of reducing the load on workstations and centralizing tool maintenance. Activities as document parsing and indexing are time consuming, hence powerful servers could be delegated to this purpose. Current service prototypes, for example, have been implemented by including the document parser KEA – Automatic Key phrase Extraction [6] [7]. This service implementation is representative to legacy system inclusion. The first version of KEA was on Unix, so we had to implement the document parsing service as legacy system via Java beans. Then KEA was ported in Java, so we moved to this new version, but without changing the accessing bean, and therefore the clients. The approach is to provide users with multiple interfaces to services: an XML-based interface (via SOAP or XML-RPC) and a Java interface. The former is to
promote access to any kind of client, the latter to support inter-component access effectively.

Once knowledge has been captured and stored by a KM engine, it needs to be spread over the SKOWE sites to provide users with comprehensive views. A key point to address this issue is the need of message exchange among collaboration managers. There are several research efforts to develop models and implementation of message-based services. Examples of publish and subscribe services range from proposals for industrial standards such as CORBA notification service [12] and Java Message Service (JMS) [13] to research prototypes such as Siena [14] and JEDI [15]. Peer-to-peer computing is another message-based communication model that is addressed by projects like JXTA [16] and protocols like Gnutella [5]. Besides basic services, there are systems that explicitly address content notification. Examples are NESSIE [17] and Yaka [18]. The approach is to search various kinds of data repositories and send notification messages about new documents to subscribed users. Such centralized systems serve registered end users by letting them pre-select subjects and notification media (usually e-mail messages). The goal of SKOWE is to embed notification services to make them transparent to the users, which hasn’t to be aware of the distributed configuration of the underlying system.

In SKOWE, collaboration managers act as peers that send and receive messages to share knowledge. Data flows from KM engine to the associated collaboration manager that forwards it to other collaboration managers. Figure 4 illustrates such a situation and highlights the different kind of communication in LANs and in Internet. Collaboration managers have been implemented as message-driven EJB to be able to react on the reception of new information and update the local knowledge. Such a communication model is suitable within LANs, since it requires centralized queue managers. To support Internet communication, Message Handlers with peer-to-peer communication capabilities have been introduced. A message handler act as queue manager with respect to EJB that reside in the same KAN, and as peer with respect to message handler located in other LANs. In such a way, message handlers form a logical network over Internet to support knowledge diffusion efficiently. Remember that, as discussed above, only concise profiling information is spread over systems sites to minimize communication overheads.
Over functionalities, system properties (e.g., flexibility, availability, scalability, security, performance) are key factors for a successful framework. Availability is ensured by the replicated nature of the SKOWE system: each site is autonomous in providing applications with concise knowledge information. The size of such information is tiny enough to minimize the overheads. Full availability is then achievable through common middleware facilities. User mobility is also supported: due to global, shared information, a user may connect to any site and retrieve her environment.

Scalability has been constantly taken into account when devising the architecture. The presence of multiple sites that provide replication and several access points addresses scalability. In fact, users may maintain a personal, yet concise, replica of the system to let them decide what information is needed. Therefore, documents are accessed (e.g., downloaded) only when necessary to reduce network traffic. Moreover, caching can contribute in reducing network traffic over the Internet.

Performance is strictly related to the above issues. Keeping the communication overhead at the minimum enhance performance. Exploiting the local EJB communication facilities ensure efficiency to inter-tier communication at a site. Moreover, information caching has been adopted to enhance performance and availability.

Security is a general problem that needs to be addressed for any Internet application. Common issues are authentication and access control. In our case, the problem becomes more challenging since we deal with knowledge, which means in-deep understanding of activities and contents. Classic approach to regulate access may not be enough; new collaboration models and ethics need to be devised and implemented. Therefore, the security issue will be further investigated in the future.

5. Conclusions and future work

The paper aimed at presenting a novel system that deliver collaborative workspaces to users. The SKOWE system is based on knowledge sharing to promote awareness and to make information available in such a way that users can learn from the system.

A key point for SWOWE is to deliver an open architecture that can be customized to accommodate different collaborative applications and services, and to meet user requirements in different environments. The adoption of a component-based middleware addresses this issue by enhancing the possibility of adding/replacing components to define the requested system. As EJB and related technology are not mature yet, further studies and experiments are needed to prove that advantages of using a high level paradigm does not over penalize performance and efficiency. Moreover, some issues, as security and multi-point message exchange, need to be further investigated to ensure the right quality of service. The resulting solution is a combination of standard component technology augmented with services to deliver a reach framework to application developers.

Preliminary prototype implementations, carried on in Klee&Co and MILK projects, have demonstrated the feasibility of the proposed solutions. Future work deals with completing the development and making tests to evaluate SKOWE from several points of view, such as usability, effectiveness and privacy with respect to user requirements; security, scalability
and performance with respect to engineering requirements. Long-term goal is to consolidate the architecture and address the development of a middleware that provides application developers with standard knowledge management services.

**Acknowledgments**

The present work has been partially supported by European IST Project n. 33165 MILK – Multimedia Interaction for Learning and Knowing. The author wishes to thank the project partners and the colleagues Alessandra Agostini, Luca Bernardinello and Giorgio De Michelis for their valuable collaboration.

**References**


Composing Distributed Components with the Component Workbench

Johann Oberleitner
Institut für Informationssysteme
Argentinierstrasse 8/E1841
A-1040 Wien, Austria
email: joe@infosys.tuwien.ac.at

Thomas Gschwind
Institut für Informationssysteme
Argentinierstrasse 8/E1841
A-1040 Wien, Austria
email: tom@infosys.tuwien.ac.at

Abstract
Although software components have gained importance, support for the composition of distributed components is still limited. Worse, if components implemented for different component models need to interact with each other, the composition process becomes a nightmare. Though, bridging technologies for different component models have been standardized, almost no implementations of these exist so far.

In this paper, we present the Component Workbench (CWB), a flexible toolkit for the composition of components. Due to CWB's modular design and a generic component model used for the internal representation of the components, it supports the composition of components implemented for different component models.

1. Introduction
Component models support the developer in the design and implementation of components adhering to a common architectural style [11,18]. Using components developed for the same component model eases the task of the developer because it helps to avoid architectural mismatch [10] if components developed by different vendors are being used.

Today's component models can be distinguished between desktop component models [16], which are also referred to as local component models [8], such as JavaBeans or ActiveX controls and distributed component models [8], such as the CORBA Component Model (CCM), Enterprise JavaBeans (EJBs), or COM+. Desktop component models are typically used in combination with integrated development environments (IDEs) and supply the programmer with user interface elements such as buttons or list boxes. In the following, however, we focus on distributed component models.

Distributed component models are based on middleware technologies providing fundamental services that conceptually operate between the operating system layer and the application layer [2]. Such services provide transaction management, persistence, or
security services, hence simplifying the implementation of distributed components rather than facilitating the implementation of clients using these components.

Components implemented for different component models cannot easily be mixed due to the models' type systems and APIs that differ considerably. Additionally, some component models, such as the Enterprise JavaBeans (EJB) component model, are tied to a specific programming language [5]. While in theory bridging technologies exist to solve some of these problems, in practice their use is rather limited.

For instance, they provide no support for transactions across component model boundaries. To solve these challenges, we have developed the Component Workbench that will be presented in the following sections.

The outline of the paper is as follows. In Section 2, we present our terminology. In Section 3, we present the interworking problem of today's component models. Section 4 presents the Component Workbench and how we have solved the above mentioned problems. An evaluation of the design based on a library administration application is given in Section 5. Future work is presented in Section 6 and related work in Section 7. Finally, we draw our conclusions in Section 8.

2. Terminology

Different researchers have adopted different definitions of a component. Hence, for reasons of clarity, the following paragraphs present the terminology used within this paper. This terminology has been adopted mostly from [4,22].

A software component is a software element that conforms to a component model and can be independently deployed. Distributed components are provided by Enterprise JavaBeans (EJBs), the CORBA Component Model (CCM), or COM+. A component's client interacts with the component through interfaces. These interfaces provide an abstraction of the functionality of the component. Typically, such an interface provides methods to invoke a component's operation, read or change a property of the component, or to inform the component about event handlers where the component can notify its clients about the occurrence of events. In this paper we refer to operations, properties and events of a component collectively as features.

A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms it uses to interact with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Different organizations have defined and realized different component models.

Application builders are tools that can combine components from different developers or different vendors to construct an application. Recently, application builders from commercial vendors provide support for distributed component models along with associated services.
Interoperability denotes how different implementations of the same middleware specification work together. Interworking means the integration of middleware systems that implement different specifications [7].

3. The Interworking Problem

The problem of today's component models is the lack of interworking of components implemented for different component models. While components implemented for the same component model can be composed easily by implementing pieces of glue code, this is not the case for components implemented using different component models.

Theoretically, it should be sufficient to only know the interface of a component and to be able to use it like any other component. In reality, however, the composition requires the developer to be familiar with all the different component models being used. Hence, it requires the developer to deal with different type systems used by different component technologies. To be able to use an enterprise bean in cooperation with a remote COM+ component the developer has to write low-level code such as Java Native Interface (JNI). Microsoft's Java Virtual Machine (MS JVM) used to support COM+ but lacks support for RMI and unfortunately is no longer supported. Hence, implementing such an application is the ideal nightmare even for professional programmers.

To ease the integration of components developed for different component models, we have designed and implemented a new toolkit, the Component Workbench (CWB). While implementing the CWB, we had to solve the challenge to provide a coherent, easy to understand, and powerful interface for developers. No matter of the component model a component belongs to, the developer has to be able to use it using the same interface. The interface has to be as easy to understand as if the programmer were only using CORBA or EJB components. Yet, the interfaces provided have to be powerful enough to accommodate the different features provided by today's component models. Hence, it is not enough to restrict the developer to use only the subset of features available by all of the component models.

4. The Component Workbench

In this section the main parts of the architecture of the CWB are explained. As figure 1 shows the architecture consists of several layers where each layer sits on top of another. Although we have implemented this architecture using the Java programming language the architecture can be applied to other programming languages as well.

For each component model there is a wrapper that realizes a common interface for the corresponding components. The design of the wrappers as well as the uniform component access is explained in section 4.1 and 4.2. On top of this layer we have arranged the scenario layer and the adapter facility of the CWB. Both are explained in section 4.3 and 4.4. The CWB tool layer consists of those parts of the CWB that are related to the user interface and the configuration of the application. A description of the design of the user interface is presented in [16].
4.1. Component Wrapper

Each component model provides its own mechanism to access the features of a component. To provide uniform access across different component models we have specified component wrappers. Component wrappers create a consistent view of different component models onto a concrete internal component representation.

We have defined several categories of functionality that have to be supported by the component wrappers:

**Instantiation**: The instantiation functions are responsible for the creation of a component or the assignment of an already existing object to a component wrapper. For the instantiation process additional information has to be provided such as the application server that has to be used or the naming information that is required to connect to a component's application server.

**Feature Access**: Since every component model supports different feature categories the component wrapper has to provide methods to find out about the supported feature categories, and the elements of each category. For example it has to be possible to access all operations a component provides. A filter criterion can be used to restrict the required features to only a subset of the available ones. Additional information about a feature can be retrieved too, such as a set of allowed filter values for a particular feature category.

**Graphical Representation**: Graphical application builders need a way to show the components on the desktop. Since distributed components typically have no client-side GUI representation, the wrapper should provide a useful representation. The advantage of providing a visual representation by the component wrapper is that it can show additional information to the user that is dependent on the wrapped component instance such as important component property values.

**Configuration Panels**: It must be possible to add components to a building tool as well as to configure the settings of a component model. This can be done with configuration panels provided by the component wrappers, too.

The internal design of a component wrapper (Figure 2) is a combination of the factory and the bridge design patterns [9]. The methods that have to be implemented by a component
wrapper to provide the above functionality is specified in the IComponent interface. Within the CWB, this interface is used to access a component in a standardized way for all supported component models. Hence, if support for a new component model has to be added to the CWB, only the IComponent interface has to be implemented.

4.2. The Generic Component Model

The API of the generic component model resembles a reflective API that can be used for components of arbitrary component models. Since this is inconvenient for many situations we have provided a set of classes that help the programmer to find and access a particular feature. The Uniform Component Access Layer consists of these classes and a mechanism to load the parameters for the instantiation of a component in a consistent way across all component models. The most challenging task of this layer is the provision of a uniform type system.

Since different component models support different features, we have defined abstractions for the most frequently used features. We have provided interface definitions for the access of properties (attributes), methods (operations) and eventsets (callbacks). Each component wrapper has to provide implementations of these interfaces. The features provided by a component can be queried using the IComponent interface that has to be implemented for each component model. The implementations of these interfaces uses the meta-information and the access mechanisms provided by the corresponding component model to provide access to the corresponding features. Figure 2 shows the relation between the IComponent interface, the interfaces describing various features of a component model and the implementation of the EJB component wrapper. Since the EJB component model does not support events there is no implementation of this feature category.

![Figure 2: Internal Design of a Component Wrapper](image)

The IComponentProperty, IComponentMethod and IComponentEventset interfaces provide operations for the access of properties, methods, and eventsets accordingly. Among the functionality of these interfaces are property read/write access, method invocation, and eventset connection. It is possible to implement IComponent to add new features at runtime with the feature access mechanism.
The component wrappers shield the developer from having to do the complicated work himself. When user interface elements are placed onto the scenario, the Enterprise JavaBeans component wrapper uses reflection to query the components for the methods and properties they provide. COM+ components that are placed onto a CWB scenario make use of COM type libraries. CORBA objects that are used within the CWB use CORBA’s interface repository to obtain the properties and methods provided by the component.

4.3. Scenario

Components can be selected, configured, arranged and deployed in scenarios. Hence, a scenario acts as a container for the components. CWB modules such as user interface elements or the export generator access a scenario via the IScenario interface. We have provided an interface instead of a class to reduce coupling and provide more flexibility for future extensions. The scenario interface specifies different methods for adding, removing, or querying for a particular component instance.

Furthermore the scenario is responsible for the persistence of component configurations. Some attributes that are important for a component's configuration cannot be stored within a component such as the user-defined name of an instance within CWB or the geometry of the graphical representation. Though these properties are not relevant for the component itself, they are of relevance for the application built from these components. A scenario can attach arbitrary objects to a component within a scenario.

In the CWB a scenario is represented as a graphical diagram that contains all components with their connections. Within this diagram a component can be selected and modified. The appearance can be manipulated too. The components involved in a scenario are already functional when they are instantiated.

4.4. Adapter Facility

Once different components have been selected and placed onto a scenario, these components are connected by the user to interact with other distributed components or with GUIs. For this purpose a flexible adapter facility has been integrated into the CWB to provide for a variety of connection styles between components.

At the time of writing we support adapters that are capable of reacting on events that are emitted by a component and propagate these events by invoking a method of another component. A user interface wizard can automatically create these adapters. The adapter is compiled from the CWB, instantiated, and the appropriate components are connected with the use of the adapter and an eventset of the component that emits the events. After an adapter has been created it can be reused for other components or for different scenarios at all. At the time of writing we just support adapters that connect one method of a sending event — this is exactly what the untyped CORBA event service provides [13] — to one method of a target component. In future releases, however, we will try to create adapters that support more complex connection patterns, such as the mapping of whole event interfaces to target components, something the COM+ event service provides [15].

Since it cannot be assumed that the parameters of the events of the sender components will fit to method parameters, we have realized different ways for solving this problem. It is
possible to change the automatically generated adapter code within a code window before it is stored and compiled, and we support typed-based adaptation that can be used to connect a chain of predefined adapters to connect components [12].

4.5. Export Generator

Once a component scenario has been designed and tested export generators can be used to create applications from scenarios that are capable of running without starting the CWB. We have provided a general interface to support a variety of export generators to cover different application types.

A simple variant of a code generator creates Java code that interacts with the application servers in a similar way, as the CWB would do. In this case the necessary glue code to connect the wrappers is created and compiled. Optionally this code is packaged together with the required wrappers into a Java archive file.

A more sophisticated code generator simplifies how the components are accessed: instead of using the access functions of the component wrappers, code can be created that accesses a component directly. Depending on the model, Java code is generated or in case of COM+ Java code with some related C++ code accessed via JNI is created. Depending on the relevant component models different libraries have to be deployed for compilation and runtime access of the components and/or the calling code between two components.

Another export generator creates scripts in XML form that can be interpreted by a tool that acts as a player for these scripts. These scripts contain the configuration and connections of the involved components in a similar way as provided by the Bean Markup Language (BML) [23] or the long-time persistence format [21] of the JDK 1.4. The advantage of our format is that it allows not only the configuration of JavaBeans components but also the configuration of components of arbitrary component models. In all cases the exported functionality can be deployed on hosts that are used for business logic.

5. Evaluation

To verify the design goals of the Component Workbench, we have implemented a small business application that uses COM+ components and EJB components. For the developer using the CWB, however, all the properties, methods, and other features of a component look the same regardless of the component model a component adheres to. To demonstrate how the CWB supports the composition of components that have been built and designed for different component models, we implemented a library administration tool for our department. Figure 3 shows the architecture of the library administration tool.
For the implementation of the library administration tool, we used Enterprise JavaBeans handling the interaction between the library database and a web service used by students. The web service is driven by these enterprise beans. The library administration tool also has to interact with an accounting program based on COM and running on Microsoft Windows. The accounting program is used to deal with the fees applied to students that do not return books within the predefined time limit. In addition it supports to enter the payments for newly ordered books.

The Accounting Application periodically emits events with its COM+ interface whenever the library's database should be checked for users that have exceeded the allowed lending time. These events can be accessed within the CWB to connect them to methods of other components. We forward these events to an EJB session bean taking care of overdue books. For each student with an overdue book a notice is generated and the overdue flag is set within the database that forbids these students to borrow new books until the flag is cleared. The flag is cleared when the student returns all overdue books and pays the late return fee. In this case the accounting logic is used to enter the payment. After the money has been paid the flag is cleared. For this task the GUI responsible for returning books comes into play and emits another event that clears the user's overdue flag.

EJB components can be integrated easily with the CWB by specifying a naming service host name and a name that denotes the bean within the namespace of the naming service. COM+ components are integrated using the programmable name of such a component [15].

After the components have been selected, the user only has to setup the connections between them. When the user sets up the connection between the COM+ component and the EJBs the CWB generates an adapter that uses the EJB component wrapper to invoke the methods of the enterprise beans. Since there is no way to access an enterprise bean's reflection API without having access to the bytecode of the EJB's interfaces we have to install the compiled versions of these interfaces on the host where the adapters reside.

After we have defined the scenario of our library application we have exported it as Java source code that has been compiled and deployed on a machine where we had access to our COM+ components and onto our EJB components.
6. Future Work

We have implemented component wrappers for CORBA, COM+ and EJBs. JavaBeans, as desktop components are supported as well. Interestingly, it has been possible to implement a component wrapper around web services accessed using the Simple Object Access Protocol (SOAP) [3] in a similar way. It would be interesting if a component wrapper for Microsoft's .NET architecture were also possible. Since .NET remoting, a .NET API similar to Java RMI, supports SOAP as communication protocol this should be possible [20].

Our component wrappers, however, are not yet completed since we do not yet support the conversion of all datatypes used in method calls between component models. We support the conversion of almost all primitive data types such as number and string types, and the conversion of some complex data types, but we have no actual implementation for constructed types. We plan to map constructed types of the different component models to types of the implementation language. The user can provide the conversions within the user interface wizard responsible for adapter conversions.

Another problem concerning component type systems are references passed via method calls across component model boundaries. There has to be a transparent conversion of references of one component model to a proxy reference. This problem is rather complex when only two component models are involved, and grows when more models are involved.

Currently, the Component Workbench is able to compose components implemented for different component models. Almost all distributed component models, however, specify services such as naming services, or transaction services that can be used by components implemented using these component models. These services, however, are not yet abstracted by the CWB. Hence, it is not yet possible to let components implemented for different component models participate within the same transaction context.

Since most component models provide support for the two-phase commit protocol, it should be possible, to let the COM+ Transaction Processing System and the Java Transaction Service cooperate. This would enable COM+ components to take part in EJB transactions and vice-versa.

The set of components available to the CWB is defined in XML-files that are written by users. These files contain data such as the available component models, the name of the components, and if necessary which naming service has to be used to access a component. We plan to integrate different naming services such as the CORBA naming service, and directory services such as Microsoft's Active Directory Service into the CWB to support developers in selecting and configuring appropriate components.

In the actual release every interaction between components has some part that is executed using wrapper code that converts messages between the component models. This can be of advantage at development time because the developer can notice every interaction between components on his machine. Obviously, this leads to a loss of performance that is not tolerable at deployment time. Therefore we plan to integrate bridging technologies in cases when performance is important and no loss in functionality can be expected. The generators
that are used at deployment time have to be changed to support different bridging technologies.

The Component Workbench is a tool for creating scenarios of existing components of different component models. Our prototype has rather restricted support for creating full applications. We are evaluating if an integration of the CWB into a full-grown integrated development environment is desirable. Recently IBM initiated the Eclipse [6] project that provides an open source framework that supports plug-ins for various utilization. Since many vendors support this project it would be interesting to integrate CWB into Eclipse.

7. Related Work

PolySPIN [1] is an approach that tries to solve the interworking problem of components written in different programming languages. PolySPIN attacks the problem by modifying the implementation of the object methods. The modified methods consult a language arbiter at each invocation that converts the call to the call semantics of the target component's implementation language. Since different programming languages use different type systems PolySPIN uses a matcher responsible to match types from different languages. PolySPIN, however, does not address the problem of objects that could interoperate on a conceptual level but whose interfaces have significant differences. The CWB solves this problem using component adapters. Additionally, the CWB does not require the original components to be modified which might be impossible in case of distributed components.

Interworking specifications support the integration of middleware systems of different kinds [7]. Interworking between CORBA and COM is specified in [17]. Implementations of this specification make use of compiler tools that automatically create mappings of the different component models [7]. Up to now only a few CORBA implementations support this form of interworking.

For instance, many different vendors of either CORBA or EJB servers support interworking between CORBA and EJB. The main reasons for better interworking are the availability of a Java language mapping for CORBA and the support of the CORBA communication protocol (IIOP) through EJB application servers in addition to Java RMI. Originally COM has been available on Microsoft's Java Virtual Machine. Since Microsoft no longer supports Java for COM+ development, there are only some approaches to let COM+ objects be accessed from Java. One successful approach seems to be Intrinsyc's J-Integra, which is a Java-COM bridge that provides COM+ access to and from Java objects running on any operating system [14]. Microsoft has integrated COM interoperability into its .NET environment. The .NET architecture uses wrapper classes that mediate between .NET and COM, both as client and server [19].

Unfortunately, these bridging technologies support just the interworking between only one pair of component models, whereas the CWB supports the interworking between an arbitrary number of component models.
8. Conclusions

In this article we have presented the interworking problem between distributed components of different models. Based on this we have explained the Component Workbench, our solution to this problem.

While implementing the CWB, we had to solve the challenge to provide an abstract component model. This model had to be easy to understand and powerful enough to accommodate the different features provided by today's component models without restricting developers to use only the subset of features provided by all of the component models.

This challenge was solved using component wrappers. No matter of the component model a component belongs to, the component wrappers map it to our abstract component model, hence providing the same interface to developers. The interface of the wrappers is easy to understand because it provides an API that is similar to CORBA's Dynamic Invocation Interface (DII) [17] and the Java's Reflection API.

To evaluate our approach, we implemented and presented a small library application that demonstrates that the CWB can be used for the composition of applications from components implemented for different component models. Compared to today's bridging technologies, the advantage of our approach is that it allows the translation between arbitrary component models whereas bridging technologies can only be used to bridge between pairs of component models.

References


Symposium on the Foundations of Software Engineering (FSE-9), pp 311-312, September 2001


FORMA\textit{ware}: Framework of Reflective Components for Managing Architecture Adaptation

\textit{Rui S. Moreira}
Computing Department  
Lancaster University  
Lancaster LA1 4YR, UK  
moreirar@comp.lancs.ac.uk

\textit{Gordon S. Blair}
Computing Department  
Lancaster University  
Lancaster, LA1 4YR, UK  
email: gordon@comp.lancs.ac.uk

\textit{Eurico M. Carrapatoso}
Faculdade de Engenharia  
Universidade do Porto  
Rua Dr. Roberto Frias 4050-497 Porto  
email: emc@fe.up.pt

\textit{INESC Porto}
Rua Dr. Roberto Frias  
4200-465 Porto, Portugal  
email: emc@inescporto.pt

\section*{Abstract}
Software engineers use abstraction to better understand, model and reason about the surrounding world. Recently Architecture Description Languages (ADLs) introduced new levels of abstraction with potential use at run-time to support system evolution. In this paper we propose the \textit{FORMA\textit{ware}} architecture that blends run-time architectural representation with a reflective programming model to address adaptation issues and promote the proximity between design and development. Reflection opens up composition architecture through a replaceable default style manager that permits to execute architecture reconfigurations. This manager enforces the structural integrity of the architecture through a set of style rules that developers may change to meet other architectural strategies. Each reconfiguration runs in the scope of a transaction that we may commit or rollback.

\section{Introduction}
Current middle tier software solutions define component model platforms that hide the heterogeneity and complexity of distributed applications. Furthermore, these platforms standardize a set of services (e.g. naming, event, security, persistence, transaction) through which components can interact with the environment and find and cooperate with other components in different machines. Pervasive component standards address the issues of customization, independence and interoperability but hide the details of composition and behavior of applications. This transparency is useful if we want to produce static and highly stable applications but it is by no means suitable if we need to address contextual or environment fluctuations (e.g. GUI, network) and requirements that may change or evolve with time (e.g. extend or include new functionalities).
In this paper we address the goal of reconfiguration to cope with variance in systems like digital libraries and co-operative applications, where interaction, multimedia and mobility aspects pose changing requirements on the supporting architecture [1]. In particular, our approach is formalized and provided in a component framework named FORMAware. This framework proposes a Pro Forma way for development: a reflective object-oriented class ontology that guides the way developers configure and reconfigure distributed applications. The use of such a component framework leads the software development process in a perfunctory manner. Moreover, it prescribes a method to formally carry out architecture constrain verifications whenever architectural adaptation (e.g. add, plug, unplug, remove, replace components) is required, since the architecture structure is opened up and maintained by the reflective component model approach.

The structure of this paper proceeds with section 2 that introduces current middleware approaches and standardization efforts in this area. In section 3 we present the motivation and design challenges of our approach. Then section 4 covers the architecture of the FORMAware framework: it describes the reflective component model approach and focuses on the explicit representation of the architecture, associated style rules and architecture transaction service. Section 5 follows with a possible reconfiguration scenario that demonstrates the usage of the FORMAware framework. Related work is presented in section 6, before the concluding remarks.

2. Current Middleware Approaches

Software engineers borrowed the term component from the mechanical and electronic areas. This term means some well identified part of the system with a specific functionality that can be replaced without affecting the behavior and performance of the system. In computer science and particularly in middleware the term has a similar meaning since we recognize the benefits of producing and combining pre-fabricated software modules built by different manufactures. Generically, a component can be any binary and independently tested software unit that may be composed by third-part users into different software applications [18]. Other definitions distinguish these binary Lego blocks, which can be modified at design time, from subroutines or libraries, which must be modified at source code [11].

More specifically, in the object oriented field components are pieces of software that support inheritance, polymorphism and encapsulation. These objects are designed to facilitate reuse. Nevertheless, under the point of view of major vendors the definition is wider and components are usually synonym of proprietary objects such as libraries or binary objects (e.g. Java classes, OLE or ActiveX components) which do not necessarily support inheritance or polymorphism [6]. In summary, some definitions use narrower concepts, but the fundamental ideas behind components are still independence (encapsulation of implementation details), composition (through well defined service interfaces), reuse (market distribution and customization) and ultimately adaptation to overcome the development costs and leverage software quality.

Current use of software components is available through Enterprise Java Beans (EJB), Component Object Model (COM+) and .NET market component models, and also the
implementations of CORBA ORBs and the CORBA Component Model (CCM). These component models hide implementation details behind publicly exposed interfaces that establish rigorously each component interaction protocol. Such models also ensure the connection of disparate distributed systems, hiding the complexities of networks and operating systems behind standard APIs. These market standardization forces try to impose their views on components to enhance uniformity, reusability and productivity. Nevertheless, now that we achieved the goals of heterogeneity and interoperability we need to open this approach to tackle the problem of dynamic reconfiguration [2]. Open systems architectures should not only standardize high-level interfaces but also promote architectures specifications where components themselves are open. This will yield not only the benefits of interoperability (e.g. between different manufacture components) and portability (e.g. between different distributed platforms) but also the benefits of extension and evolution (e.g. through changes made on the configuration of components), without affecting the integrity of the architecture.

The issues of supporting the representation (cf. Architecture Description Languages) and analysis (e.g. architecture style rules) of component configurations were addressed in the last decade by the software architecture discipline [19]. Programs like the Domain-Specific Software Architecture (DSSA) and Software Technology for Adaptable Reliable Systems produced some key technical foundations for systematic reuse of processes, methods and tools in software architectures [17], [20]. The rules imposed on the composition of architectural elements (cf. components and connectors) are commonly referred to as architecture style. These rules constrain the way components may be bound and reconfigured (e.g. matching interfaces and component types, security and dependability requirements, etc.), hence providing structural constraints that we may use to evaluate the consistency of dynamic configurations, i.e. conditions on which the configuration can change [21]. Nevertheless run-time architecture representation and manipulation are still a difficult technical issue. Practically all ADLs support the representation of software architecture configurations and static analysis, but only a few support dynamic analysis and evolution and even so with some restrictions (e.g. C2 SADL [14], Darwin [7], Rapide [13]).

We are particularly interested in the high-level view of ADLs to describe the elements of an architecture (representation issues) and to use the structural constraints imposed by style to validate the dynamic evolution of configurations (process issues). This will permit us to impose restrictions on the reconfiguration actions and consequently guarantee the structural integrity of the adaptation process.

3. Motivation and Design Challenges

We recognize the advantages of object-oriented development and component-based software. Moreover, we believe that future distributed services, especially in the case of mobile, interactive and multimedia systems (e.g. digital libraries, learning systems, cooperative applications) will be more architectural and resource demanding. Consequently, next generation middleware architectures should support reconfiguration to adapt to short and long term demands. By short-term conditions we mean those we can predict during design and long-term requirements are related with evolutionary scenarios impossible to
forecast when we first create our services. For example, users mobility poses arduous platform and network constrains (e.g. changes in the interface requirements, processing capabilities, connection properties and protocols) that we sometime may predict. Large-scale systems also have hard scalability problems when peaks of requests force the system to reconfigure, to accomplish service continuity or graceful degradation. In real-time services the problems are even more demanding since we add time constrains to the resource management limitations (e.g. stream buffering and compression to adapt to bandwidth limitations). More unpredictable situations can arise when architects need to extend or evolve a system to support component refinements or include new services (e.g. support the interoperability between different digital libraries, include an account and billing systems in a previous free-access service, change the accesses service to a more secure protocol).

Given the forces above we believe that next generation middleware platforms should be more flexible and focus on the problem of adaptation. The key challenges that we propose to address in the design of our framework are:

- **Promote a modular component-based development**: use first-class entities to represent and manipulate basic and composite components; also allow explicit component wiring through provided and required interfaces;

- **Provide a principled approach to flexibility**: a uniform separation of concerns through principled mechanisms for introspection and adaptation of basic and composite components;

- **Provide explicit architecture awareness**: incorporate a high-level representation of the systems architecture (e.g. architectural view) in the programming model will permit us to represent and select a given architecture policy/strategy to manage the style rules and symbiotic relationships between components;

- **Guarantee safe architectural changes**: explicit representation of a modifiable set of style rules will imposed structural conditions over the topology of the architectural and guarantee architectural integrity;

- **Guarantee atomic architectural changes**: achieve consistent reconfiguration of composite architectures based on a transactional adaptation process with inherent control over the synchronization and state of the components involved the adaptation. Also provide the ability to coordinate and control (e.g. commit or rollback) the changes invoked on the composite component architecture.

To make effective use of component software development, engineers must be able to create, customize, combine and reconfigure components in an efficient and easy way. Our solution proposes a reusable and tailorable object-oriented hierarchy of classes that combines and implements a set of variant design patterns to capture architecture knowledge into the programming model. Moreover, this component framework opens up the content and architecture of its modules to allow the creation, combination and management of multiple reflective components according to a chosen architectural style. This style consists of a set of rules that constrain the way components may be bound and reconfigured.
4. The FORMAware Architecture

4.1. Promote Modular Component-Based Development

Middleware platforms are generally based on the notion of components which are only a part of an overall ontology (cf. Component Model) that encapsulates the platform semantic and a set of services that hide implementation details and promote the component-based development (CBD) [6]. The FORMAware component model promotes both components and interfaces (Provided and Required) to first-class objects. Components may be basic units of computation (cf. components) or communication (cf. connectors) and may be plugged together through their interfaces to form a composite component. A component will provide the services contractually advertised by its provided interfaces as long as it requires interfaces are supported by other components (see figure 1a). Furthermore, composite components do not have a flat composition graph. Instead their topological composition model captures the architecture knowledge which enforces the structural style (through a set of style rules) chosen by the developer (see figure 1b).

![Diagram of Reflective Component Model](image)

**Figure 1**: Reflective Component Model - (a) Components Structure; (b) Composition Pattern.

4.2. Provide a Principled Approach to Flexibility

In contrast with current component models that hide component behavior behind public interfaces, we propose a reflective component model that coherently provides structural self-awareness about the components architecture. **Reflection** is an architectural pattern that proposes a flexible and suitable design for dynamic adaptation [3]. The FORMAware design follows this pattern according to the principles described in [2] to automatically open up the components by means of introspection and adaptation. The separation between structural concerns and behavioral representation permits us not only to observe and control the component activities (e.g. threads) and their resource consumption (e.g. memory, processor) but also to maintain the architectural integrity over the reconfiguration actions performed on the graph of components [2]. This partitioning of the meta-space into two distinct viewpoints (cf. topological and strategic [5]) supports the reification of specific self-aware information with particular semantic meaning for each aspect that we want to control. The FORMAware framework provides the mechanisms to create introspectors and adapter meta-objects to both get and set the content and the structure of atomic and composite components (see figure 2).
4.3. Provide Explicit Architecture Awareness

In this paper we are particularly interested in the *reification* of the architecture and the rules governing the architecture style for the purpose of reconfiguration. The *structural meta-space* is divided in two *meta-models*. The interface meta-model opens the content of components in terms of the provided and required interfaces. The architecture meta-model then *reifies* the architectural graph and the style constraints [15].

Figure 2: Access to the Introspection and Adaptation Meta-objects.

The design of the framework combines the use of several patterns (e.g. proxy, façade, composite, strategy and template method [9]) to open up the architecture management (e.g. graph of components, symbioses between components). Developers may ask (get) and choose (set) which style strategy (cf. *StyleManagerI*) to use in each composite component (figure 3). The framework provides two ways to change and/or extend the architecture styles. Developers may define their own style policies (e.g. Layer, PipeFilter, Broker, Blackboard) and set them as the strategy for checking style rules (cf. black box framework approach [8]). A more white box approach permits users to change the behavior of the default style manager (cf. *DefaultArchitectureStyle*) through the re-definition of template
methods [9]. This pattern permits us to redefine which checks and primitive methods are called by the strategy operations. Hence it becomes possible to extend or modify any existent style manager (figure 3) by re-implementing some or all of the tests and primitive methods or even reordering them in the template methods (cf. strategy operations).

4.4. Guarantee Safe Architectural Changes

Each operation invoked on the architecture of a composite via its proxy adapter (cf. AdapterRArchitecture) is forwarded to the style manager by the architecture context (figure 3). Then the style manager checks the style rules constraints for that operation. A style rule is a class developed by the architects of a given style and is instantiated on the style manager. Each rule has a type (e.g. CheckCType, CheckIType, CheckSymbiotics, CheckCLocal, CheckPLimit, etc.) and applies to a set of architectural operation types (e.g. Add, Bind, Export, etc.). The rules provide a verifyValidity() method that checks the constraints declared for that rule (figure 3). For example, a rule may check if the type of the component matches the style in use. Another rule may check if the component has matching interfaces. The location of the component may also be checked for compliance with the roles (e.g. client, server). Component properties will also be checked by rules inspecting the limits imposed for that kind of components.

The style manager uses the type of operation (to which the rules apply to) and the type of the rule to select which rules to verify. We apply a guarded execution pattern that uses the getSelectedListStyleRule() method to handle the selection of the rules. This method looks up for the rules that should be checked in the specified operation type. The returned rules are then verified. If all the rules are met the style manager uses the architecture graph operations (cf. ArchitectureGraphI) to reflect back the changes into the architecture.

4.5. Guarantee Atomic Architectural Changes

Any architectural reconfiguration entails the dangers of inconsistency if no constraints are imposed on the operations invoked on the architecture. To guarantee the consistency of the final configuration we check the global set of structural rules before any set of architectural operations may be committed. Furthermore, we provide the mechanisms to rollback any operations that break the architectural constraints of the system. Thus, committing only the reconfiguration operations that keep the integrity of the architecture. We do this through a transactional reconfiguration process that permits us to initiate and manage atomic architectural operations on composite objects.

The transaction manager (cf. ATransactionManager) implements a service responsible for managing architecture transactions and uses an associated lock manager (cf. LockManager) to control the execution of the transaction requests (figure 4) [10]. A transaction is initiated by a client through the beginTransaction() method on the adapter proxy (cf. AdapterRArchitecture). The manager creates a transaction object (cf. ATransaction) and returns a resource manager (cf. ATransactionResource). The transaction manager also enlists each created transaction in the lock manager. For each enlisted transaction, the lock manager, creates a scheduler thread (cf. ATransactionScheduler). This thread is responsible for scheduling the actions to be performed for that specific transaction.
For each architectural operation invoked on the resource, it creates a request object (cf. ATransActionRequest) and passes it to the transaction. The requests will be scheduled in the lock manager according to a chosen policy (e.g. wait safe state, force safe state, skip safe state). Each scheduler implements a specific lock and clone policy on the components that are going to be changed by the architectural operations (e.g. lock all-or-nothing).

The lock manager is also responsible for initiating a dispatcher thread (cf. ATransActionDispatcher) that executes the requests enqueued by all the schedulers. Hence, the lock manager maintains a table of transactions which holds a queue of requests scheduled for each transaction. These requests will be dequeued and executed by the dispatcher according to a certain policy (e.g. Round-Robin).

The architectural operations are encapsulated by an action request (e.g. add, replace, remove, plug, unplug, etc.). Each scheduler is responsible for interpreting the requests invoked through the resource. Then it generates and enqueues the actions to be performed for that transaction. The scheduler may insert new actions into the transaction requests stream, to lock, clone, etc. For example, if we start a transaction to execute a replace request, the scheduler would enqueue the following requests to be dispatched in the scope of the transaction: lock, clone, unplug, add, plug, remove, unlock, commit. The generated requests depend on the scheduler policy.

5. Application Development

5.1. Overview

In this section we will introduce a basic digital library service that was prototyped following an object-oriented methodology [4]. This service works perfectly under predefined requirements but if the requirements change due to environment restrictions or...
client mobility the architecture is not open enough to adapt to those new circumstances. Suppose for instance that we need to improve the security protocol (to prevent unauthorized access to the digital library service) or that we need to provide new means to access the service (e.g. through a PDA over a wireless LAN). These new requirements will force us to re-design and re-implement the system with the associated time and cost drawbacks.

5.2. Traditional Approach

The requirements gathering and analysis phase assembles a description of how the service behaves according to the different end-users and how it inter-operates with the environment [4]. The service foresees 3 kinds of users that access the repository of distributed digital documents: end-users (search and retrieve information); producers (submit new items) and librarians (manage the library items). These users pose the following requirements: local network accesses; transparent distribution; restricted document upload and advanced search mechanism. Then the service analysis phase describes what the service must do, from the developer’s point of view. This corresponds, in Open Distributed Processing terms, to the high-level information viewpoint models [4]. In more detail, for each user a specific Session object is created with particular facilities that depend on the user permissions. The Session is responsible for executing the user requests (e.g. queries) and assembling the answer back to the user (figure 5).

The design phase defines the service interfaces and details the implementation behavior. We have used a distributed variant of the method factory design pattern and implemented it with commercial technologies (cf. Java AWT, JavaMail, Java Beans Activation Framework and OrbixWeb communication framework). The implementation produced a functional service but rather monolithic and with few adaptation capacities.

5.3. Approach Based on the FORMAware Framework

For this service we propose two Broker style composite components: one that implements the Factory functionality which creates the other composite, providing the Session services (figure 6a). Using an architecture style supported by the FORMAware framework would guarantee the automatic generation of basic components (from application classes) by means of the WrapperGenerator tool. Furthermore, it would allow us to assemble the composite components and then use the reflective facilities to change the architecture and support the new requirements.
The explicit representation of connectors will force us to develop the factory and session connections (incorporating the proxy and skeleton behavior) separately from the other components. This extra effort comes with plug-and-play benefits because the wiring will now be done outside the code of the components. We will now describe how to generate basic components, how to assemble them to form composite components and, finally, how to reconfigure these components to change or evolve the architecture of the service.

**Generation of Basic Components and Connectors**

The classes that represent the basic components, basic connectors and interfaces (cf. provided and required) can now be generated from the session, factory and connection classes, by using the wrapper generator tool. For example, the classes in bold (figure 6b) represent the digital library session component and associated provided interface that are automatically generated. Similar component classes will be generated for the factory class and for each role (cf. client proxy/stub and server proxy/skeleton) of the broker connectors (session and factory connectors).

**Architecture Composition - Assembly of Basic Components**

Given the basic component classes and associated interfaces (generated previously), we can now build each of the composites of the service. The architecture of composite components is assembled using a Java class (e.g. DigLibCompositeComponent) that for each role of the composite creates the instances of the components and connectors, and then the plugs between them. For example, the "script-like" code to deploy the server role of the factory composite component will use the adapter object of the composite architecture to firstly choose the architecture manager (cf. Broker):

```java
AdapterRArchitecture adapter = this.createAdapter();
adapter.setArchitectureStyle(new Broker());
```

Then, we need to create the instances of the components and connectors:

```java
factoryComp = new DigLibSessionFactoryBasicComponent(host, port);
factoryConn = new FactoryCORBABasicConnector(RCompositeRole.SERVER);
```

We may now add these components to the architecture graph by calling the `addRComponent()` method available in the adapter, and passing to it the component references and respective directives (e.g. host and role). This method will forward the call to the Broker style manager that will make the style verifications:
adapter.addRComponent(factoryComp, directives);
adapter.addRComponent(factoryConn, directives).

Before we can plug the factory component to the factory connector, we need to create the respective introspector objects. With these introspectors we may get the references to the provided and required interfaces that we need to plug:

IntrospectorRComponent introsFactoryComp = factoryComp.createIntrospector();
IntrospectorRComponent introsFactoryConn = factoryConn.createIntrospector();
ProvidedInterface piFactory = introsFactoryComp.getProvidedInterface("FactoryI");
RequiredInterface riFactory = introsFactoryConn.getRequiredInterface("FactoryI");
adapter.plugComponents(factoryComp, piFactory, factoryConn, riFactory).

Or simply indicate the names of the components, the respective interfaces names and the directives to plug the components:

adapter.plugComponentInterfaces("Factory","FactoryI","FactoryCORBA","FactoryI",directives);

The composite component described above will have similar code instructions to create and deploy the client side architectural elements, required when a user asks the service (please refer to [15] for a more detailed design of the deployment mechanisms). The final step will be to create a class that will run the service and publish it in the naming service (cf. adapter pattern [9]). The adapter will then forward the calls to the DigLibCompositeComponent whenever clients ask for a connection to the service.

**Architecture Reconfiguration - Adaptation**

Consider now one of the scenarios described earlier to include a security mechanism before passing the login request to the digital library. First, given the classes implementing the encryption and decryption algorithms, we would need to generate the respective component and interface classes. We could then deploy these new components in the architecture by using the reflection and transaction facilities provided by the framework:

ATransactionResourceRI resource = adapter.initTransaction();
resource.addRComponent(new DecryptBasicComponent(...), directives);
resource.unexportRInterface("Factory","FactoryI",directives);
resource.plugRInterfaces("Factory","FactoryI","Encrypt","FactoryI",directives);
resource.exportRInterface("Encrypt","FactoryI",directives);
resource.commit().

Similar operations would be used to deploy the components on the client side. This generic architectural operations (e.g. add, remove, replace, plug, unplug, export, import, expose, unexpose, etc.) may be used to cope with requirements forcing architecture reconfiguration.

### 6. Related Work

Since early publications on software architecture in the beginning of the 90s, this discipline grew in adepts, complexity and contributions. Preliminary publications made comparisons with traditional engineering disciplines, such as civil and chemical engineering, where architecture concepts, styles and tools also play important roles [19]. These movements, toward a software architecture discipline, covered mainly the formal representations of
architectures by ADLs [14], the application of domain specific architectures [20], the synthesis and runtime adaptation of software architectures [1], [12], [16], [21]. We are mainly interested in these dynamic reconfiguration issues.

The Arch Studio tool suite supports run-time reconfiguration at architecture-level [16]. It facilitates the reasoning about the consequences of system reconfiguration while preserving system integrity. This framework provides an event-based environment for monitoring and evaluating observations (e.g. performance, constraints). Moreover it permits us to perform evolution in consistency with an architecture model. The approach followed in [12] proposes a middleware architecture that allows detecting QoS fluctuations in the environment to trigger control signals to the application. The core component is a Configurator that uses fuzzy control to determine the optimal adaptation control actions. The ADL proposed in the ASTER project supports architecture descriptions embedded with non-functional properties (formally expressed in linear temporal logic) of the architectural elements, given at design time [21]. In an engineering perspective, they use the temporal logic descriptions to analyze new possible configurations.

The FORMAware framework also pursues style consistency to maintain architecture integrity, but our perspective is closer to the developer since it incorporates the architectural knowledge in the programming model and exposes it (via introspection and adaptation meta-objects) through a principled and generic way (cf. reflection). Moreover, our approach permits us to use, extend and change the architecture ontology via its set of style rules, hence it is not limited to a particular style. Finally, the proposed solution processes the adaptation in a transactional way, providing mechanisms to commit or rollback the reconfiguration actions.

7. Conclusion

The management and evaluation of architecture composition are key issues that are not addressed by current component models. We believe that component-based development supported by reflective constructs that expose architecture composition can be combined in a component framework to tackle runtime architecture reconfiguration and to bring us closer to the goal of software systems adaptability. In this paper we proposed a component framework solution, called FORMAware, that incorporates component-based development and architecture knowledge. Furthermore, this framework provides flexible mechanisms to recompose components at runtime to address scalability, mobility and general architecture evolutionary scenarios. The functionalities described in this paper have been designed and implemented in Java and we are refining the set of Exceptions to be thrown during the architectural transactions, whenever a style rule transgression occurs. Future developments will focus on the implementation of the Layer style since it promotes separation of concerns, hence well suited for assembling and managing adaptable systems.

Acknowledgements

Mr. Rui Silva Moreira is being supported by Fundação para a Ciência e a Tecnologia (FCT), grant SFRH/BD/4590/2001, Portugal. Mr. Rui Silva Moreira was partially funded, until September 2001, by the Universidade Fernando Pessoa (UFP), Porto, Portugal.
work presented in the paper was also sponsored by the Instituto de Cooperação Científica e Tecnológica Internacional (ICCTI) under a grant for the FAMA project between INESC Porto and INRIA Rocquencourt.

References


[12] Li, B., Nahrstedt, K., Dynamic Reconfiguration for Complex Multimedia Applications, white paper, University of Illinois at Urbana-Champaign, 1999.


Type Based Adaptation
An Adaptation Approach for Dynamic Distributed Systems‡,§

Thomas Gschwind
Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8/E184-1
A-1040 Wien, Austria
email: tom@infosys.tuwien.ac.at

Abstract
Recently, component models have received much attention from the Software Engineering research community. The goal of each of these models is to increase reuse and to simplify the implementation and composition of new software. While all these models focus on the specification and packaging of components, however, they provide almost no support for their adaptation and composition. This work still has to be done programmatically. In this paper we present Type Based Adaptation, a novel adaptation technique that uses the type information available about a component. We also describe the design and implementation of our reference implementation thereby verifying the feasibility of this approach.

1. Introduction
Today's component models can be distinguished between server side component models and local component models. Local component models describe pieces of software similar to libraries that can be used by builder tools to create an application. Server side component models, however, describe components similar to services that can be accessed by other components. In this article, we will focus on server side component models.

Most component models such as the CORBA Component Model (CCM) [20],[21],[22], the JavaBeans component model [13], the Enterprise JavaBeans (EJB) component model [5],[19], COM [6], or .NET rely on black box components with well-defined and publicly available interfaces. Based on the knowledge of these interfaces, the components can be composed to interact with each other. One component, for instance, might request a weather service from a naming or trading service. If the interface provided by the weather service matches the interface expected by the requesting component interaction between them is possible. Otherwise, there is currently no means for the two components to interact with each other. We propose to address this problem using type-based adaptation. In particular, we are interested in Dynamic Distributed Systems [12] where the communication

‡ This work was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories and the European Union as part of the EASYCOMP project (IST-1999-14151).
§ This work was in part done during an internship with Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA-94304.
patterns have not been defined \textit{a priori} and thus the interfaces provided by a service are not known to the client until run-time.

Type based adaptation builds on top of the interface a component \textit{expects} and the interface \textit{provided} by a component, information provided by modern component models. In case of server side component models such as the CORBA component model (CCM) or the Enterprise JavaBeans component model (EJB) the interfaces provided by a component is the only type information available. We will show, how to extract information about the expected interfaces.

Type-based adaptation does not rely on any additional description of the semantics of the component as could be provided by a semantic description framework such as the DARPA Agent Markup Language (DAML) \cite{4}. Semantic description frameworks are not yet widely available and rely heavily on standardization.

The outline of this paper is as follows. In Section 2 we present a running example of a sample application that can use type-based adaptation. Section 3 shows the model underlying our adaptation approach. Based on this, we discuss design issues in Section 4 and describe an implementation in Section 5 along with its evaluation in Section 7. Possible extensions and future work are considered in Section 8. In Section 9 we present related work and we draw our conclusions in Section 10.

2. Motivation

In a dynamic distributed system the communication patterns between its components are not determined \textit{a priori}. This allows the system to adapt more easily to the users' needs. For the implementation of such a system a middleware layer such as provided by CORBA or Enterprise JavaBeans (EJBs) is used. While both CORBA and EJBs provide server side components they lack the possibility to adapt the components available. Thus, component $a$ can only interact with component $b$ if $b$ provides the interface expected by $a$. As shown by the following example, the flexibility of a component model can be improved by the provision of a transparent adaptation mechanism such as type based adaptation.

![Figure 1: Internet Shopping Application](image)

The use case \cite{7} in Figure 1 shows an Internet Superstore where the User browses and selects products available. After the User has selected the products he wishes to buy, he
selects the shipping address from an Addressbook Service with which the User manages his addresses. When the User wishes to pay for the products, the payment is performed via a Payment Component that charges the User's account.

Almost all of today's Internet stores have an internal addressbook service. This, however, is inconvenient because the User has to manage a different addressbook for each store he wishes to buy goods from. He would prefer to use an external addressbook service to manage his addresses and simply instruct the Internet Superstore to use the addressbook provided by that service. As long as the Addressbook Service exactly implements the interface expected by the Internet Superstore this is possible today. If not, the components are unable to interact and the benefits of the above approach is lost. With multiple service providers, however, it is unlikely that they all will implement the same interface. This is where type based adaptation comes in. Type based adaptation allows for the transparent and automatic adaptation of the interface of one service (e.g., the Addressbook service) to match the interface required by another (e.g., the Internet Superstore).

3. Adaptation Model

Server-side components consist of multiple interfaces, each represented by a different object [20]. For a component using another component only the interfaces provided by the other component are important. Since an interface defines a contract that specifies the behavior [17], type-based adaptation uses this type information as the basis of whether a component can be substituted with another. The mapping information that define how one interface has to be mapped into another, however, has to be provided by a human capable of understanding the different interfaces that need to be mapped. A human has to decide whether two interfaces can be translated into each other, and, if so, to specify their translation.

Only the presence of this information makes automated adaptation possible. To provide mapping information type based adaptation uses adapters. These adapters algorithmically describe how to translate different interfaces into each other. The interfaces provided and expected by the components are the only type information required. The adapters themselves are written in typical programming languages such as C++ or Java. Finally, the adapters are stored in a repository with some meta-information about the adapters such as the interfaces they translate.

Our approach can be seen as an extension of the adapter pattern [9],[18]. The difference, however, is that the adapters are first-class objects described on their own and that an adapter repository exists having full knowledge about the adapters available and the transformations they describe. The repository stores information such as the interfaces the adapters translate, and optionally their performance characteristics or whether their translation is lossless. Based on this information the adaptation-process can be automated.

In case of server side component models such as the CORBA component model (CCM) or the Enterprise JavaBeans component model (EJB) the type of a component is represented by the interfaces it implements. Thus, the implementation of an adapter A that translates from an interface \( I_{\text{from}} \) to an interface \( I_{\text{to}} \) is straightforward.
In fact, code as it needs to be written for these adapters exists already in many of today's software systems in the form of wrappers or in the form of subclasses, the basic form of wrapping. Unfortunately, in such code especially when subclassing is being used, the adapter is part of a bigger component and thus cannot exist on its own. To be used in combination with the adapter repository this code has to be factored out into a separate class, the adapter. Afterwards, it can be used in combination with the adapter repository and thus can be reused automatically in other systems where similar interface transformations are necessary.

As shown by the sample adapter-repository in Figure 2, an adapter repository forms a directed graph \( G \) where the interfaces are represented by the vertices \( (V(G) = \{a, b, c, d, e, f\}) \) and the adapters by the edges \( (E(G) = \{ad, da, ae, ea, bc, fc\}) \) between the interfaces they can translate. In case an adapter required is missing adapters can be combined. For instance \( da \) and \( ae \) can be combined to simulate an adapter \( de \). To find a suitable combination of adapters a simple shortest path algorithm is sufficient. Additionally, the algorithm can be tuned to prefer adapters of specific characteristics by applying different weights to each edge (adapter). For instance, if the user were interested in a lossless adaptation, the weight of lossy adapters should be set to \( \infty \).

We propose to implement this algorithm in a separate adaptation component that can be used by a client or naming service. Whenever it is necessary to perform an adaptation the adaptation component can be queried for an adapter or a combination thereof to perform the required translation. This gives the user the impression of having automatic composition at hand. Only if no such adapter exists, the composition of the components cannot be performed automatically and the user has to provide a new adapter for the composition to succeed. After the adapter has been provided, it can be added to the adapter repository and made available to other users of the adaptation component.

A different approach to the adaptation problem is the use of a semantic description framework that allows to describe the interface, its methods and data structures using a common ontology. Such an approach might be implemented using DAML [4]. We claim, however, that this approach only adds another level of indirection since no commonly accepted ontology for the semantic description exists. Different ontologies, however, will be unavoidable since companies are unwilling to release any information about their
products in their early development states. Such information, however, is crucial for the standardization of a common ontology.

4. Design Issues

In a dynamic distributed system, each component implements a different service and can be executed by a different host as shown in Figure 3. Typically, when one component needs a service from another it queries a name server or trader for the service using a well known identifier. The naming service returns a reference which the client casts to a specific interface as shown in Figure 4. Now, the client can interact with the service returned by the naming service. If the service requested by the client, however, does not match the interface expected by the client an exception will be thrown and the client will be unable to interact with the service.

![Figure 3: Accessing a Service in a Distributed Component System](image)

```
try {
    Context ctx=getInitialContext();
    Object dobj=ctx.lookup("CookieServer");
    CookieHome ch=(CookieHome)
        PortableRemoteObject.narrow(dobj,CookieHome.class);
    Cookie c=ch.create();
    System.out.println(c.getCookie());
} catch(Exception e) {
    e.printStackTrace();
}
```

![Figure 4: Typical Client Code in a Distributed Component System](image)

4.1. Adaptation Component

Instead of throwing an exception it is better to use an adaptation component responsible for the transparent instantiation of the adapters when necessary. An ideal location for this is the client since typically the naming service or the service provider do not know the interface expected by the client. Still, it might make sense to add adaptation support to a trading service if the query language allows clients to request a component by its interface.

4.2. Repository Location

Since an adaptation component has to be available to the client and probably to the naming service, a straightforward approach would be to locate an adapter repository at both sites. While placing an adaptation component at each site is fine we recommend to use a central adapter repository or better to link the various adapter repositories. Hence, the adapters have to be pieces of mobile code [8] loaded on demand by the adaptation component.
While this approach increases the number of adapters available and the number of interfaces that can be translated into each other it has to be used in combination with mobile code. The security risks posed by mobile code can be solved by the following two approaches. First, the adapters should be executed within a safe sandbox environment. The Java Virtual Machine [16], for instance, can provide such an environment that does not allow the downloaded code to execute arbitrary instructions. Additionally, much work exists in the context of Java and Mobile Agents that can be reused for our approach [1],[10],[11].

Additionally, we have to determine where the adapters should be executed. We recommend the adapters to be executed by the party that wants the two components to interact with each other, typically, the client. This pushes the security risk and the adapter's resource consumption to the party benefiting from the composition. Thus, the service providers do not suffer any disadvantage.

5. Implementation

To evaluate the feasibility of our approach, we have implemented type based adaptation for the Enterprise JavaBeans (EJB) component model [5],[19]. Most of the implementation decisions, however, can be applied to other component models as well. EJB is a server side component model that provides support for security, persistence, and transaction management allowing the developer to focus on the application logic [5].

A typical Enterprise JavaBean consists of a home interface serving as the component's factory, a remote interface specifying the component's functionality and the component's implementation. After an EJB has been deployed in an EJB server, the EJB server provides a container for each component that takes care of the component's lifecycle and the interaction with its clients. For a client, however, an EJB component looks similar to an RMI service. Figure 4 shows the typical client code to obtain a reference to the home interface, to create a new instance of an EJB component and to invoke one of its method.

Since our reference implementation is based on Java which simulates a homogenous environment little additional support has to be provided to download the adapters on demand from a central repository. As discussed in the previous sections the following items must be considered for an implementation of type based adaptation:

**Adapter Repository:** This component is responsible to store the adapters and to provide lookup operations based on their meta-information.

**Adaptation Facility:** This component is responsible to compute an optimal chain of adapters.

**Adapter Description:** Each adapter comes with a description of the interfaces it can translate.

**Packaging:** Each adapter is stored in its own package along with the adapters description.
6. The Adapter Repository

The adapter repository itself is straightforward since it only has to store the adapters and their descriptions. Any data structure that can store a graph with parallel edges (multiple edges connecting the same two nodes) is sufficient. Since the graph, however, will consist of a large number of blocks (independent components within the graph) of equivalent interfaces, we have chosen to use an adjacency list that stores all the out-edges for a given vertex \( v \). For the lookup of adapters we have chosen to use Dijkstra's shortest path algorithm which has a complexity of \( O(|E(G)|) \) which typically is much smaller than \( O(|V(G)|^2) \) [3].

6.1. The Adaptation Facility

The code in Figure 4 illustrates that the narrow operation (a system independent cast operation) is the perfect place for the adaptation component. At this point of execution the interface provided by the server is available as part of the object reference and the interface expected by the client is passed as an argument to the narrow operation. Additionally, it allows us to transparently plug type based adaptation into an existing system by upgrading the middleware layer only.

The adaptation component only has to provide lookup operations as shown in Table 1 to be used by the middleware layer's narrow operation. While the first method returns an already instantiated adapter, the second method returns an object describing a combination of adapters. This object provides methods that wrap a service with the according adapter.

Frequently, a trading service also allows the client to lookup a service based on some properties. For instance, a travel agent might request any component that implements the `at.ac.tuwien.Weather` interface to display weather information on its web site. If the naming service supports type based adaptation, it can even return a different component as long as it can be translated into `at.ac.tuwien.Weather`. This is especially of interest if no component implementing the interface is registered at the naming service.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object getAdapter(Object from, Class to)</code></td>
<td>Instantiates an adapter that provides the interface to to the client and interacts with the service represented by from. The object returned is the adapted object.</td>
</tr>
<tr>
<td><code>Adapter getAdapter(Class from, Class to)</code></td>
<td>This method looks up an adapter or combination of adapters that provide the interface to to its clients and interacts with a service providing interface from. The object returned is a factory that can be used to create instances of the adapter.</td>
</tr>
</tbody>
</table>

Table 1: Methods Provided by the Adaptation Component

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Object lookup(Class to)</code></td>
<td>Lookup a service that provides the interface to to its clients or can be translated to that interface.</td>
</tr>
</tbody>
</table>

Table 2: Methods provided by the Naming Service

138
Since the JBoss EJB Server [15] we used for the reference implementation does not provide such a naming service we have implemented our own supporting the lookup of a component based on the interface it provides (Table 2). If multiple servers with the same interface are registered they are returned in a round robin manner.

6.2. Adapter Description

The description of an adapter has to specify the interface the adapter translates from and the interface the adapter maps to. Additionally, it should be possible to supply additional information about the adapter such as whether it performs a lossless translation, the adapter's performance complexity or other properties.

This description could be maintained by the adapter class itself and could be be accessed using introspection as it is used by the JavaBeans component model [13]. Alternatively, it could be stored externally using a configuration file.

```xml
<?xml version="1.0"?>
<adapter name="SampleAdapter">
  <mapsfrom>
    <interface>com.yahoo.AddressDatabase</interface>
  </mapsfrom>
  <mapsto>
    <interface>com.amazon.AddressBook</interface>
  </mapsto>
  <implementation type="classname">
    at.ac.tuwien.infosys.tba.SampleAdapter
  </implementation>
  <lossless>false</lossless>
</adapter>
```

Figure 5: Sample Adapter

We have decided to support both choices. This allows us to keep the adapters simple as well as to extend type based adaptation to other application domains where the use of introspection might not be possible. For the configuration file we have chosen to use XML [2],[14]. Using XML has the advantage that existing tools can be used to parse the adapter specification. Only a document type definition or XML Schema has to be provided that describes the syntax of the adapter's specification.

A sample adapter description is shown in Figure 5. The SampleAdapter converts from a fictitious com.yahoo.AddressDatabase interface to a fictitious com.amazon.AddressBook interface. Additionally, the specification indicates that the adapter's implementation is provided by the at.ac.tuwien.infosys.tba.SampleAdapter adapter and that the transformation is not lossy. Thus, when the adapter is applied some information about an address might be lost.

6.3. Packaging

To simplify the adapter's installation and transfer to another site we have decided to put all the class and resource files required for the adapter into a single archive. In general, we
recommend to use a format similar to the format already exploited by one of the existing component models. Since our implementation is based on Java, we have chosen to use a .jar-archive for the reference implementation.

7. Evaluation

We implemented a weather service and an addressbook service that we used for experimentations with our system. Both systems were implemented using Enterprise JavaBeans [5]. For the weather service we implemented a set of different weather services providing access to weather information along with a set of adapters able to convert between them. Finally, we implemented a client to see whether our implementation was able to provide for a transparent adaptation of the different weather services.

Our system was able to adapt the different weather components transparently. In some cases, however, the adapters were unable to provide some information expected by the client. This is due to the fact that the adapter cannot generate information that is not provided by the component it adapts. This problem, however, could be solved by allowing the adapter to contact several different services to collect the required information.

For the addressbook service we downloaded and installed the petstore and ebank web applications from Sun's website. Both applications were extended to allow the customer to specify our external addressbook service instead of having to type in the shipping and mailing addresses over and over again. This extension allows the user to specify a location of an external addressbook service to be used. After the user has specified the external addressbook, the web application contacts the user's addressbook service and presents a list of the mailing and shipping addresses to the user.

In this scenario, however, the petstore application acts as a client of the addressbook service and thus executes the adapters. While the adaptation was performed as expected, for security reasons, however, we think that the adapter should be executed by the web browser. Otherwise, the web browser's user might be able to inject malicious code on the petstore server provided it has access to the adapter repository. Additionally, executing the adapter within the web browser would allow the user to control what address data is transferred to the web application and thus increases the user's privacy. This issue, however, will be attacked in future versions of our implementation.

8. Future Work

While we have only presented type-based adaptation in the context of dynamic distributed systems, it can be extended to other application domains as well. For instance, it could be used in combination with more traditional software development tools such as Integrated Development Environments (IDEs). In a typical IDE components are composed by selecting an event a component can generate and specifying the method or connector to be executed. Usually, this connector has to be implemented by the Developer. Otherwise, the interaction between the components would be limited to the execution of existing methods matching the event's signature [13].
Type based adaptation eases this problem by allowing the user to reuse connectors that have been written previously. It allows the IDE to understand the purpose of the connectors and enables the IDE to present the Developer with a set of connectors that can be reused for each newly created connection. For instance, a connector toggling a certain property could be reused fairly frequently.

It might also be possible to use type based adaptation to integrate a component in different component environments. To identify interfaces across different components, however, a uniform type identifier consisting of the component model as well as the interface would have to be introduced. In this case, the adapter would not only provide the code for translating between the interfaces but also the bridge-code necessary to translate one protocol into the other. Before we will be able to attack this problem, however, it is necessary to gain more experience with type based adaptation.

One issue we have not resolved yet is the performance degradation if multiple adapters need to be combined. We assume, however, that in a typical situation most of the computation takes place within the components. Additionally, we expect that typically no more than two or three adapters will have to be combined. Thus, the intercommunication between the components is less important to be optimized.

Even though performance is less important it should not be ignored entirely. If several adapters need to be combined partial evaluation could be used to fuse the adapters together. This inlines a large number of method calls and thus lowers the performance penalty if a large number of adapters are combined. Additionally, constants passed from one adapter to another can result in the elimination of a considerable number of redundant computations.

9. Related Work

The interworking problem between different components has already been identified by the NIMBLE [24] language which is part of the Polylith [23] system. Unlike conciliation, NIMBLE does not take the object-oriented view into account and solely operates on a procedural level. Compared to type-based adaptation, however, their adaptation is static and their adaptations cannot be chained. In dynamic distributed systems, however, it is important that the adaptation is performed at run-time since the components that need to interoperate with each other cannot be known a-priori.

Another approach using adapters is the composite adapter design pattern presented in [25]. While type based adaptation focuses on the adaptation of the intercommunication between server side components, this pattern focuses more on the software engineering side by trying to enable the independent development of an application and the framework model the application uses.

The idea of the composite adapter is to implement all wrapper classes adapting the application classes to the framework as inner class of a composite adapter class. While the composite adapter class takes care of the instantiation of the wrapper classes, the inner classes only implement the adaptation code. To simplify the implementation of the adapters, a new scoping (adapter) construct is proposed.
Another interesting adaptation approach is used by Jini. Jini uses proxies responsible to interact with a given device. Whenever, a client wants to interact with a device, it downloads the device's Java proxy and interacts with it. The proxy is responsible to shield the application from having to deal with the device's wire protocol. This allows Jini to access devices from different component environments as long as an according proxy exists. An architectural overview of Jini can be found in [26].

10. Conclusions

In this article, we have presented type based adaptation a simple but flexible and powerful adaptation technique. Type based adaptation supports the automated adaptation of components. Since the adaptation can be performed during run-time, as we have shown, it is the ideal adaptation approach for server-side component models.

Type based adaptation uses an adapter repository that stores prebuilt adapters. These adapters specify how one interface can be translated into another. Hence, the repository contains all the information necessary for an automatic adaptation process. To some extent the repository can be compared to semantic description frameworks [4] but using an algorithmic approach to specify the interface's translation instead. One advantage is that only the adapter repository needs to be standardized whereas for semantic description languages the terminology to describe the semantic of each application domain needs to be standardized. Another advantage is that multiple adapters can be combined to build a more powerful one. Hence, it is not necessary to provide adapters for all the different combinations.

We have demonstrated the feasibility of type based adaptation in combination with dynamic distributed component systems. Our results, however, indicate that it can be used in a variety of other application domains as well. We have implemented a prototype for dynamic distributed component systems that operates on a per-interface level and enables automatic adaptation. As we have shown with our examples automatic adaptation is important for dynamic distributed systems since it allows services as already available on the Internet to give more flexibility to their customers by allowing them to decide what services should interact with each other.

Acknowledgements

I would like to thank Mehdi Jazayeri for his continuous support and Pankaj Garg, my supervisor during my internship with HP labs, for his help and support of the project.

References


Fred Buckley and Frank Harary. Distance in Graphs. Addison-Wesley, 1989.


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1st edition, January 1995.


Session IV – Technologies
An Evaluation of the Use of
Enterprise Java Beans 2.0 Local Interfaces

Hans Albrecht Schmid
University of Applied Sciences, Konstanz
Brauneggerstr.55
D 78462 Konstanz, Germany
email: schmidha@fh-konstanz.de

Abstract

The collocated invocation overhead of Enterprise JavaBeans (EJB) 1.1 remote interfaces is often a severe problem, in particular with distributed application architectures like J2EE where collocated calls, like those among session and entity beans are frequent.

EJB 2.0 introduces local interfaces as a solution to the collocation overhead and related problems, based on the implicit assumption that EJB components can be clustered. This paper evaluates the use and usefulness of local interfaces: it analyzes which role EJB components from typical state-of-the-art applications play in clusters. It is shown that, under frequent conditions, components intended to be cluster-internal must be made cluster facades, and possibly have both remote and local interfaces, or the cluster size increases dramatically.

The consequence is that an improved application performance, for which local interfaces have been introduced, can often be attained only at the cost of an over-proportionally increased system complexity and programming effort.

1. Introduction

According to the Enterprise JavaBeans (EJB) 1.1 specification [EJB99], an EJB component has a remote interface that represents the business aspects (for clarity we call it a business interface), and a home interface that represents the life-cycle aspects. Both kinds of interfaces are Java RMI (Remote Method Invocation), or RMI over IIOP (for IIOP see [CACM98]) remote interfaces. The consequence is that collocated invocations are RMI remote invocations. A collocated invocation is made from the same network node and Java virtual machine as the invoked component.

A collocated RMI invocation causes costs that may go up to the same order of magnitude as those of a remote invocation, as e.g. 0.8 milliseconds (abbreviated: ms) versus 1.3 ms measured with Windows NT Java 2 JDK 1.3 on a Pentium III 500 Mhz and 100MBit local network (compare also [OH99]). This means that a collocated invocation is up to 10 000 times more expensive than a local one.

The collocated invocation overhead may often cause severe performance problems and reduce the power of an application server considerably. This is a particular problem with a distributed application architecture like J2EE that allocates the business components as collocated EJB entity beans or session beans on a central server [CCL00], and in a
transactional environment like an EJB environment in which all resource consumption is optimized to avoid any unnecessary overhead.

The Enterprise JavaBeans 2.0 specification [EJB01] introduces local interfaces in addition to remote interfaces as a solution to this problem. An EJB 2.0 Enterprise JavaBean may have a remote and/or a local interface and a matching remote and/or a local home interface.

The use of local interfaces has disadvantages which are known, like the loss of location transparency. We analyze a different aspect that is not addressed in the EJB 2.0 specifications and elsewhere, namely that the introduction of local interfaces is based on the assumption that EJB components can be clustered. From this analysis, we get a better and deeper understanding of the characteristics of local and remote interfaces, as described in section 2.

In section 3, we evaluate under which assumptions and restrictions local interfaces can be used in applications. In particular, we investigate under which conditions their use results in performance improvements without introducing additional complexity, design effort and programming effort. An astonishing result is that, under conditions met by practical applications, this is frequently not the case.

2. Clusters of EJBs with Local Interfaces

We will use electronic-shopping as a running example. A remote shopping-servlet as an electronic-commerce client has collected the products which a customer wants to order, and creates an order. To reduce remote invocations, we introduce, in addition to the entity beans Product, Order and LineItem, a session bean CreateOrder. All beans like the CreateOrder session bean and the entity beans are collocated.

When the customer wants to order the collected products, the shopping-servlet calls CreateOrder and passes to it the products. Suppose the shopping-servlet invokes 5 operations of EJB session and entity beans. CreateOrder first creates an order, then iterates over the collected products to get each item and the quantity to be ordered. It gets the product number, name and price for each item, and creates a line item, which it adds to the order. After the grand total has been calculated, CreateOrder returns the order to the shopping servlet. Suppose the collocated session and entity beans invoke 100 operations among themselves.

2.1. Facades and Cluster-Internal EJBs

A set of EJB components may be partitioned in clusters if the components of a cluster are collaborating closely, and if there is little collaboration among the components of different clusters. All EJBs of a cluster are allocated to the same network node, Java virtual machine and usually to the same container so that the close collaboration is local and fast. Therefore, local interfaces may be used for a fast operation invocation among the EJBs of a cluster.

The session bean CreateOrder and the Order, LineItem, and Product entity beans form a cluster since the collaboration among them is very close. The collaboration with objects or EJBs external to the cluster is very loose.

In a cluster of EJBs, we distinguish two kinds of EJBs (compare : [GHJV95]):
A facade EJB is used by a cluster-external client, like a servlet or an applet. (An object or component A is said to use an object or component B if A has a reference to B and invokes its operations via this reference.) For example, CreateOrder is a facade, since it is used by the shopping servlet (see Figure 1).

A cluster-internal EJB is used only by other EJBs from the same cluster. For example, the Order, LineItem, and Product entity beans are cluster-internal EJBs since they are used only by CreateOrder or themselves (see Figure 1).

2.2. Remote and Local Interface

A remote interface, like that of CreateOrder (see Figure 1; remote interface designated by a box with inscribed "R"; local interface designated by a box with inscribed "L", home interface not shown) may expose a serializable type or a remote interface type, but must not expose a non-serializable local type.

That means the CreateOrder remote interface must not expose the Order interface even though it keeps a local reference to it in an attribute; it might expose a remote interface type of some other EJBs (as indicated by the dashed line in Figure 1) if required.

On the other hand, a local interface may expose a local type or a remote interface type. Typically, a local interface of a cluster-internal EJB, like Order, exposes only local interface types of other cluster-internal EJBs, like that of LineItem. But if required, a local interface of an EJB like Order might expose also a remote interface type of another EJB in the cluster, like that of CreateOrder (as indicated by the dashed line in Figure 1), or outside of the cluster (see Product in Figure 1).

2.3. Business Interfaces

A cluster-internal EJB has only local interfaces, since it is used only by EJBs or objects from the same cluster. It has a local (business) interface (see Order, LineItem and Product
EJB in Figure 1) and a local home interface. A local interface and local home interface may be used only by a collocated client from the same cluster.

A facade EJB has remote interfaces, both a remote business interface (see CreateOrder EJB in Figure 1) and a remote home interface. Remote interfaces are provided for use by cluster-external clients, which are usually remote; but they may be also collocated. A remote interface may be used also by a component from the same cluster, if required (see Order using CreateOrder in Figure 1).

A facade EJB may have, in addition to the remote interfaces, also local interfaces. This should be rather an exception; but it may be necessary that an EJB from the same cluster invokes operations of a facade EJB efficiently. If such an EJB would use the remote interface, it would have to pay the remote interface invocation overhead, though it is collocated. By providing also a local interface and a local home interface with a facade EJB, you can avoid this overhead.

2.4. Home Interfaces

A cluster-internal EJB has only a local home interface. For example, the Order, LineItem and Product entity beans have only a local home interface, called e.g. OrderLocalHome, LineItemLocalHome and ProductLocalHome. Each of these local homes returns a reference to the respective local business interface. CreateOrder passes to JNDI each the name of one of these local homes and gets as result of the lookup-operation each a reference. From each local home, CreateOrder obtains the references to the matching local business interface.

A facade EJB has a remote home interface. If it has also a local business interface, it has also a local home interface. For example, CreateOrder has a remote home interface with create- and find-operations that return a reference to the CreateOrder remote business interface.

2.5. Collocation of Clusters

Note that you may allocate different clusters on the same network node, Java virtual machine and, possibly, container. If you do not modify the code written for the case that the clusters may be remote, they will not make use of the fact that they are collocated. An EJB from one cluster does not use cluster-internal EJBs of collocated clusters, and the access from one cluster to the facade EJBs of another cluster is done via the remote interfaces (with the corresponding invocation overhead). In this way, we obtain a location transparency for clusters. The price we pay for this cluster-location transparency is the RMI remote invocation overhead of facade EJBs and the impossibility of invoking cluster-internal EJBs from collocated clusters.

3. Use of Local Interfaces

In this section we analyze under which conditions you can attain the desired performance benefits by the use of EJB 2.0 local interfaces.

Consider a typical use-case of a Web shop like shopping, shown in the UML activity diagram in Fig.2. It consists of several activities like select product, create order, and display order (note that this is a simplified, not a complete shopping use case). The select
product activity displays products and keeps a list of those selected by the customer. In the
create order activity, an order is created by the order in collaboration with its line items and
the products selected by the customer. The display order activity displays the order so that
the customer can check if it is ok.

The example use case is quite typical:
1. There is at least one activity, like create order, in which a set of entities collaborates
tightly
2. There are other activities, like select product or display order, which collaborate loosely
with one or a few of these entities.
3. Different activities use usually different, but overlapping set of entities. For example, a
product is used by select product and create order, and an order is used by create order and
display order (see UML activity diagram Fig.2).

An application typically consists of several use cases. Some of these use cases will usually
use overlapping sets of entities. For example, an additional use case: update shop products,
of the shop will also use products.

To save execution overhead, we should put a set of tightly collaborating entities as cluster-
internal components in a cluster. One of the entity beans, or a session bean that is
introduced should form the facade of the cluster. For example, order, line item, and
product; form a cluster since they collaborate strongly in the activity create order.

We have introduced a facade session bean, called CreateOrder. A remote client, like the
shopping servlet, uses entities from the cluster not directly (to avoid a great number of
remote invocations), but only indirectly via the cluster facade, like CreateOrder.

However, it is a potential problem that entities intended to be cluster-internal EJBs are
possibly used by other activities of the same or even of other use cases. We will take the
activity create order from the shopping-servlet as an example to analyze the problem and
find out which consequences it has. We have to consider in detail how the session bean
CreateOrder obtains information about the product items from the shopping servlet, and
how it returns information about the completed order.
3.1. Cluster with Single Facade EJB

We can form a cluster as shown in Figure 1, with Order, Line Item, and Product as cluster-internal entity beans and Create Order as a facade session bean, only if CreateOrder has a remote interface with operations that do not expose the cluster-internal (usually entity) bean types.

This means that:

1. The Select Product and Shopping activities cannot use Product EJBs and pass them to CreateOrder.
2. CreateOrder cannot return the completed Order EJB to the shopping activity.

When the shopping servlet invokes the createOrder-operation of CreateOrder and adds items passing e.g. the product-numbers, CreateOrder creates the Order and the Product beans via the local home and has a reference to their local interfaces. It handles all product Line Items which the shopping servlet adds in a similar way so that it has local references to them. Eventually, CreateOrder returns e.g. a string that contains the order to the shopping servlet.

That means all invocations of entity beans, from the CreateOrder session bean or among themselves, are done efficiently via the local interfaces.

To summarize: EJB local interfaces are very useful and may improve the performance considerably if a cluster has mainly cluster-internal beans and a single, or relatively few, facade beans. This is only possible if all information related to the cluster-internal beans of the cluster (which are typically entity beans) is handled outside of the cluster in the form of basic Java types or general classes like String, and also passed to and from the facade bean in this form.

3.2. Forced Facades with a Single Interface

However, you may often find conditions that do not follow the given scenario. For example, the facade session bean CreateOrder might return the Order bean to the shopping servlet for display and confirmation by the customer, and obtain the Product beans as items to be ordered from the shopping servlet. Let us analyze the consequences of these seemingly small changes.
If an EJB like Order, intended to be a cluster-internal EJB, is returned to a remote client like the shopping servlet, it is used from outside of the cluster and must become a facade EJB with a remote interface (see Figure 3). We call such a facade a forced facade. A forced facade is tightly used by other beans of the cluster, and loosely used by clients outside of the cluster. The consequence of the introduction of a forced facade is that all collocated invocations from within the cluster are burdened with the RMI invocation overhead.

A serious problem is that the introduction of a forced facade, like Order, may have the consequence that other cluster-internal beans have to be changed in forced facades, too. Suppose the Order bean has in its interface a getItem operation that returns a LineItem. Since a local interface cannot be a parameter of the remote Order interface, the interface of the LineItem bean has to become a remote interface, and LineItem must be changed in a forced facade bean (see Figure 3).

As a consequence, CreateOrder creates an Order and LineItems via the remote home and invokes, though collocated, the operations of Order and LineItem via the remote interface. So we pay the remote interface invocation overhead even for cluster-internal invocations.

With regard to the Product beans, the situation is similar. If the shopping servlet passes these beans to CreateOrder, as a parameter in the remote interface, then they must become forced facade beans like Order and LineItem. In this case, the cluster contains one facade bean, CreateOrder, but no cluster-internal beans; all remaining entity beans are forced facade beans.

The consequence is that we pay the remote interface invocation overhead for all cluster-internal invocations.
As a summary, we cannot make use of local interfaces to improve the performance of clustered beans, which are used also from other activities. The reason is that we need to realize them as forced facade beans with remote interfaces. This causes the same collocated invocation overhead as for EJB 1.1.

### 3.3. Forced Facades with Dual Interfaces

We can avoid the RMI remote interface invocation overhead for forced facade beans at the cost of an increased interface and programming complexity, by introducing dual interfaces. We obtain a forced facade with dual interfaces by adding a local interface to a forced facade.

A *forced facade with dual interfaces* has both a local and a remote interface. The local interface provides the operations used by beans from the same cluster, and the remote interface provides those used by cluster-external clients or beans. The local and remote interface may contain the same operations or different ones. A problem is that the semantics of seemingly identical operations may be different, since a local interface has a reference semantics, and a remote interface a copy semantics.

For example, we may transform Order in a forced facade with dual interfaces by adding a local interface to the Order forced facade bean. The local interface OrderLocalIF contains the operations that are invoked by CreateOrder, like addLineItem( LineItem). The remote interface OrderRemoteIF contains the operations that are invoked by the shopping servlet (see Figure 4), like the getItem-operation that returns a string.

![Figure 4 Cluster with forced facades with dual interfaces: Order and LineItem](image)

Which consequences has the transformation of the Order bean for LineItem? If the LineItem interface is exposed only in the local interface OrderLocalIF, then the LineItem bean can and should be transformed back in a cluster-internal bean. If the LineItem
interface is exposed also in the remote interface OrderRemoteIF, then the LineItem bean could and should be transformed in a forced facade bean with dual interfaces (see Figure 4).

3.4. Reference Transformations between Dual Interfaces

The introduction of dual interfaces creates a new problem: a facade bean like CreateOrder that uses a forced facade bean with dual interfaces, like Order, must sometimes use the local interface and at other times the remote interface of Order. So it has to transform one into the other one.

For example, CreateOrder creates an Order via the local Order home so that it can access the Order entity bean via the local interface OrderLocalIF. After the creation of the Order is completed, CreateOrder should return the Order to the shopping servlet. But CreateOrder cannot return over its remote interface a reference to the local OrderLocalIF interface. CreateOrder has to do a kind of local-to-remote reference transformation, it has to transform the reference to the local OrderLocalIF interface in one to the remote OrderRemoteIF interface. There are two ways to do this transformation:

- Either, the designers of the Order EJB have provided for this problem and added to the OrderLocalIF an operation getRemoteOrder that returns an OrderRemoteIF (remember that a local interface may contain a remote interface as a parameter).

- If the Order designers have not considered this problem in advance, and OrderLocalIF has no such operation, CreateOrder has to do the transformation by itself. After it has obtained the primary key of the Order instance (by a call to OrderLocalIF), it gets via the EJB environment, which is a JNDI naming context, the Order remote home. The find-operation which it invokes with the primary key, returns the OrderRemoteIF of the order instance.

When we transform the Product bean in a forced facade bean with dual interfaces, a similar, but different kind of reference transformation has to be done: since the shopping servlet passes a remote Product reference to CreateOrder, CreateOrder has to transform the product remote interface reference in a product local interface reference.

This is a remote-to-local business interface transformation. It differs from the local-to-remote transformation: a designer of the Product EJB cannot add to the remote product interface an operation getLocalProductInterface returning the local product interface, since a remote interface cannot expose a local one. The only possibility is that CreateOrder does the remote-local transformation by itself, similarly as local-remote transformation described: after CreateOrder has obtained the primary key of the product instance via the remote product interface, it calls the find-operation of the local Product home with the primary key.

3.5. Forced Facades Summary

To summarize, we have to introduce forced facade beans if an activity of a remote client uses entities, which are internal to a cluster of another activity. Introducing forced facades with dual interfaces allows to realize the potential performance benefits of local interfaces, since all invocations from the same cluster are done via local interfaces.
But both an EJB designer and application programmer pay for these performance benefits with a heavily increased design and programming complexity:

An EJB designer of a forced facade with dual interfaces should know or be able to predict beforehand, which operations are required in the local interface and which ones in the remote interface. This will be very difficult or even impossible since it usually depends heavily on the context (i.e. the application and use case) in which an EJB is used. Consequently, a designer will probably put most of the operations in both the local and remote interface. This causes both a semantics-related problem: though the operations look the same, they have different semantics; and a development time and maintenance problem: the work load is heavily increased.

An EJB user of a forced facade with dual interfaces has to find out which interface to use, and has to cope with the different interface semantics. In addition, an EJB user has to program reference transformations, always remote-to-local ones, and the local-to-remote ones in the case where it has not been provided for by the EJB designer.

### 3.6. Cluster with Multiple Facades

As we have seen we have to introduce forced facade beans in the following case: an activity of a remote client uses entities, which are internal to some cluster that is used (via a facade) in another activity. But we should avoid, if possible, to introduce forced facade beans with single interfaces due to the severe performance degradation, which they cause, and forced facade beans with dual interfaces due to the increased programming complexity. The only way to avoid the introduction of forced facades is to isolate a remote client like the shopping servlet completely from all entity beans of a cluster like Product, Order and LineItem.

We isolate a remote client from cluster-internal beans, which it uses, by putting for each activity of the client a new facade bean (which is usually a session bean) between the activity and the cluster-internal (entity) beans, which it uses.

Consider, for example, the shopping servlet. As described, the create order activity uses the cluster consisting of the facade session bean CreateOrder and the entity beans Product, Order and LineItem. Since the Select product activity uses Product, we put a session bean called SelectProduct between the select product activity and the Product entity bean. Similarly, since the display order activity uses the Order, we put a session bean called DisplayOrder between the display order activity and the Order entity bean (see Figure 5).
Let us consider DisplayOrder in more detail. It keeps a reference to a local Order and LineItem (see Figure 5) and has a remote interface with the operations:

String[] getNextLineItem(), and

void setCreatedOrder( CreateOrder c).

The operation setCreatedOrder is of particular interest; it is called from the shopping client to inform DisplayOrder about the Order instance to be displayed. However, the client cannot pass directly the Order entity instance (since it has no reference to it), so it passes the instance of the stateful session bean CreateOrder from which DisplayOrder can get the Order instance.

As a result, the activities of the shopping servlet use only the facade beans SelectProduct, CreateOrder and DisplayOrder when they need access to the entity beans of the cluster. In this way, we isolate all activities of a remote client from the entity beans of the cluster.

An application usually consists of several use cases with activities that use also cluster-internal entities. For example, a Web shop has besides the use case shopping the use case update shop products, which also uses products. We have to apply for all activities of all other use cases that use cluster-internal entities the approach we described for the other activities of the shopping servlet.

That means: if an entity bean like Product used in a new activity is already a part of an existing cluster, we have to add the facade beans of the new activity and possibly additional entity beans used by the new activity to the existing cluster. With each additional use case and activity, we may have to increase the cluster. We will often end up with one or a few huge clusters that comprehend all entity beans and the activity-specific facade (session)
beans. Further, we obtain a very large number of facade (session) beans, nearly as many as there are basic activities in all the use cases of a system.

In some cases, we may be able to reduce the number of facade (session) beans. For example, it may be possible to combine the facade (session) beans CreateOrder and DisplayOrder in a HandleOrder bean with an interface that includes the operations of both the CreateOrder and DisplayOrder interface. However, the reductions we can reach in this way are restricted.

There are many drawbacks connected with this approach:

1. Obtaining one large cluster of all entity beans used in an application is in itself a contradiction to the concept of forming clusters of tightly collaborating components. If all facade (session) beans may use all entity beans, this is component oriented spaghetti-code.

2. Program development and maintenance becomes more complex since when developing an activity you have to design and code not only a client program, but in addition a session bean, and its interface to the client program.

3. Facade (session) beans are not reusable since each one is constructed for the support of one or a few particular activities of a use case. However, the essence of components is reusability. If this is missing, there is nearly no justification to spend the additional effort to build a reusable component.

4. Facade (session) beans created in this way comprise usually no logical unit of work and no transaction management. However, this makes the concept of server-side "activity" components really useful.

3.7. Multiple Clusters with Possibly Multiple Facades

Can we avoid to end up with one large cluster of all application-related beans, for typical application structures? The reason why we obtained one huge cluster is that if two activities each use a smaller cluster, but the clusters overlap in at least one element, then we had to put both clusters together in one.

Suppose, for example, there is a cluster of closely collaborating beans, like the CreateOrder and DisplayOrder facade beans with the cluster-internal entity beans Order, LineItem and Product (see Figure 6). When a bean, like the facade session bean Checkout, of a second cluster collaborates only very loosely with a cluster-internal bean of the first cluster, like Order, we have to combine both clusters in one cluster, if we want to avoid to change Order in a forced facade bean.
Instead of unifying two given overlapping clusters in one cluster, we may do the following: the bean from the second cluster, which loosely uses a cluster-internal bean of the first cluster, should not use it directly, but indirectly via a session bean of the first cluster. That means the Checkout bean would not use the cluster-internal entity bean Order, but the facade session bean DisplayOrder when it needs to access the order in the checkout process (see Figure 6).

The disadvantage of this solution is that we may get very complex use-relations among facade session beans that may even become cyclic. Cyclic use-relationships among session beans are not allowed. But even if the use-relationships are non-cyclic, the session beans will become hard to maintain.

4. Conclusions

We have evaluated the use of EJB 2.0 local interfaces for typical application structures with the following characteristics:

- An application consists of several use cases.
- A use-case consists of several activities.
- An activity (or several activities) collaborates closely with an associated set of entities. Other activities may collaborate loosely with one or a few entities from a set of entities associated with other activities.

For this common kind of application structure, local interfaces do not offer a very good solution. In many cases, if we want to introduce some session beans as facades we may be forced to introduce a large number of forced facade beans, possibly with dual interfaces. This has as a consequence great drawbacks like performance degradation or program development and maintenance complexity.

If we want to avoid this and introduce per activity a session bean as a facade, we cannot avoid a distribution of the responsibility of an activity over the remote client program and facade session bean, with an increased program development and maintenance complexity. In addition, we get either one single large cluster of all EJBs of one application; or if we avoid this, we may get complex use-relationship and dependency structures among facade session beans, or even from entity beans to session beans.
In addition to these disadvantages comes the missing location transparency and the differing semantics of remote and local interfaces.

What we require is a distributed component technology that allows
-a fast collocated invocation for the close collaboration of components within a cluster, as well as
-a slower remote invocation for the loose collaboration among components from different clusters
-but both kinds of invocations should have the same semantics
-and a component developer should not have to provide both a remote and a local interface.

We have developed a distribution technology with a single interface and dual gates that meets these requirements.

Acknowledgements
My thanks are due to Michael Pasquariello who supported me in projects related to EJBs and distributed component technology.

References
[GHJV95] E.Gamma, R.Helm, R.Johnson, J.Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley, 1995
Dynamic Instrumentation for Jini Applications

Denis Reilly
School of Computing and Mathematical Sciences
Liverpool John Moores University
Byrom Street, Liverpool, L3 3AF, UK
email: d.reilly@livjm.ac.uk

A. Taleb-Bendiab
School of Computing and Mathematical Sciences
Liverpool John Moores University
Byrom Street, Liverpool, L3 3AF, UK
email: A.Talebbendiab@livjm.ac.uk

Abstract
Distributed systems are notoriously difficult to develop and manage due to their inherent dynamics, which manifest as component configurations that may change “on-the-fly”. Middleware technologies dramatically simplify the development of distributed systems, but, until recently, little attention has focused on combining software instrumentation techniques with middleware technologies to understand dynamic behaviour and assist with the runtime management of distributed systems. This paper presents a dynamic instrumentation framework, which provides support to monitor and manage Jini applications. The framework adopts a service-oriented approach that employs Jini’s support for code mobility, Java’s dynamic proxy API and Jini’s remote event model to enable runtime insertion and removal of instrumentation services.

1. Introduction
Emmerich, [1], describes a distributed system as: “a collection of autonomous hosts that are connected through a computer network with each host executing components and operating a distributed middleware to enable components to coordinate their activities giving the impression of a single, integrated computing facility”. The distributed middleware, or simply middleware, plays a crucial role by providing APIs and support functions that effectively bridge the gap between network operating system and distributed application components and services. The development of distributed systems is greatly simplified by middleware, but it is still a daunting task due to different component technologies, different protocols and the dynamic behaviour inherent in distributed systems, which can give rise to component/service reconfigurations that occur “on-the-fly”.

Until recently, little attention has focused on the instrumentation of distributed systems, or more particularly on instrumentation as a middleware service to monitor dynamic behaviour. This is not to say that instrumentation has gone unnoticed, since it has been applied for sometime in software engineering to debug and test software applications and

** The term instrumentation is used to refer to “software instrumentation”
also for performance monitoring. Traditional, instrumentation approaches involved the insertion of additional software constructs at design-time, or when the system was off-line, during maintenance, to observe specific events and/or monitor certain parameters. This static instrumentation can be used with distributed systems, but only with limited success due to their dynamic runtime characteristics. This suggests a need for dynamic instrumentation that can be applied at runtime to accommodate any architectural reconfigurations and hence provide a faithful representation of the system’s behaviour.

Jini is a Java-based middleware technology developed by Sun Microsystems [2]. Essentially Jini, together with Java’s Remote Method Invocation (RMI), allows distributed applications to be developed as a series of clients that interact with application services typically via RMI. Jini applications consist of a federation of application and lookup services and collections of clients and Java Virtual Machines (JVMs) distributed across several computing platforms. The real strength of Jini comes from its service-oriented abstraction, which considers a service as a logical concept such as a printer, or chat service that can be discovered dynamically by a client and used according to a contract of use. This abstraction proves extremely useful for developing instrumentation as services, in much the same way as any other Jini application service, and is the basis of our dynamic instrumentation framework.

Based on ongoing research into the development of Jini applications, our main concern in this paper is the development of a dynamic instrumentation framework, implemented through Jini technology, which may be used in turn to monitor and manage Jini applications. In developing such a framework there are a number of questions to address, which include:

(a) What parameters of an application need to be monitored?
(b) What type of instrumentation is required to monitor these parameters?
(c) How can this instrumentation be managed in relation to the application?
(d) What are the instrumentation/management design alternatives?

Through this paper we consider the first three of these questions and postpone the fourth question until our future work, since it exceeds the scope of the paper.

The paper is structured as follows: Section 2 provides a brief historical review of instrumentation and current “state-of-the-art” practice in distributed systems. Section 3 considers the development of the service-oriented framework by first considering the parameters that need to be monitored and then the instrumentation monitor and management services used to monitor these parameters. Section 4 considers the basic design elements that feature in the implementation of the framework and describes an early instrumentation case study. Finally, section 5 draws overall conclusions and mentions directions for future work.

2. Instrumentation

Originally, instrumentation was used to debug and test applications that run on single processor machines and for analyzing the performance of real-time systems. The parallel
computing community later adopted instrumentation to debug, evaluate and visualize parallel applications. More recently distributed system developers have recognized the potentials of instrumentation, used in a *dynamic* regime, to monitor and manage today’s distributed systems.

Probably the earliest documented use of software instrumentation was that of *dynamic analyzers*, first considered by Satterthwaite, [3], which consisted of:

(a) An instrumentation system

(b) A monitoring and display system.

Dynamic analyzers were used as part of a more comprehensive test environment, [4] (Sommerville), and were closely associated (even integrated) with the compiler, through which they could be switched on or off by a compiler directive. Two of the main problems of dynamic analyzers, as noted in [4], were: first, they relied on source code instrumentation, which was not always possible when the program relied on additional pre-compiled libraries. Second, the instrumentation code often affected program performance, which presented problems in real-time applications.

Significant advances in the use of instrumentation came from the parallel computing community through the development of support tools to debug, evaluate and visualize parallel programs, [5] (Rover) and [6] (Simmons and Koskela). These advances saw instrumentation applied in a much more structured manner, which followed a “tools” or “library” based approach, [7] (Waheed and Rover) and [8] (Geist *et al.*). Emphasis was placed on analyzing processes and the transportation and storage of data, within a parallel application. Visualization tools, [9] (Heath and Etheridge) and [10] (Hao *et al.*), were usually based on GUIs developed using graphics libraries such as OpenGL, Tcl/Tk and X/Motif that provided the user with a consistent view of the application and its environment. Performance, which is significant in parallel programs, was evaluated using *unobtrusive* instrumentation that did not carry an additional computational overhead. The parallel computing instrumentation tools were capable of synchronizing with applications and providing limited interaction facilities, but they were still generally static in nature.

The more recent “state of the art” developments have adopted component-based and service-oriented abstractions to provide dynamic instrumentation capabilities. Sun Microsystems has made a significant contribution through the Java Management Extension API, (JMX) [11], which is an optional extra to Java v1.3 that facilitates the instrumentation of Java-based applications. JMX uses Management Beans (*MBeans*), which are arranged into instrumentation and agent levels to monitor distributed services. The MBean approach is also used by the Openwings community, [12], who adopt the component-connector-port view of a distributed system, through which components are connected together using protocol specific connectors that plug into ports, within the components, to facilitate synchronous and asynchronous communications between components.

The DARPA funded initiative for Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA) is actively investigating the use of software gauges to dynamically deduce component configurations and examine distributed systems as an assemblage of components, [13] (Wells and Nagy), [14] (Wolf and Kean) and [15]
(Garlan and Stratton). Reflective techniques are used by Diakov et al. in [16] to monitor distributed component interactions by combining CORBA’s interceptor mechanism together with Java’s thread API to “peek” into the implementation of CORBA components at runtime.

More specific to Jini-based systems, the Rio†† project, [17], has made a significant contribution through an architecture that simplifies the development of Jini federations by providing concepts and capabilities that extend Jini into the areas of QoS, dynamic deployment and fault detection and recovery. Rio makes use of Jini Service Beans (JSBs), Monitor Services and Operational Strings, were the latter are used to represent the collection of services and infrastructure components as an XML document. Also of interest in Rio is the Watchable framework, which provides a mechanism to collect and analyze programmer-defined metrics in distributed systems.

Further useful Jini-based contributions come from [18] (Fahrimair et al.) and [19] (Hasselmeyer and Voß). In [18] Fahrimair et al. describe the Carp@ system, which is a reflective based tool for observing the behaviour of Jini services. Carp@ combines the ideas of reflection together with Jini services to provide a meta architecture that reflects on a Jini application. Carp@ instruments a Jini application using a model based on the use of Carp@ Beans for instrumentation that communicate through channels and ports. In [19] Hasselmeyer and Voß describe a generic approach to instrumentation that effectively instruments a Jini lookup service using Java’s dynamic proxies to trace component interactions in a Jini federation. In [20] Hasselmeyer extends on this earlier work by considering the management of service dependencies in service-centric applications.

Our contribution makes use of several ideas described in [19], particularly the use of dynamic proxies and monitor services. However, we use different approaches to those of [19] to implement monitor services and instrument a Jini application. Also our use of monitor services is focused on monitoring specific parameters, which are significant to the Jini applications developed from our previous work.

3. Service-oriented Instrumentation

This section considers the first three questions raised in the introductory section (Section 1), which are repeated below:

(a) What parameters of an application need to be monitored?

(b) What instrumentation is needed to monitor these parameters?

(c) How can this instrumentation be managed in relation to the application?

As mentioned previously, consideration of the fourth question exceeds the scope of this paper and is postponed until our future work.

†† Rio is a Jini Community [21] project
3.1. Instrumentation Parameters

The selection of a series of parameters depends largely on the abstraction of a distributed system. The abstraction used by Openwings and the Carp@ system is based on components, connectors (or channels) and ports. Through this abstraction components, which consist of one or more objects, communicate through connectors or channels that are plugged into ports on the components. The alternative abstraction, used in Rio and [19], is the service-oriented abstraction that regards a distributed system as a federation of services and clients were clients may be either “pure” clients or service providers themselves in a peer-to-peer regime.

Our approach adopts the same service-oriented abstraction as Rio, and [19], which based our selection of parameters primarily on services (application services and lookup services), but also took into account the Jini application as a single entity (i.e. the federation of services, pure clients and JVMs), as shown in Table 1. Although Table 1 does not provide an exhaustive list, we feel that these parameters can provide useful information relating to performance and behaviour of Jini applications.

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Service</td>
<td>number of clients</td>
</tr>
<tr>
<td></td>
<td>dependent services</td>
</tr>
<tr>
<td></td>
<td>period of use before discarded</td>
</tr>
<tr>
<td></td>
<td>events generated</td>
</tr>
<tr>
<td></td>
<td>events received</td>
</tr>
<tr>
<td></td>
<td>member function access</td>
</tr>
<tr>
<td></td>
<td>values of attributes</td>
</tr>
<tr>
<td>Lookup Service</td>
<td>frequency of client access</td>
</tr>
<tr>
<td></td>
<td>frequency of service registration</td>
</tr>
<tr>
<td></td>
<td>number of services registered</td>
</tr>
<tr>
<td></td>
<td>size of serialized code</td>
</tr>
<tr>
<td></td>
<td>Physical location of service</td>
</tr>
<tr>
<td>Jini Application</td>
<td>number of services per group</td>
</tr>
<tr>
<td></td>
<td>number of groups</td>
</tr>
<tr>
<td></td>
<td>number of pure clients</td>
</tr>
<tr>
<td></td>
<td>number of JVMs</td>
</tr>
<tr>
<td></td>
<td>client / service access patterns</td>
</tr>
</tbody>
</table>

Table 3: instrumentation parameters according to category

Next we consider question (b) concerning the instrumentation that is needed to monitor these parameters.
3.2. Instrumentation Monitor Services

Within Table 1 there are four different types of parameter:

(a) Information parameters – such as the “id.” of a service or its physical location.
(b) Numeric data parameters – such as the number of clients or number of application services registered with a lookup service.
(c) Dynamic parameters – which are event driven, such as the invocation of a service method, or specific events generated and/or received by a service.
(d) Complex parameters – such as client/service access patterns, which are obtained by post-processing the previous parameters.

In order to monitor the different types of parameter we use an instrumentation monitor service (or simply monitor service) for each application service to be instrumented. The monitor service is a Jini service, and it can make use of two utility classes, namely logger and analyzer, which are standard Java classes (not Jini services). The responsibilities of the monitor service and logger and analyzer utilities, with respect to the different parameter types, are:

(a) Monitor service – “connects” with an application service or a lookup service and monitors dynamic parameters and invokes logger and/or analyzer utilities.
(b) Logger utility – invoked by a monitor to record information and numeric data parameters.
(c) Analyzer utility – invoked by a monitor to post-process parameter types (a), (b) and (c) and hence compute complex parameters.

Finally we consider question (c) concerning the management of the instrumentation.

3.3. Monitor Management Services

Monitor services alone can only monitor behaviour and request the use of loggers and analyzers. They need to be created and controlled during their lifecycle, which suggests a need for monitor management services (or simply management services). The management services create and interact with the monitor services, during their lifecycle, triggered by specific remote instrumentation events generated within a Jini federation. Because management services need to be aware of such remote events they too must be Jini services and must first register with a Jini lookup service. Essentially the management services provide the dynamic capabilities to the instrumentation framework by responding to event requests for instrumentation services. They are also responsible for maintaining information relating to the current monitor service configuration, which provides a snapshot of the current instrumentation services. An overview of the instrumentation framework is shown in Figure 1, which is explained further below.
The right side of Figure 1 shows a Jini application, which consists of: one or more application services; one or more lookup services; one or more pure clients and one or more JVMs. The left side of Figure 1 shows the instrumentation system that monitors the Jini application, which consists of: one or more monitor services, each of which is associated with a single management service and a single optional logger and/or a single optional analyzer; a single monitor observations and measurements record and a single monitor configuration record. Each management service first registers with a lookup service and then creates and interacts with a monitor service. By registering with a lookup service a management service is made aware of remote instrumentation events that may occur within a Jini application. The logger and/or analyzer instrumentation utilities, which are not Jini services, can be called by a monitor service to record or compute parameters. Also the logger utilities can directly log parameters of the applications JVMs and the analyzer utilities compute information, which, when gathered together, reflects the performance and behaviour of the Jini application as a single entity. The monitor services, logger and analyzer utilities can all update an observations and measurements record, which could range from a series of files to a detailed GUI. Finally, the management services maintain a monitor configuration record that reflects the current monitor service configuration, which again could be a series of files or an XML document.

4. **Implementing Instrumentation in Jini Applications**

In this section we consider the “on-going” development and implementation of the instrumentation framework and in particular, how instrumentation services can be implemented by combining Jini’s code mobility capabilities, Java’s dynamic proxies and Jini’s event model. The section finishes with the description of an early case-study used to “field-trial” the instrumentation framework.
4.1. Overview of Jini Technology – Code Mobility

As mentioned previously, a Jini application consists of a collection of application services, a collection of clients that use the application services, a collection of lookup services, with which the application services register, and a collection of JVMs that may be distributed across several computing platforms. In order to provide an overview of Jini we shall make use of material from Newmarch [22], to help consider a simple application that consists of three components: an application service (or simply a service), a client and a lookup service. There is also an additional “hidden” fourth component, which is a network connecting the previous three together, and this network will generally be running TCP/IP (although the Jini specification is independent of network protocol).

Jini’s API provides code mobility in that code can be moved around between these three components, over the network, by marshalling the objects. This marshalling involves serializing the objects (using Java’s Serialization API) in such a way that they can be moved around the network, stored in a “freeze-dried” form, and later reconstituted by using included information about the class files as well as instance data. This marshalling is represented in Figure 2 using the block arrows with broken lines.

Two events must take place in order for the client to use the application service:

(a) First the application service, which consists of an implementation and a proxy, must register with a Jini lookup service. Sun Microsystems Jini implementation provides a lookup service (called reggie), which “listens” on a port for registration requests. When a request is received a dialog between the application service and lookup service takes place after which a copy of the service proxy is moved to and stored in the lookup service.

(b) Second the client must find the service. This again, unsurprisingly, involves the lookup service, which also listens for incoming requests from clients that want to use services. The client makes its request using a template, which is checked against the service proxies that are currently stored on the lookup service. If a match is found a copy of the matching service proxy is moved from the lookup service to the client.

At this stage there are three copies of the service proxy in existence (Figure 2), one in the application service, one in the lookup service and now one in the client. Jini application service proxies are implemented as Java interfaces that specify the signatures of methods, which are implemented by the service implementation. The client can interact with its copy of the service proxy by invoking any of the specified methods. These method invocations are then routed back to the service implementation, typically using RMI, resulting in the invocation of a method in the service implementation. The overall effect of this routing is that the client “thinks” that it has its own local copy of the service implementation and proceeds to use the service by invoking its methods being unaware of the physical location of the service within the network.
4.2. Dynamic Proxies

The proxy is a design pattern, specified by Gamma et al., [23], which is frequently used in object-oriented programming to force method invocations on an object to occur indirectly through a second object – the proxy object, which essentially acts as a surrogate or delegate for the underlying object being proxied. Several variations on the basic proxy, also described in [23], are: access, remote, virtual and dynamic proxies. Sun Microsystems realized the potential of the dynamic proxy for implementing strongly typed, or type-safe Java applications and introduced the Dynamic Proxy API in Java v1.3 for use in conjunction with Java interfaces. The paragraph below draws on material presented in the Java language API documentation, [24], to explain the mechanism of the dynamic proxy.

Java uses interfaces to define behaviour that can be implemented by any class anywhere in a class hierarchy. A Java interface defines a set of methods but does not implement them, instead relying on a class to implement the methods, which thereby agrees to provide the behaviour specified by the interface. A dynamic proxy class extends on Java’s use of interfaces by providing capabilities to actually implement a list of interfaces that are specified at runtime. A method invocation through one of the interfaces on an instance of a dynamic proxy class is encoded and dispatched to another object through a uniform interface. In more detail, the method invocation is dispatched to a special invocation handler method in the dynamic proxy instance, where it is encoded, using Java’s reflection API, to deliver a Java object (java.lang.Object) identifying the method that was invoked and an array of Java objects containing the arguments used in the invocation. This provides an extremely powerful tool for implementing instrumentation, because dynamic interactions between application services and clients, occurring as method invocations, can be made available to an instrumentation monitor service (i.e. a dynamic proxy) in Java object form!

4.3. Implementing Monitor Services

The dynamic proxy provides an ideal mechanism for the implementation of the monitor service. Section 4.1 stated that Jini application service proxies are implemented as Java interfaces, so an application service proxy can be supplied as the list of interfaces that a monitor service, as a dynamic proxy, should implement. This is illustrated in Figure 3, in which the top half of the figure shows the same basic Jini system that was considered
previously in Section 4.1 and the bottom half now shows the application service proxy (small ellipse) “wrapped up” in a monitor service (large ellipse) to create a “compound” proxy, such that the monitor service (i.e. dynamic proxy) implements the application service proxy (i.e. Java interface).

The power of the dynamic proxy, when used as a wrapper for the application service proxy, is that it allows normal access to the application service proxy from the client to proceed as before. However, after a client has invoked a service implementation method, via the application service proxy, then control returns back to the monitor service, which now has the Java objects that identify the method invoked and the parameters used in the invocation along with any return values. When used in this way the monitor service can be regarded as providing an invocation handling function that provides information, which is extremely useful for observing Jini client/service interactions.

Figure 8: monitor service overview

Figure 4 shows an outline UML representation of a monitor service that instruments an application service, which implements the methods: method1, method2 and method3.
4.4. Implementing Management Services

Monitor management services are implemented just like any other Jini service, which requires that they register with a lookup service, but what is significant to their implementation is their event handling capabilities, based on Jini’s remote event API.

In a typical instrumentation scenario management services are created on demand by an instrumentation factory (section 4.5). After creation management services register with a lookup service through a management proxy (a Java interface) and then wait for instrumentation requests from within the Jini application. Requests are made using specific remote instrumentation events and a management service is notified of these events by the lookup service with which it has registered. If an event is received for an application service to be instrumented, before use by a client, then the management service creates a monitor service (as a dynamic proxy) and adds the application service proxy to the monitor service. The client then receives the monitor service containing the application service proxy. The client can then invoke methods on the application proxy and after each invocation control returns to the monitor service as considered previously (Section 4.3).

If an application service is currently being used by a client in an “un-instrumented” state then it can be dynamically instrumented by arranging for the management service to notify the client that its copy of the application service proxy is “out of date” through a specific remote instrumentation event. After this notification the management service sets about the creation of a new compound proxy by creating the monitor service to which it then adds the application service proxy as before. The client is then sent the new revised proxy, which is the monitor service containing the original application service proxy and as before the monitor service can monitor method invocations made on the application service proxy.

Essentially, management services orchestrate the dynamic instrumentation by acting on Jini remote instrumentation events. However, management services also maintain configuration information about the current instrumentation monitor services, within the federation,
through the use of an XML document, similar to the Operational Strings used in Rio. The XML document may be accessed by any client, application service, monitor service or management service, within the application, or even by the user, through a GUI, to provide a snapshot of the current instrumentation configuration.

![Diagram](image)

**Figure 10: management service overview**

Figure 5 essentially supplements the previous Figure 3 with a management service (assumed to have been created by a factory class), which has registered with a lookup service, through its management proxy, so that it may be notified of remote instrumentation
events within the application. On receiving an instrumentation event, the management service has then created a monitor service and added the application service proxy to the monitor service and registered the monitor service with the lookup service. The monitor service has requested the use of a logger utility (but no analyser – simply to avoid diagram clutter), which is used to assist in maintaining the observations and measurements record that reflects the service/client interaction. Finally, the management service updates the monitor configuration record to reflect the addition of a new monitor service. An outline UML representation of the management service is shown in Figure 6.

4.5. Instrumentation Case Study

As mentioned previously, the development and implementation of the framework is ongoing. However, to date, the basic architecture underlying the framework has already been implemented and tested on an existing Jini application, namely EmergeITS\(^3\), [25], (Reilly and Taleb-Bendiab), which was already conveniently available from our previous work. EmergeITS is intended to realize the concept of intelligent networked vehicles, primarily for use by the emergency fire service. Essentially EmergeITS allows emergency fire service personnel to access a variety of application services, from centralized corporate systems through remote in-vehicle computers and Palm and WAP phone devices. Figure 7 shows a simplified version of the EmergeITS architecture, which consists of a collection of application services and a service manager responsible for registering application services and managing their leases. Application services are discovered and used accordingly by one or more in-vehicle client computers.

---

\(^3\) EmergeITS is a collaborative project between the School of Computing and Mathematical Sciences at Liverpool John Moores University and Merseyside Fire Service <http://www.cms.livjm.ac.uk/emergeits>
An important design decision facing EmergeITS was the choice of service implementation. The alternatives were “conventional” lease-based services, which subclass RMI’s `UnicastRemoteObject` class, or “lazy” activation-based services, which subclass RMI’s `Activatable` class. Conventional lease-based services exist within a server, where they are kept alive, consuming memory, even when idle, until they are eventually discarded. A lazy activatable service, considered further in [22], registers with a lookup service, via a proxy, just like a conventional service, but after registration, when the activatable service becomes idle, it is allowed to “die” or “sleep”, thereby consuming no memory. RMI’s daemon, `rmid`, maintains references to the dormant service so that it can be resurrected when needed by a client. On first impression, activatable services seem like an attractive option. However, the trade off with activatable services is that although memory usage is reduced a new JVM needs to be started, when a service is called into use (i.e. its methods are invoked). The decision to “mix and match” both types of service implementation was based on intuition in that it was self evident as to which services would be accessed frequently (implemented as conventional lease-based services – shown as plain rectangles in Figure 7) and which would be accessed infrequently (implemented as activatable services – shown as shaded rectangles in Figure 7).

Recently, the EmergeITS application was instrumented to essentially “field-trial” the mechanics of our instrumentation framework and also to confirm our intuition regarding the previously mentioned EmergeITS design decision. An Instrument Factory, controlled by an Instrumentation Manager (a main driver program), was used to create instrumentation services (management and monitor services) as and when they were needed. The instrumentation services were applied such that conventional lease-based services were instrumented when first registered with a lookup service (i.e. before use by any client) and activatable services were instrumented on activation (i.e. whenever used by a client).

The parameters of interest, during the trial, were the number of JVMs started, as a consequence of invocations made on activatable services, and the memory conserved
through the use of activatable services. Logger and analyzer utilities were used to record
and post-process the information acquired by the instrumentation monitor services
respectively. This information was stored in the Application Recorder, which was used to
gather the performance and behavioural parameters obtained by the individual
instrumentation services to reflect the overall performance and behaviour of the EmergeITS
application as a single entity. Figure 8 shows a snapshot of the instrumentation system in
operation after the creation of five instrumentation services: IS1, IS2, IS3, IS4 and IS5. Only
one client is included, to avoid clutter, which is shown accessing the Database/XML
Servlet activatable service and the figure is annotated with the sequence of events necessary
to dynamically instrument this activatable service.

Figure 8: instrumentation of EmergeITS
5. Conclusions

1. Static instrumentation is insufficient for monitoring and measuring the behaviour and performance of distributed systems. Dynamic instrumentation is needed to accommodate the dynamics inherent in distributed systems and provide a faithful representation of their behaviour and performance.

2. Dynamic instrumentation has grasped the attention of several major contenders interested in the development of distributed systems, namely Sun Microsystems, the Jini Community and the Openwings Community, which has led to cooperative efforts resulting in the development of JMX, Rio and Openwings architectures respectively.

3. Four important questions that need to be addressed relating to dynamic instrumentation are: what should be monitored, what instrumentation is needed to perform the monitoring, how can the instrumentation be managed and what are the design alternatives? We have considered the first three questions and intend to consider the fourth in our future work.

4. Instrumentation can be implemented using only a small number of instrumentation services and supporting utilities. Our framework is based on the use of a monitor service per application service, which can call on the use of logger and/or analyzer utilities to monitor Jini application services and lookup services.

5. Management services are needed to control, coordinate and maintain configuration information relating to the instrumentation monitor services applied to a Jini application, or more generally a distributed system.

6. The dynamic proxy API of Java v1.3 provides an ideal construct for implementing dynamic instrumentation services in Jini or Java applications due to its ability to implement a list of Java interfaces that may be specified at runtime and acknowledge method invocations made through these interfaces.

7. Instrumentation services can be used to assess the performance of Jini applications, which combine the use of conventional lease-based and lazy activatable service implementations to provide information on which “mix-and-match” design alternatives may be evaluated.

We intend to continue the development of our dynamic instrumentation framework on several fronts:

(a) First, to combine the low-level instrumentation parameters of Table 1 to produce meaningful metrics against which performance and behaviour can be measured.

(b) Second, to extend the ideas of our framework to consider further distributed component/service technologies, particularly CORBA and Web Services.

(c) Third, to consider the issues of dynamic service dependencies, discussed in [20], so that dependencies can be identified, instrumented and managed to facilitate fault tolerance and distributed application migration.
It is also our intention to obtain further test results, which can be used to compare our approach against those of others concerned with the management and monitoring of distributed applications. This in turn will help to address the fourth question regarding instrumentation design alternatives.

References


176


Session V – Services
Message Queuing Patterns for Middleware-Mediated Transactions

Stefan Tai*, Alexander Totok**, Thomas Mikalsen*, Isabelle Rouvellou*
*IBM T.J. Watson Research Center, New York, USA
{stai, tommi, rouvellou}@us.ibm.com
**Courant Institute of Mathematical Sciences, New York University, New York, USA
totok@cs.nyu.edu

Abstract

Many enterprise applications require the use of object-oriented middleware and message-oriented middleware in combination. Middleware-mediated transactions have been proposed as a transaction model to address reliability of such applications; they extend distributed object transactions to include message-oriented transactions. In this paper, we present three message queuing patterns that we have found useful for implementing middleware-mediated transactions. We discuss and show how the patterns can be applied to support guaranteed compensation in the engineering of transactional enterprise applications.

1. Introduction

Object-oriented middleware (OOM) and message-oriented middleware (MOM) are two widely used but different kinds of middleware. OOM, as exemplified by CORBA and Enterprise JavaBeans, promotes a synchronous, coupled, interface-driven communication style; MOM, as exemplified by MQSeries [7] and implementations of the Java Message Service (JMS) [17], promotes an asynchronous, decoupled, data-driven communication style. Many enterprise applications require the use of both OOM and MOM for integrating diverse legacy systems [14].

Two different notions of transactions correspondingly exist with OOM and MOM to address different kinds of reliability concerns. Distributed object transactions [15] are based on the X/Open distributed transaction processing model; they group a set of (remote) object invocations so that their execution is enclosed into an atomic sphere. Message-oriented transactions (messaging transactions) [2] group the publication and consumption of different messages (enqueuing/dequeuing to/from message queues) into an atomic unit-of-work. Distributed object transactions assume synchronous interactions and a life-cycle dependency between transaction participants (i.e., all object servers need to be available at the time of transaction processing), whereas message-oriented transactions separate the execution of sender and receiver programs, assuming eventual processing of requests by life-cycle independent message receivers.

Object transactions and messaging transactions each have their strengths and advantages, but often need to be combined in order to implement transactions that span across OOM application components and MOM application components. Different strategies for integrating the two have been identified [19]. One approach is to extend object transactions to
include message-oriented transactions, as proposed by the Dependency-Spheres model [20] and the X²-TS model [12]. Dependency-Spheres and X²-TS transactions are middleware-mediated transactions [13] – distributed object transactions that integrate messaging as a communication style using messaging middleware for mediated interaction with loosely-coupled application components.

Middleware-mediated transactions require sophisticated system support to deal with recovery and fault-tolerance of object and messaging components. In this paper, we describe three message queuing patterns that we have found useful for implementing middleware-mediated transactions. We demonstrate the use of these patterns to support guaranteed compensation in the engineering of transactional enterprise applications.

The paper is structured as follows. In Section 2, we briefly summarize the concept of middleware-mediated transactions. In Section 3, we present the three message queuing patterns of fire-and-remember, recoverable state machine, and detached compensation for orphaned transactions. We discuss related work in Section 4, and conclude with a summary in Section 5.

2. Combining Object Transactions and Messaging Transactions

Figure 1 illustrates a common enterprise application scenario, where OOM application components (Object1, Object2, Object3) interact with each other and with MOM application components (Recipient1, Recipient2, Recipient3) using object interfaces (OOM brokering) and message queues (explicit MOM mediation), respectively.

With standard middleware, no atomic unit-of-work grouping the diverse component actions and interactions can be defined. Standard middleware requires the single logical transaction of Figure 1 to be separated into multiple, independent transactions:

- The object transaction of the OOM application components, which may include on-commit message "put" requests to the queues of the MOM application components
- The transaction of Recipient2, which includes a message read from the queue of Recipient2, some processing, and a message "put" to the queue of Object1
The transaction of Recipient3, which includes a message read from the queue of Recipient3, some processing, and a message "put" to the queue of Object1

The transaction of Object1, which includes reading the result messages produced by Recipient2 and Recipient3 from the queue of Object1, and performing some subsequent processing

Standard middleware does not define and support any dependency management of these individual transactions; it is the responsibility of the application developer to program the global transaction and to ensure atomicity and system consistency in the presence of application and system failures.

Middleware-mediated transactions [13] overcome this limitation. Middleware-mediated transactions have been proposed as a framework for combining object transactions and messaging transactions, integrating concepts of standard transaction middleware, extended transaction models [4], and distributed programming language systems [8]. With middleware-mediated transactions like Dependency-Spheres [20], a global transaction context is established so that messaging interactions and individual messaging contexts of MOM application components are associated with a transactional object context. Therefore, in addition to object invocations, a transactional object client can also

- immediately publish or consume (sets of) MOM messages during an ongoing distributed object transaction as part of the object transaction,
- have message recipients with independent life-cycles and execution contexts be associated as participants of the object transaction, so that message recipient actions can affect the outcome of the object transaction, and vice versa.

Middleware-mediated transactions require compensation support for messages, as messages can be sent with immediate visibility during a transaction. Compensation is generally well-accepted as an important way to deal with failures [2], and fundamental not only to middleware-mediated transactions, but also to other extended transaction models such as Sagas [5] or ConTracts [16], to workflow and process support systems [11] [6], and conversational transactions of business-to-business interactions [3]. When a transaction fails, a cancellation/compensation message is imperative to undo any processing actions that the message recipient may have performed. The definition of the compensation semantics is an application responsibility, but the guarantee that a recipient receives a compensating message if a transaction fails (and also, that no recipient receives a compensating message if it has not prior received a primary message) should not be an application responsibility, but rather a function of the middleware.

In this paper, we report on our experiences in the design and implementation of middleware system support for guaranteed compensation in middleware-mediated transactions. We present three message queuing patterns, which are all motivated by transaction processing and compensation needs, but which may also be useful to address other, possibly non-transactional, application problems as well. The patterns have been implemented as part of the Dependency-Spheres middleware system, the principal architecture of which is described in [20] and [21].
3. Message Queuing Patterns

We introduce the three message queuing patterns of fire-and-remember (Section 3.1), recoverable state machine (Section 3.2), and detached compensation for orphaned transactions (Section 3.2). Each pattern is described in three subsections of problem/objective, solution/implementation, and use/consequences.

These patterns represent solutions that we have found useful and practicable as extensions to standard middleware. We believe that they can be applied not only as middleware extensions, but also to solve related design and implementation problems on the application level.

3.1. Fire-and-Remember

Standard message middleware [7] [17] supports two kinds of message sends: the immediate delivery of a message at the time it is created, and the (MOM-transactional) commit-based delivery of a message. A message is any application data plus MOM control data (and optionally also application control data), such as a unique message id or a message delivery timestamp. In a middleware-mediated transaction processing context, however, it may be useful to send a message predicated on the failure of the transaction (on-abort); this is not supported with standard MOM transactions.

Problem/Objective

Consider an application that implements a middleware-mediated transaction which includes messages that are sent out with immediate visibility. The application wants the simplicity to "fire and forget" when sending out a message. Yet, as firing takes place within a transaction, the application must "remember" something about the message in order to compensate in the event of a failure. The problem is amplified when the transaction failure model is complex. For instance, a particular set of acknowledgments of message receipt or message processing by final recipients may be required in order for the transaction to succeed ("conditional messaging") [21].

Ideally, the application would create a corresponding compensating message at the same time the primary message is sent, and have the delivery be predicated on the failure of the transaction. However, standard middleware does not support this type of message send. Therefore, the application typically will

- create some data structure for a compensating message,
- add application data for compensation and control data such as timestamps and the id of the primary message that it compensates, and
- store the data persistently.

Should the transaction abort, the application must then be (made) available to start a new messaging session to create and send out actual compensating messages using the stored data.

The objective of the "fire and remember" pattern is to support on-abort delivery semantics allowing an application to fire a primary message, create a corresponding compensating
message at the same time, and have the middleware remember the compensating message to deliver it in case of a transaction failure. For every primary message that is sent, the existence of a compensating message can then be guaranteed. Furthermore, the application is no longer required to be (made) available should the transaction abort.

Solution/Implementation

The fire-and-remember messaging pattern is suggested to implement on-abort message transmission. For this purpose, a simple middleware system extending the functions of standard middleware (Middleware Extension MWx) is implemented, and a persistent message queue that is used to temporarily store compensating messages (COMP.Q) is set up (see Figure 2).

![Figure 2: Fire-and-Remember” Messaging Pattern](image)

The COMP.Q is used as the destination for compensating messages. The compensating messages are put on this queue at the same time that the primary messages are sent out. The application encodes the actual message recipients for each compensating message using an agreed message property field, so that the MWx system can forward the compensating messages at a later time. The application itself can then forget about the messages.

The MWx system observes the sender’s transaction, and once a transaction fails,

- the corresponding compensating messages are read from the COMP.Q queue (qualified reads based on the transaction id),
- the actual message recipient addresses are extracted from the messages, and
- the messages are forwarded to their designated destinations.

Using MOM transactional capabilities, compensating messages are forwarded atomically to ensure compensation across all required participants. This includes the atomic grouping of the "get" requests from the COMP.Q queue with the "put" requests to the target application queues.

The MWx system uses an agreed message property field to associate compensating messages to a transaction. The MWx system further encodes which action to take should the transaction succeed. For example, all compensating messages may be deleted.
Use/Consequences

The pattern requires a compensation queue (COMP.Q) to be set up. A further requirement is that the \texttt{MWx} system needs to be able to determine the transaction status of the sender application; the \texttt{MWx} system can register itself with the transaction monitor as a participant of the transaction (in order to be notified about the transaction status), or, the \texttt{MWx} system can be part of the transaction monitor itself.

We have implemented the \texttt{MWx} system as part of the Dependency-Spheres transaction monitor. We have found the "fire-and-remember" pattern to be a very simple and inexpensive solution that can easily be implemented. The pattern helps to significantly reduce application code and application programming complexity, as most of the required functions are now provided by the middleware.

3.2. Recoverable State Machine

Engineering a software system typically entails the definition of a model of the behaviour of the system. A state machine is such a model that specifies the sequences of states that a system goes through in response to events (such as signals, operations, or passing of time). It describes which state-dependent activities take place whenever an event occurs and when the system transitions from one state to another. In the following, we present a solution that represents and executes state machines reliably using persistent message queues and MOM transactions.

Problem/Objective

The implementation of a recoverable state machine involves the use of some persistent store to represent states, and the use of a reliable mechanism for transitioning between states. A persistent store is required in order to survive any hard- and software failures. A reliable transition mechanism is required to ensure that only consistent states are stored.

Databases and database transactions are a common, established approach to address this problem. However, if any pre-requisite for a database is to be removed (that is, a database is not otherwise required or desired), message queuing represents an alternative. Persistent queues and MOM transactions compare to databases, even though they have not been designed to replace databases (queues have been designed for remote messaging that is also persistent and transactionally reliable). Furthermore, by using queues for state machine representation and execution, MOM functionality can be employed to integrate any state-based notification that is of interest to the application. For example, in a transaction processing context, the event of entering the "aborting" state of a transaction may cause compensating messages to be automatically be sent out (see below).

Solution/Implementation

In our approach, we represent each state of a state machine as a persistent message queue. In the following, we illustrate the pattern with a state machine that models the states of a transaction. The transaction whose state is to be monitored is represented as a message, which is moved like a token from queue to queue whenever a state transition occurs. Each state transition is a MOM transaction that atomically groups the "get" operation from one state queue with the "put" operation to another state queue.
Figure 3 depicts the state machine of middleware-mediated transactions, and the corresponding persistent message queues, which are part of a larger middleware system (the transaction monitor) supporting the functions (API) and semantics of middleware-mediated transactions.

Once a transaction is started, a message representing the transaction is created and put into the **ACTIVE** queue. The transaction remains active until the application tries to commit or to explicitly abort the transaction, or until a failure that ends the transaction occurs. A MOM transaction that moves the transaction message token to the **COMMITTING** or **ABORTING** queue is associated with these events as part of the commit() and abort() operation implementations, and as part of the component of the middleware system that detects application and system failures (see Section 3.3 below). The transaction message token remains in the respective queues as long as the commit or abort processes are ongoing (see [20] for a description of these processes). Whereas the commit process may also be timed out by the application, the aborting process cannot be timed out, but must be fully completed.

**Use/Consequences**

The use of persistent queues and MOM transactions allows to monitor multiple transactions independently of each other; the set of all active, aborting, aborted, committing, or committed transactions is easily determined by browsing the respective queues. As each queue is a persistent store that will survive any hard- and software failures, they reliably record the consistent state of transactions. Should a system failure occur (such as a crash of an application client or even a crash of the transaction monitor itself), the state of all transactions can be reconstructed based on the persistent queue data. Each transaction message token is guaranteed to exist in only one of the state queues at a given time, as MOM transactions are used to reliably move the message token between the state queues.
The queue-based state machine implementation can be combined with other messaging solutions and MOM functions, such as the "fire-and-remember" pattern introduced above. Compensating messages that are stored in the COMP.Q queue only need to be sent out when compensation is required, which corresponds to the transaction entering the ABORTING state. A message listener for the ABORTING queue could be used to automatically trigger a corresponding compensation process (the transaction monitor component that sends out the compensating messages that are stored in the COMP.Q queue).

A potential concern with this approach is its ability to scale: the number of queues that need to be set up and administered increases with the number of states in the state machine, and the number of MOM transactions executed increases with the state transitions taken. However, the distributed state representation lends itself naturally to workload management, failover, and parallelism.

3.3. Detached Compensation for Orphaned Transactions

Middleware-mediated transactions and other compensation-based processes and applications must perform compensation not only for explicitly aborted transactions or processes, but also for "orphaned" transactions. An orphaned transaction is an active transaction that cannot be committed or aborted because the application client process that started it has terminated abnormally. This can occur due to a system error such as a machine crash. In the following, we describe a pattern for detecting and driving compensation for orphaned transactions.

Problem/Objective

In middleware-mediated transactions, participants are loosely-coupled programs, with independent life-cycles, that communicate using asynchronous messaging. In this environment, there is no obvious difference between a participant not reacting due to a failure or due to deliberate muteness. Determining the failure of a participant whose processing is viable to the success of the transaction is therefore difficult.

Despite the difficulty, the failure of a transaction owner must be determined, as it results in an orphaned transaction that will not be terminated. Orphaned transactions consume resources unnecessarily and leave the involved participants forever in doubt about the outcome of the transaction.

A timeout can be used to detect orphaned transactions. However, this requires careful selection of the timeout value; otherwise, a non-orphaned transaction might be aborted, or, an orphaned transaction might be detected too late.

A heartbeat-based solution in combination with a detached process can be used to more accurately detect and abort orphaned transactions. This approach compares to work on group communication systems and distributed systems protocols for detecting process failures based on heartbeats. In a middleware-mediated transaction environment, using MOM to implement heartbeating enables the natural integration with compensation processes for orphaned transactions.
Solution/Implementation

An application client (transaction owner) regularly sends out heartbeat messages for its transactions to a dedicated queue, the ALIVE.Q. The time interval for sending heartbeats is specified by the application. A detached process (that is part of the transaction monitor) uses the ALIVE.Q queue to receive heartbeat messages. If it does not receive an expected heartbeat for a transaction, the application client process is considered dead and the transaction is orphaned, and compensation is initiated.

Figure 4 (left side) illustrates the heartbeat implementation for the Dependency-Spheres system. Each Dependency-Sphere transaction context (represented by the object DSObject of the D-Sphere system) is a two-threaded process. The main thread follows the application logic of the transaction originator, in which distributed object invocations and distributed messaging operations occur as part of the logical D-Sphere transaction. This thread is also associated with any conventional transaction context of the underlying distributed object transaction service (JTS [18]) that the D-Sphere system uses. The second thread, the heartbeat thread, sends out heartbeat messages to the ALIVE.Q. A HEARTBEAT_INTERVAL time is defined, which describes the cycle after which the heartbeat thread checks the liveness of the main thread and, if the main thread is alive, sends out a heartbeat message to the ALIVE.Q.

Figure 4 (right side) illustrates the detached process for detecting orphaned transactions and for driving compensation, as implemented in the Dependency-Spheres system. This implementation applies the two previously described patterns of "fire-and-remember" and "recoverable state machine".

The detached process (the Compensation Engine component) finds out about all active transactions by browsing the ACTIVE.Q state queue. For each active transaction, corresponding heartbeat messages are expected in the ALIVE.Q. A CHECK_ALIVE_INTERVAL time (that is greater than the HEARTBEAT_INTERVAL time) is specified to set
the time after which the compensation engine should repeat the task of browsing and comparing the ACTIVE.Q entries with heartbeats arriving in the ALIVE.Q. If a heartbeat message is found for an active transaction, the transaction is considered alive. If no heartbeat message is found, the transaction is considered orphaned and the compensation engine moves the transaction message token from the ACTIVE.Q to the ABORTING.Q. This will cause the compensation process for the transaction to be started, which comprises the forwarding of the compensating messages that are stored in the COMP.Q queue.

**Use/Consequences**

This pattern can be combined with the two previously described patterns to implement guaranteed compensation in middleware-mediated transactions. Application and system failures are tolerated, including failures of the compensation engine (the detached process) itself, and the delivery of compensating messages is guaranteed.

A potential concern with this approach is that the message queuing topology in combination with the MOM implementation could interfere with the timely delivery of heartbeat messages. The problem of timely delivery of messages, however, is not unique to this pattern. Further, we believe that appropriate topologies and MOM implementations exist.

4. **Related Work**

The patterns discussed in this paper address the development of middleware system support for transactional enterprise applications. They describe solutions employed in the context of the Dependency-Spheres research system prototype.

The development of system support for middleware-mediated transactions relates to the development of workflow management systems (WFMS) [11] and to the development of system support for other kinds of extended transaction processing models. In comparison to WFMS, our work describes a lower-level middleware-oriented approach to support the focused objective of middleware-mediated transactions; WFMS provide a much broader functionality since their goal is to coordinate users and programs in addition to activities and data. The arguments for using persistent queues and message-oriented middleware to build a workflow management system [10], however, are the same as in our discussion. As Leymann and Roller point out, the use of persistent queues introduces almost for free many features that help improve system and application robustness. In this paper, we introduced three specific message queuing patterns that reinforce these statements and showed how these patterns can be applied to implement guaranteed compensation.

Compensation, as mentioned earlier, is a well-accepted way to deal with failures, and is part of various advanced transaction models of theory and practice [4] [1], and recent proposals of general activity coordination frameworks [9]. Common to all compensation-based approaches is the need to monitor and drive compensation processes reliably until the processes are properly completed. The approach for guaranteed compensation in Dependency-Spheres as introduced in this paper accomplishes this need; compensation is implemented reliably in that different kinds of application and system failures, including failures of the compensation engine itself, are detected and tolerated. To our knowledge, related work on compensation has not reported on how to implement guaranteed compensation, but concen-
trated on other aspects such as formal specification or the construction of compensation graphs for partially executed business processes.

5. Summary

Object-oriented middleware and message-oriented middleware are often used in combination for purposes of enterprise application integration. Consequently, the combination of distributed object transactions and message-oriented transactions into middleware-mediated transactions is requested. In this paper, we described three message queuing patterns that we have found useful for implementing middleware-mediated transactions:

- **Fire-and-remember.** This pattern supports on-abort delivery semantics allowing an application to fire a primary message, create a correlated message (such as a corresponding compensating message) at the same time, and have the middleware remember the correlated message to deliver it in case of a transaction failure.

- **Recoverable state machine.** This pattern represents and executes a fault-tolerant state machine using persistent message queues and MOM transactions. A message is used to represent the entity that is to be monitored (e.g., a transaction, object, system), queues are used to represent entity states, and MOM transactions are used to reliably transition between states.

- **Detached compensation for orphaned transactions.** This pattern describes a heartbeat-based solution in combination with a detached process to detect and drive compensation for orphaned transactions.

These patterns are used in the Dependency-Spheres system prototype to implement guaranteed compensation as a middleware extension. The ideas presented may further help to solve design and implementation problems encountered by MOM applications.

Acknowledgments

We would like to thank Stan Sutton for his useful comments on a previous version of this paper. We would also like to thank Stuart Jones and Stewart Palmer for insightful discussions about message queuing in general.

References


Towards Dynamic Reconfiguration of Distributed Publish-Subscribe Middleware

Gianpaolo Cugola, Gian Pietro Picco
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy.
email: \{cugola,picco\}@elet.polimi.it.

Amy L. Murphy
Dept. of Computer Science
University of Rochester, NY, USA.
email: murphy@cs.rochester.edu

May 4, 2002

Abstract
Publish-subscribe middleware allows the components of a distributed application to subscribe for event notifications and provides the infrastructure enabling event routing from sources to subscribers. This model decouples publishers from subscribers, and in principle makes it amenable to highly dynamic environments. Nevertheless, publish-subscribe systems exploiting a distributed event dispatcher are typically not able to rearrange dynamically their operations to adapt to changes impacting the topology of the dispatching infrastructure.

In this work, we first describe two solutions available in the literature that constitute the extremes of the reconfiguration spectrum in terms of the number of nodes potentially affected by the reconfiguration. They differ essentially in the tradeoffs between simplicity and efficiency. Then, we introduce our contribution as a new algorithm that strikes a balance between the aforementioned solutions by tolerating frequent reconfigurations at the cost of moderate overhead.

1 Introduction
Publish-subscribe middleware are rapidly gaining popularity mostly because the asynchronous, implicit, multi-point, and peer-to-peer communication style they foster is well-suited for many modern distributed computing applications. While the majority of deployed systems is still centralized, commercial and academic efforts are currently focused on achieving better scalability by exploiting a distributed event dispatching architecture.
Beyond scalability, the next challenge for publish-subscribe middleware is dynamic reconfiguration of the topology of the distributed dispatching infrastructure. Companies are frequently undergoing administrative and organizational changes, and so is the logical and physical network enabling their information system. Mobility is increasingly becoming part of mainstream computing. Peer-to-peer networks are defining very fluid application-level networks for information sharing and dissemination. The very characteristics of the publish-subscribe model, most prominently the sharp decoupling between communication parties, make it amenable to these and other highly dynamic environments. However, this can be true in practice only if the publish-subscribe system is itself capable of dealing with reconfiguration. In particular, all the aforementioned sources of reconfiguration affect the topology of the event dispatching network, forcing the middleware to reconfigure its operations accordingly.

The vast majority of currently available publish-subscribe middleware has ignored this problem thus far. With the exception of a few systems adopting a very simple and inefficient solution, none of the proposals in the literature deal with dynamic reconfiguration. In [6], we tackled this problem for the first time by presenting an algorithm that minimizes the number of nodes involved in the reconfiguration. However, this algorithm is mostly suitable for environments where reconfiguration is somehow controlled, or in any case does not occur frequently. In this paper, we present instead a different algorithm that is designed expressly for highly dynamic environments, and that tolerates frequent reconfigurations at the cost of potentially incurring moderate overhead.

The paper is structured as follows. Section 2 provides a concise introduction to publish-subscribe middleware and a discussion about the possible sources of reconfiguration. Section 3 briefly describes a straightforward solution to the problem of dynamic reconfiguration of publish-subscribe systems and the aforementioned solution delimiting reconfiguration. The comparison between the two provides the insight leading to the novel algorithm for highly dynamic environments that constitutes the main contribution of this work, and that is described in Section 4. Related research efforts are discussed in Section 5. Finally, Section 6 draws some conclusions and discusses future avenues of research.

2 Background and Motivation

In this section we provide an overview of publish-subscribe systems, together with a description of the reconfiguration scenarios that motivate our work.

2.1 Publish-Subscribe Systems

Applications exploiting publish-subscribe middleware are organized as a collection of autonomous components, the clients, which interact by publishing events and by subscribing to the classes of events they are interested in. A component of the architecture, the event dispatcher, is responsible for collecting subscriptions and forwarding events to subscribers.

The communication and coordination model that results from this schema is inherently asynchronous; multi-point, because events are sent to all the interested components; anony-
mous, because the publisher need not know the identity of subscribers, and vice versa; implicit, because the set of event recipients is determined by the subscriptions, rather than being explicitly chosen by the sender; and stateless, because events do not persist in the system, rather they are sent only to those components that have subscribed before the event is published.

These characteristics result in a strong decoupling between event publishers and subscribers, which greatly reduces the effort required to modify the application architecture at run-time to cope with different kinds of changes in the external environment.

Given the potential of this paradigm, the last few years have seen the development of a large number of publish-subscribe middleware, which differ along several dimensions\footnote{For more detailed comparisons see [4, 5, 12].}. Two are usually considered fundamental: the expressivity of the subscription language and the architecture of the event dispatcher.

The expressivity of the subscription language draws a line between subject-based systems, where subscriptions identify only classes of events belonging to a given channel or subject, and content-based systems, where subscriptions contain expressions (called event patterns) that allow sophisticated matching on the event content. Our approach is applicable to both classes of systems but in this paper we assume a content-based subscription language, as this represents the most general and challenging case.

The architecture of the event dispatcher can be either centralized or distributed. In this paper, we focus on publish-subscribe middleware with a distributed event dispatcher. In such middleware (see Figure 1) a set of interconnected dispatching servers\footnote{Unless otherwise stated, in the following we will refer to dispatching servers simply as dispatchers, although the latter term refers more precisely to the whole distributed component in charge of dispatching events, rather than to a specific server that is part of it.} cooperate in collecting subscriptions coming from clients and in routing events, with the goals of reducing network load and increasing scalability.

The systems exploiting a distributed dispatcher can be further classified according to the interconnection topology of dispatching servers, and the strategy exploited to route subscriptions and events. In this work we consider a subscription forwarding scheme on a tree topology, as this choice covers the majority of existing systems.

In a subscription forwarding scheme [4], subscriptions are delivered to every dispatcher, along a single tree spanning all dispatchers, and are used to establish the routes that are
followed by published events. When a client issues a subscription, a message containing the event pattern is sent to the dispatcher the client is attached to. There, the event pattern representing the subscription is inserted in a subscription table, together with the identifier of the subscriber. Then, the subscription is propagated by the dispatcher, which now behaves as a subscriber with respect to the rest of the dispatching tree, to all of its neighboring dispatchers. In turn, they record the subscription and re-propagate it towards all their neighboring dispatchers, except for the one that sent it. This scheme is typically optimized by avoiding propagation of subscriptions to the same event pattern in the same direction\(^3\).

The propagation of a subscription effectively sets up a route for events, through the reverse path from the publisher to the subscriber. Requests to unsubscribe from a given event pattern are handled and propagated analogously to subscriptions, although at each hop entries in the subscription table are removed rather than inserted.

Figure 1 shows a dispatching tree where a dispatcher (the dark one) is subscribed\(^4\) to a certain event pattern. The arrows represent the routes laid down according to this subscription, and reflect the content of the subscription tables of each dispatcher. To avoid cluttering the figure, subscriptions are shown only for a single event pattern.

### 2.2 Sources of Dynamic Reconfiguration

Publish-subscribe systems are intrinsically characterized by a high degree of reconfiguration, determined by their very operation. For instance, routes for events are continuously created and removed across the tree of dispatchers as clients subscribe and unsubscribe to and from events. Clearly, this is not the kind of reconfiguration we are investigating here. Instead, the dynamic reconfiguration we address can be defined informally as the ability to rearrange the routes traversed by events in response to changes in the topology of the network of dispatchers, and to do this without interrupting the normal system operation.

Triggers for such a reconfiguration are many, with the effect being the disappearance of one or more links between dispatchers, and possibly the appearance of new ones. A link can disappear either because it is being explicitly removed at the application layer, or because the underlying communication layers are no longer capable of ensuring communication between the two nodes.

The first case is clearly the most controlled one. As an example of this case, the publish-subscribe systems deployed in enterprise usually rely on a backbone of interconnected dispatchers. A system administrator may need to substitute one link with another to change the topology of the event dispatcher, e.g., to balance the traffic load or to adapt to a change in the underlying physical network. The result of such an operation should be an automatic reconfiguration of the distributed dispatcher to adapt event routes to the new topology.

Unfortunately, the sources of reconfiguration are not always under the control of applications. A dispatcher may become disconnected from one of its neighbors because the link

---

\(^3\)Other optimizations are possible, e.g., by defining a notion of “coverage” among subscriptions, or by aggregating them, like in [4].

\(^4\)More precisely, only clients can be subscribers. With some stretch of terminology, here and in the following we will say that a dispatcher is a subscriber if it has at least one client that is a subscriber.
connecting the two has failed. Mobile computing defines a scenario where this is particularly likely to happen. Mobility undermines the assumptions traditionally made in distributed systems by enabling the network topology to change dynamically as the mobile hosts move and yet remain connected through wireless links. This is brought to an extreme by mobile ad hoc networks (MANETs) [10], where the networking infrastructure is totally absent and physical links come and go according to the distance between hosts. In all these cases, lack of communication with a dispatcher results in the inability to route subscriptions and events towards it, due to the partitioning of the dispatching tree. A reconfiguration process is needed not only to restore the tree connectivity, but also to properly rearrange the routing information on the tree.

A somehow intermediate scenario is provided by peer-to-peer systems. In fact, the ability to perform scalable content-based event routing provided by distributed publish-subscribe middleware can be exploited to implement data sharing applications based on a peer-to-peer architecture. This idea has been exploited in PeerWare [7], a peer-to-peer middleware developed in the context of the EU project MOTION\(^5\), and is also described in [9]. In this setting, each peer node plays the same role of a dispatcher in a publish-subscribe middleware, contributing to message routing. Consequently, the underlying routing mechanism must be able to cope with frequent changes of the topology of the peer network, determined by users (and hence peers) joining and leaving the peer-to-peer system.

### 3 Reconfiguration Extremes

In this paper, we focus on reconfigurations that involve the removal of a link and the insertion of a new one, thus keeping the dispatching tree connected. Issues of the loss of a dispatching node are more complex because the dispatching tree is partitioned into more than two pieces. We will consider this in future work. Simpler reconfigurations, involving only link removal or insertion, can be dealt with using plain subscriptions and unsubscriptions, as we describe later on.

The problem of dynamically reconfiguring a publish-subscribe system can then be seen as composed of three subproblems. The first problem is to manage the reconfiguration of the dispatching tree itself, retaining connectivity among dispatchers without creating loops. The second problem is to reconfigure the subscription information held by each dispatcher, keeping it consistent with the changes in the reconfigured tree and without interfering with the normal processing of subscriptions and unsubscriptions. The third problem is to minimize the loss of events during the reconfiguration.

In this paper, we focus on correctly reconfiguring the subscription information, i.e., on the second of the aforementioned problems. We assume that the underlying tree is somehow reconfigured, and we tolerate (minor) event losses. The rationale for this choice lies in the fact that the consistency of the subscription information is key for the correct functioning of a publish-subscribe system, and hence also for limiting the number of events lost. Moreover, the algorithms for keeping the underlying tree connected strongly depend on the specific

reconfiguration scenario, and in any case some existing solutions are likely to be adaptable, as we briefly discuss in Section 4.3. Also, by operating in a dynamic environment, the applications we consider must tradeoff some degree of reliable delivery. It is possible to extend the solution presented here to incorporate some fault tolerant techniques, but we leave this for future research.

Under these premises, a simple and reasonable way to compare the effectiveness of different reconfiguration algorithms is to consider the number of dispatchers involved in the reconfiguration, i.e., the number of dispatchers whose routing tables are changed during the reconfiguration process. Intuitively, the smaller this number the less the reconfiguration interferes with the system. Hence, not only is the overhead reduced, but so is the disruption of event routes, and consequently the likelihood of an event loss. If we base our comparison only on this value, two approaches represent the extremes of a wide spectrum: a straightforward algorithm that attacks the reconfiguration problem using the same strategy adopted when the tree of dispatchers must be split in two subtrees or when two subtrees must be joined, and a more efficient algorithm that minimizes the number of dispatchers involved. The remainder of this section describes these solutions and compares them. This comparison helps us gather some observations that motivate the need for a different approach when the target scenario exhibits high dynamicity. A description of an algorithm tailored for such environments is given in Section 4, which represents the main contribution of the paper.

3.1 A Straightforward Approach

In principle, the removal of an existing link and the insertion of a new one can be carried out by using exclusively the primitives available in a publish-subscribe system.

The reconfiguration triggered by a link removal can be dealt with by using unsubscriptions. When a link is removed, each of its end-points is no longer able to route events matching subscriptions issued by dispatchers on the other side of the tree. Hence, each of the end-points should behave as if it had received from the other end-point an unsubscripton for each of the event patterns the latter was subscribed to. The insertion of a new link triggers a similar process that uses subscriptions to reconfigure the routing.

This approach is the most natural and convenient when reconfiguration involves only either the insertion or the removal of a link, and is actually adopted by some publish-subscribe middleware. On the other hand, when it is necessary to replace a link with a new one, thus effectively reconfiguring the topology of the tree while keeping the same set of nodes as dispatchers, this strategy leads to results that are far from optimal. In fact, if the route reconfigurations caused by link removal and insertion are allowed to propagate concurrently, they may lead to the dissemination of subscriptions which are removed shortly after, or to the removal of subscriptions that are then subsequently restored, thus wasting a lot of messages and potentially causing far reaching and long lasting disruption of communication.

Figure 2 illustrates this concept on the dispatching tree of Figure 1. According to the straightforward mechanism, when the link between $A$ and $B$ is removed the two end-points trigger unsubscriptions in their subtrees, without taking into account the fact that a new link has been found between $C$ and $D$. Depending on the speed of the route destruction and con-
striction processes, subscriptions in $B$’s subtree may be completely eliminated, since there are no subscribers in that tree. Nevertheless, shortly afterwards most of these subscriptions will be rebuilt by the reconfiguration process. The resulting reconfiguration of subscription information is not only inefficient, but it may greatly increase the loss of events.

The drawbacks of this approach are essentially caused by a single problem: the propagation of reconfiguration messages reaches areas of the dispatching tree that are far from the ones directly involved in the topology change, and which should not be affected at all. This observation leads to the idea of delimiting the area involved in the reconfiguration, a key element of the approach described in the next section.

### 3.2 A More Efficient Approach

To identify the minimal set of dispatchers affected by a link removal followed by a link insertion, we observe that each dispatcher routes events and subscriptions based on the local knowledge gathered from its neighbors. Similarly, its actions are limited to messages sent to its immediate neighbors. In other words, each dispatcher has knowledge only about its immediate “next hops”. In [6], these considerations lead to the definition of the reconfiguration path as the only portion of the dispatching tree affected by the reconfiguration. The reconfiguration path includes the two end-points of the removed link and all the dispatchers connected to them through the new link.

If we consider the example of reconfiguration in Figure 2, where the link $(A, B)$ is being replaced with the link $(C, D)$, the reconfiguration path is $(A, E, F, C, D, B)$. From the above considerations about the way subscription forwarding publish-subscribe systems work, it is easy to understand that dispatchers that do not belong to the reconfiguration path will not experience any change in their subscription tables. They will continue forwarding events the same way they were doing before. As an example, before reconfiguration (see Figure 1) dispatcher $X$ was sending events to $B$, which was forwarding them to $A$ to reach the subscriber $S$. After reconfiguration (see Figure 2), $X$ continues sending events to $B$ even though now $B$ forwards them to $D$ to reach the same subscriber $S$. $X$ has no knowledge of the fact that $B$’s routing table has changed.

An algorithm that leverages off of this concept of reconfiguration path is described in [6]. Its processing starts from one of the two end-points of the removed link and uses a special
source routed message that moves from dispatcher to dispatcher along the reconfiguration path, changing the routing tables according to the new topology.

3.3 Comparison

A first comparison of the two approaches described above, based only on the number of dispatchers involved in the reconfiguration, could lead to the conclusion that the second solution is always to be preferred over the first one.

Nevertheless, it turns out that the correctness of the straightforward solution is not affected by multiple reconfigurations occurring in parallel. More precisely, if during a reconfiguration another link break occurs the two reconfigurations may proceed in parallel without influencing each other. Indeed, since the reconfiguration mechanism adopts only standard subscriptions and unsubscriptions and it does not affect the correct propagation of subscription and unsubscription messages, the overall reconfiguration process will complete correctly, independent from the number of link replacements involved.

Unfortunately, the same consideration does not hold for the algorithm described in [6] that, by rearranging the subscription information while unfolding along the reconfiguration path, strongly relies on its connectivity. As a result, this approach is quite sensitive to multiple reconfigurations. In particular, when different reconfiguration paths have one or more links in common or when an additional link break does not allow a running reconfiguration process to complete as expected, special mechanisms must be put in place to guarantee the correctness of the overall reconfiguration process. Currently, these mechanisms are still under investigation, and hence the applicability of the approach covers only controlled environments where requests for multiple reconfigurations can be serialized and answered in sequence.

The above considerations motivate the need of a different algorithm for very dynamic environments such as MANETs or peer-to-peer networks, in which multiple reconfigurations occurring in parallel are more the rule than the exception. This algorithm should try to balance the performance, in terms of the set of dispatchers involved, of the solution described in [6] with the better resilience to multiple reconfigurations characterizing the straightforward solution. The next section describes our proposal for such an algorithm.

4 Striking a Balance

To design a new algorithm for highly dynamic environments, we begin by observing that the drawbacks of the straightforward algorithm described in Section 3.1 mainly result from the fact that the unsubscription process determined by a link removal and the subscription process taking care of a link insertion proceed completely in parallel, while some coordination would likely minimize the traffic. This consideration leads to the idea of identifying the impact of subscriptions and unsubscriptions on an already established tree to determine if

---

6Here we assume that the process keeping the tree of dispatchers connected is capable of correctly handling multiple reconfigurations in parallel without introducing loops among the dispatchers and without resulting in partitioned trees.
some kind of synchronization could improve the performance of the straightforward algorithm without sacrificing consistency when multiple link breaks occur in parallel.

4.1 Identifying the Tradeoffs

To describe the impact of subscriptions and unsubscriptions on a publish-subscribe system that adopts a subscription forwarding strategy, it is useful to classify dispatchers into subscribers, forwarders, and splitters. For each event pattern $p$, a subscriber is a dispatcher that has at least one client subscribed to $p$. A forwarder is a dispatcher which is not a subscriber and whose routing table has a single entry tagged with $p$ (i.e., graphically this means that it has a single outgoing arrow labelled with $p$). Finally, a splitter is either a dispatcher whose routing table has two or more entries tagged with $p$, or a subscriber.

With these definitions in mind, we can derive the following general rule for systems based on the subscription forwarding strategy described in Section 3.1: a subscription issued by a client is propagated in the dispatching tree only up to the closest splitter, if it exists; to the whole tree, otherwise. Clearly, in the special case where the new subscriber is also a splitter nothing happens.

To understand this rule we observe that, for each event pattern $p$, there exists a minimal spanning tree containing all the dispatchers subscribed to $p$. For instance, in Figure 3 this minimal spanning tree is composed of dispatchers $A$, $B$, $C$, $D$, $E$. The routing tables of the dispatchers belonging to this subtree are organized in such a way that events matching $p$ reaching one of them are forwarded to all the others. Moreover, the routing tables of all the other dispatchers route events matching $p$ to this subtree but not viceversa, i.e., events reaching this subtree are not forwarded outside of it. Hence, we observe that the point of attachment for a new subscriber to such minimal subtree is constituted by the closest splitter.

With reference to Figure 3, for the new subscriber $S$ to join the subtree, only the routing tables of all the dispatchers along the path from $S$ to the subtree ($F$ and $B$ in the figure) must be changed. A similar rule holds for unsubscriptions, which propagate up to the first splitter that remains such even after it has rearranged its subscription table by processing the unsubscription message.

---

7As already mentioned, these definitions do not take into account optimizations based on the notion of “coverage” among subscriptions, although they could be generalized to do so. Instead, the definitions are based on the usual optimization of avoiding to forward a subscription already present in the system.
From these rules it is possible to derive two considerations. First, the price that must be paid for adding a subscription is limited. In general, it does not involve a propagation along the entire tree but only along the route to the closest splitter, unless there are no subscribers. Second, the more splitters that exist the shorter the path that a subscription must follow. These considerations lead to the idea of an algorithm that behaves like the straightforward one but performs the subscription step before unsubscribing. This way, the tree is kept “dense” of subscriptions, thus reducing the overhead caused by the propagation of subscriptions. It is true that this strategy may add subscriptions that must be removed immediately after, but in any case these subscriptions will propagate only up to the first splitter. Moreover, this solution has the beneficial side effect of minimizing the disruption of event routes, by minimizing the likelihood that a subscription is removed only to be restored shortly after. The next section describes such an algorithm in detail.

4.2 Rearranging Subscription Tables

In the following, we assume that the links connecting the dispatchers are FIFO and transport reliably (i.e., with no loss) subscriptions, unsubscriptions, events, and other control messages. Both assumptions are typical of mainstream publish-subscribe systems, and are easily satisfied by using TCP for communication between dispatchers.

The operation of the algorithm starts when a broken link between two dispatchers is detected. The actual processing is triggered by one of the two end-points, called the initiator, chosen according to some ordering criteria. The initiator starts a tree reconstruction process that tries to reconnect the tree without creating loops. For the moment, we gloss over the details of how the new link is identified and assume that this information becomes somehow available to the initiator after a given delay. We provide details about how this can be accomplished in reality at the end of this section. Here we focus on the processing needed to reconfigure the information for routing events over the reconnected tree.

The algorithm unfolds as follows:

1. When the end-points of a link detect that it is broken, they both start a timer $T$. In addition, the initiator starts the tree reconstruction process.

2. After the initiator (e.g., $A$ in Figure 2) has determined the new link that needs to be established in order to reconnect the tree, it sends an OPENLINK message to the end-point of such link belonging to the same semitree as the initiator ($C$ in Figure 2). The OPENLINK message is sent using a unicast channel that does not follow the dispatching tree, and must be acknowledged by the recipient using the same “out of band” channel. OPENLINK contains a reconfiguration identifier $recID$, which distinguishes it among multiple, concurrent reconfiguration processes. For each link, the value of $recID$ is determined by the end-points when the link is first established successfully. Hence, the value of $recID$ associated to the link is known to both end-points when reconfiguration actually occurs.

3. Upon receiving the OPENLINK message, the end-point of the new link belonging to the initiator semitree opens the new link and forwards the OPENLINK to the other
end-point. Each end-point behaves as in a merge of the two semitrees, as described in Section 3.1, by exchanging their subscriptions over the new link, unless the old and new link share an end-point. In this case, subscriptions that exist only towards the other end-point of the new link are not forwarded. Moreover, immediately after the exchange, they start the propagation throughout their semitree of a flush message containing the recID originally contained in openlink.

4. At each dispatcher in each of the semitrees, subscriptions are propagated according to the normal processing. According to the discussion in the previous section, they propagate only up to the first splitter. Instead, the flush message is broadcasted to all the dispatchers in the semitree.8

5. If the flush message reaches the end-points of the broken link or the timeout T expires, whichever occurs first, each of the end-points behaves as during a partition, by starting an unsubscription process for all subscriptions that came originally from the other end-point of the vanished link. In the case where the timeout expires, the corresponding recID is temporarily saved, to allow discarding of delayed flush messages.

As we discussed before, this algorithm makes sure that subscriptions rerouting events through the new link are laid down before the obsolete subscriptions that served the only purpose to route events through the vanished link are removed. The openlink message is essentially used to activate the new link and trigger the spreading of subscriptions between the semitrees. Instead, the flush message is used to notify the end-points of the vanished link that it is now safe to remove unnecessary subscriptions. This property is ensured by the fact that the new subscriptions propagate ahead of the flush message in FIFO channels. Essentially, openlink triggers the portion of the reconfiguration taking care of merging the semitrees through the new link, while flush triggers the partitioning of the semitrees across the vanishing link.

Clearly, in a highly dynamic environment connectivity may change during a reconfiguration, e.g., by causing multiple, concurrent link breaks. This does not constitute a problem if the reconfiguration paths determined by the breaks are not overlapping, in that reconfiguration can proceed independently. If instead the reconfiguration paths are overlapping, an additional link break may determine a temporary inability to communicate between the initiator and the end-point of the new link until a new tree reconstruction process has completed. Effectively, the second reconfiguration is “nested” in the first one, which cannot complete until the second is over. Besides increasing the overall time needed to recover from the first break, this situation may lead to a delay, if not a loss, of the flush message.

This situation is handled by the last step of the algorithm, that performs the same action no matter whether a flush message has been received, delayed or lost. Interestingly, the effect of such action in these situations is different, and is determined by the configuration.

---

8Ideally, the flush message needs to propagate only along the reconfiguration path, up to the end-points of the vanished link. This can be accomplished if the tree reconstruction process provides information about the reconfiguration path. However, broadcasting along the whole tree is more resilient to concurrent reconstructions.
of subscriptions that have already been laid out. When a flush message is received, the corresponding new subscriptions are already setup correctly. Hence, the unsubscription process will remove only unnecessary subscriptions, along the reconfiguration path. On the other hand, if the timeout has expired two cases are possible. In the first case, no route reconnecting the tree exists. Hence, the unsubscriptions will rightfully propagate throughout the tree and possibly outside the reconfiguration path, up to the first splitter. In the case where the flush message has been simply delayed, some overhead will result, depending on how fast the subscriptions ahead of the flush message have travelled, with respect to the unsubscriptions triggered by the expiration of the timeout.

This latter aspect of the algorithm is controlled by the value of the timeout $T$. If $T$ is too small, an unnecessary unsubscription process is likely to be triggered while the flush is still on its way. On the other hand, if $T$ is too large superfluous subscriptions are retained for a longer time, steering events towards dead branches of the tree. Essentially, the value of $T$ must be chosen by evaluating a tradeoff between the responsiveness of reconfiguration and the bandwidth overhead caused by superfluous subscriptions. We are currently developing simulations to determine reasonable values for $T$ and to investigate how they impact performance.

Notably, the reconfiguration described by this algorithm does not interfere with the normal processing of events and (un)subscriptions. In the solution we describe here, we are not trying to enforce any custom, source routed processing of messages like in the solution we described concisely in Section 3.2. Instead, we are relying on the standard processing that, by design, deals with the concurrent publishing of events and issuing of (un)subscriptions. We simply intervene in the timing when these operations are triggered to deal with reconfiguration. The only addition is the presence of a flush message that, however, does not impact the normal processing.

Finally, our algorithm intuitively loses fewer events than the straightforward solution. In fact, in the case where the flush message is correctly received by the initiator, the routes for events are never disrupted. The only events lost are those that reached the end-points of the vanished link before the subscriptions exchanged through the new link. Instead, the straightforward solution may lose events in areas potentially very far from the one where reconfiguration is occurring (i.e., from the reconfiguration path), since the uncoordinated propagation of subscriptions and unsubscriptions may temporarily remove routes. In the cases where the timeout expires and the unsubscription process is triggered, the amount of events lost is intuitively in between these two extremes.

### 4.3 Keeping the Tree Connected

Thus far, we focused only on how to update the routing information on the dispatching tree, without considering how a broken link is detected and a new route, involving a new link, is determined. In this section we hint at some ways of providing this functionality.

**Detecting a Broken Link**  If the links between the nodes of the tree are actually mapped directly on physical communication links between the nodes, then detecting a link break can
be dealt with in the same way as routing protocols for MANETs (e.g., DSR [2] or AODV [11]): essentially using MAC-level or application-level beaconing. A beacon is a packet that is periodically broadcasted with a time-to-live of 1, and hence reaches only the stations that are physically in communication range. When a station no longer detects a beacon\(^9\) from another station, the link between the two can be considered broken. A similar approach can be adopted both in wired networks and when the logical link to be monitored does not map directly to a single physical link. In these cases a special point-to-point protocol, e.g., ICMP, must be used to implement the beaconing mechanism.

This proactive approach, however, constantly monitors the network. An alternative, lazier approach can detect link breakages only when a communication failure is notified at the application level, e.g., by an error returned while transmitting data on a socket. Clearly, this is possible only if the underlying transport protocol is reliable.

**Replacing a Broken Link With a New Route**  After a broken link is detected, a new one must be found to reconnect the two partitioned subtrees without creating loops. The initiator must request a new route to its neighbors; new routes must be computed, possibly in a distributed way; they must be delivered back to the initiator, which will select one. A number of mechanisms can be used for this purpose.

For instance, it is reasonable to assume that each dispatcher maintains a cache of the network addresses of the dispatchers connected to its neighbors (i.e., each dispatcher has a partial visibility of the system topology). When a link vanishes, the initiator can send a message containing the list of dispatchers known to be part of the disconnected subtree, that gets propagated along the tree up to a certain number of hops. Each dispatcher receiving this message can then determine if it can reach one of the dispatchers on the list and how far it is, and send back a reply containing this information. The initiator uses the information to select the best route. The goal behind this process is clearly to keep the topology of the logical network of dispatchers as close as possible to the topology of the underlying physical network. In alternative, existing mechanisms for maintaining multicast trees can be used. For instance, for MANETs the strategy adopted by MAODV [13], heavily based on network-level broadcast and propagation of route requests, can be applied or adapted to our needs.

Thus far, we assumed that only a single link is added. This is reasonable in wired networks, where the routing infrastructure hides the details of communication between dispatchers. However, this may not hold true in a MANET or whenever the dispatching network is mapped directly on the network topology. In this case, one link is often not sufficient to reconnect the two partitioned subtrees, and additional intermediate nodes are needed. The new link can then be stretched into a sequence of nodes, whose end-points constitute the end-points of what we considered thus far as the new link.

\(^9\)Typically, a \textit{k-out-of-n} policy is adopted, to avoid rapid fluctuations in connectivity.
5 Related Work

Most publish-subscribe middleware are targeted to local area networks and adopt a single, centralized dispatcher. In recent years, the problem of wide-area event notification has attracted the attention of researchers [16] and some systems have been presented, which adopt a distributed dispatcher, such as TIBCO’s TIB/Rendezvous, Jedi [5], Siena [4], READY [8], Keryx [17], Gryphon [1], and Elvin4 [14] in its federated incarnation.

To the best of our knowledge, none of these systems provide any special mechanism to support the kind of reconfiguration proposed in this paper. Siena [4] and the system described in [18] adopt the straightforward solution we describe later in Section 3.1 to allow subtrees of dispatchers to be merged or trees to be split. Jedi [5] provides a different form of reconfiguration that allows only clients (not dispatchers) to be added, removed, or moved to a different dispatcher at run-time. A similar capability has been conceived also for Elvin [15], that supports mobile clients through a proxy server, although this feature is not included in the latest (4.0.3) release.

Finally, some research projects, like IBM Gryphon [1] and Microsoft Herald [3], claim to support a notion of reconfiguration similar to the one we address in this work, but we were unable to find any public documentation about existing results.

6 Conclusions and Future Work

Currently available publish-subscribe systems adopting a distributed event dispatcher do not provide any special mechanism to support the dynamic reconfiguration of the topology of the dispatching infrastructure to cope with changes in the external environment. Solutions available in the literature at best exploit a straightforward solution whose simplicity is often outweighed by its inefficiency, since it involves areas that should not be affected by reconfiguration. Previous work by the authors has shown instead that there is a way to constrain reconfiguration, at the cost of increased complexity and poor tolerance to frequent topological changes.

In this work, we presented a solution that strikes a balance between these two reconfiguration extremes, by tolerating frequent reconfigurations at the cost of moderate overhead. Essentially, a mechanism is provided to ensure that the new routes caused by reconfiguration are laid down before the obsolete ones are removed. Besides optimizing the reconfiguration of routing information, this approach is also intuitively better at delivering events during reconfiguration.

Future work will investigate quantitatively the benefits of this solution against the other ones we described in this paper, using a simulation approach. Moreover, we will verify the feasibility of our approach by implementing a prototype and validating in the field.

References


205
Active replication of software components

Juan Pavón, Luis M. Peña‡‡
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
email: jpavon@sip.ucm.es, luismpena@yahoo.com

Abstract
This paper considers active replication of distributed objects over CORBA and Java RMI. It describes a replication model and tools whose main purpose is the simplification of the design and implementation of applications with replicated components that inter-communicate to collaborate on a task or to maintain their consistency with client requests. The starting point of this work is Sensei, a group communications system that supports the replication of components as groups of objects working under a virtual synchronous model. It describes the requirements to support the component abstraction, as a key concept to facilitate the development of fault tolerant applications. This model is compared with the replication model defined by OMG for a fault tolerant service in CORBA.

1. Introduction
The replication of application components [1] over different hosts is a common practice to achieve fault tolerance. Usually, the development of actively replicated applications is carried out under the virtual synchronous model [2]; this model provides an abstraction where the replicas are associated in dynamic groups and where every replica observes the same communications in the same order §§. Under these conditions, the replicas reach the same final state without the strong communication efforts that a transaction-based system would require, maintaining consistency among these replicas.

The virtual synchronous model is defined over reliable multicast primitives. Above these primitives, it is possible to implement serialization patterns [3] to obtain type-safe communications, and wrappers supporting the object orientation abstraction. A recent example of the application of these concepts is the CORBA fault tolerance service [4] when supporting active replication. This active replication is built on top of a virtual synchronous communication system and it automatically multicasts any request to every replica. This allows for a very simple design, as each replica can be programmed as a standalone object, with the only requirement being that every request must be deterministically driven: replicas independently process the same requests, continuously sharing a common state.

‡‡ This work is sponsored by the Spanish Committee for Science and Technology (TIC2000-0737-C03-02).
§§ It is not necessary for the order to be identical; in fact, the model precludes the use of messages with causal order, where concurrent communications can be processed in a different order, and still reach the same final state.
Nevertheless, several reasons which are subsequently explained in depth, like its replication model or its state transfer support, make us think that the CORBA fault tolerant service is mainly focused on passive replication.

This paper presents a model of active replication, which is supported by a set of tools (Sensei) that simplify the design and implementation of replicas that inter-communicate to collaborate on a task or to maintain their consistency with a client’s request. Our starting point is the definition of a group communication system with an object-oriented interface, and support of typed communications. Our first approach was to include an object-oriented interface and a serialization pattern on top of Ensemble*** [5], a well-known group communication system. Later, we developed our own communication system, called SenseiGMS†††, directly implementing an object-oriented interface with typed messaging.

The rest of this paper is structured as follows. The second section introduces our group communication system, SenseiGMS. The third describes problems with the active replication support in CORBA. The fourth section studies the use of replicated components, and the logic required to support them. The fifth section shows the state transfer mechanisms and the sixth is focused on the use of transactions.

2. **SenseiGMS**

SenseiGMS is a group communication system accessible via CORBA [6] and JavaRMI [7] interfaces, and supporting reliable multicast primitives that are built directly over CORBA and RMI. The multicast communications subsystem can be changed with Ensemble (although in that case it loses the support for JavaRMI).

Replicas must implement the interface `GroupMember`, and they can join/leave a group or send reliable multicast communication to other replicas using the interface `GroupHandler`. During their lifetime in the group, members receive views, structures containing information about group composition. Messages sent between members must be defined by the application; using RMI, messages are defined as `java.util.Serializable` instances, while in CORBA messages are defined as `valuetypes`; in this way, inheritance is allowed and the access is local on each replica that receives the message. Nevertheless, it is not mandatory to use typed messages; generic non-typed messages can be used as well. The basic definition of these interfaces using CORBA IDL is the following:

```plaintext
typedef long GroupMemberId;

valuetype Message {};

interface GroupHandler{
    GroupMemberId getGroupMemberId();
    boolean isValidGroup();
    boolean leaveGroup();
    boolean castMessage(in Message msg);
    boolean sendMessage(in GroupMemberId target,
```
A replicated server defines its IDL or Java interface inherited from `GroupMember`; for example, a simple replica that generates sequential numbers can be specified in Java as:

```java
interface NumberGenerator extends GroupMember {
    public int getNumber() throws RemoteException;
};
```

If this replica requires a message to be sent containing the last generated number, it could define a message that in CORBA would be:

```corba
valuetype NumberGeneratorMessage : Message {
    public long number;
};
```

### 3. The CORBA Fault Tolerant Service

The CORBA model simplifies the deployment of replicated applications, as it isolates any replicated behavior from the application. Figure 1 shows this model, where any request is intercepted by a virtual synchronous communication system that multicasts the request to each replica, which is programmed as a non-replicated server.

![Fig. 1. The CORBA active replication model, with every client request automatically sent to every replica. The service must ensure the proper sequential ordering on every request, discarding duplicated requests also. Implementing the virtual synchronous communication system below the ORB improves performance.](image-url)
The CORBA Fault Tolerance Service imposes a common model for active and backup replication. Although it unifies the design of replicated applications, we believe that its features are not fully suitable for those requiring active replication, as we explain below.

In the CORBA active replication model, every request is automatically multicast to every replica in the group. Group communications are expensive, requiring more resources and taking longer than the equivalent point-to-point communications, and therefore they should be minimized. Queries that do not update the replicas do not need to be serviced by all the replicas, but this optimization is not possible with the current model, at least when using the CORBA specification exclusively.

A possibility would be to define queries as attributes in the IDL definition, but even this approach does not forbid a specific implementation to update its state when that attribute is read.

The state transfer supported by the service is based on logging mechanisms. The service infrastructure asks replicas periodically for their states, which are stored together with the messages processed afterwards, in order to be able to build the final state using only those logs. This mechanism is perfectly suitable for passive replication, but imposes a penalty under active replication, as replicas are continuously requested to give their state. Note that the actively replicated application is not obliged to use the logging mechanisms, but then it has to implement its own state transfer.

As stated in the specification, the service is not valid in case of non-deterministic behaviors, but this is not an uncommon requirement. For example, the previous section showed an interface for a server giving unique numbers. If its specification is to give a random and still unique number, it could not be deployed using the CORBA service, as each replica would likely return a different number and share an inconsistent state.

This problem could be solved using an external service such as the random number generator, but by logic this service could not be replicated itself, therefore presenting a single point of failure in the design. Additionally, while the CORBA service specifies that duplicated requests to the group are discarded, it does not specify that the requests made by the members of a group receive the same treatment. That is, each member would access the random generator service independently, receiving different numbers and leaving the problem still unsolved.

What is more important, this problem is not specific to this non-deterministic case: an access to any external component is performed by all the replicas. This requires more resources, but it can be a problem in itself. For example, if the external component has a specific cost per access, the group should avoid accessing it many times, but this is not possible under normal circumstances. Even if the service could take the requests made by the group, it could still not take the non-CORBA requests such as, for example, those done via http. A specific implementation can override this problem under special circumstances, but under the current specification, the use of components is, at least, problematic.

While this paper is not intended to be a criticism of the CORBA service, these issues lead us to believe that the CORBA fault tolerant service is mainly focused on passive replication. Note that this focus makes sense, as most dependable systems are more
effectively modeled using passive replication. The main problem we associate with the current model is the use of the whole server as the replication unit, something completely necessary in the case of passive replication. Using active replication, a finer granularity will produce better results, giving way to a collaborative approach among the replicas to achieve a better performance. The following section is centered on the use of components such as the replication unit.

4. Replicated components

A valid implementation for the NumberGenerator interface shown in the second section is as follows. Each replica stores the last number generated by the group. When a replica receives a request, it sends a message to the group, and every replica then increments the stored number. Finally, the replica that received the request returns the number generated to its client.

If the number returned must be random, every replica is programmed to contain a set with free numbers. The replica that receives the request generates first a random number, which is included on the message that is sent to the group. On reception, each replica removes from its list the first free number following the one received on the message.

Let’s consider a different solution, less optimal but far more generic, based on replicated components. In this solution, servers do not communicate with each other, but share a set container with the generated numbers. When a server has to generate a new number, it obtains first a random number and then accesses the shared set to find the first free number after the generated one. This free number is then returned to the client and marked as generated in the set. Using a bitset to implement the container, the code is written as follows:

```java
number = randomNumber;
while (replicatedBitSet.get(number)) {
    number++;
}
replicatedBitSet.set(number);
```

The interesting point about this approach is that the code is similar to the code written for a standalone server, it only differs on the component used that must now include the replication logic. This replication logic is that shifted from the server to the containers, which may be available as generic components.

The previous code is obviously unsafe. It is not thread-safe and it is therefore not process-safe: two servers running on different processes could generate the same number, violating their specifications. On a multithreaded environment, this problem is solved using monitors, which provides the solution to this case: the need to have replicated monitors, changing the previous code to:

```java
replicatedMonitor.lock();
while (replicatedBitSet.get(number)) {
    number++;
}
```
replicatedBitSet.set(number);
replicatedMonitor.unlock();

Replicated monitors are reentrant; note however that a monitor is owned by a replica, and therefore two threads on the same replica cannot be synchronized with a replicated monitor.

The virtual synchronous model defines an entity called Group Membership Service (GMS) that allows replicas to join existing groups, creating them if they do not yet exist. As a result, the replica would eventually receive an event from the GMS accepting it into the group. This service must detect failures on replicas to exclude them from the group. This detection is based on unreliable failure detectors [8], which means that a replica can be mistakenly expelled from the group. Each component on an application must request its entrance on its own group, and the exclusion of a component will likely mean the inability of the server to service requests. In principle, it is not possible to assure that the failure detector will exclude a component but not others in the same host, and therefore some coordination is needed.

Sensei addresses this problem by defining *domains*, which are just *supergroups* where components are registered. Domains also handle dynamic components: a group member could create a component at runtime, and the component is automatically replicated over the other members in the group. This feature is especially interesting to enforce the abstraction of replicated components as simple *shared* components. For example, if the NumberGenerator has to give unique numbers on a per-group base, where a group is only specified by a string, a replicated *map* is needed, associating a replicated *bitset* to each group:

```java
ReplicatedBitSet group = replicatedMap.get(groupName);
if (group == null) {
    group = new ReplicatedBitSet(domain);
    replicatedMap.put(groupName, group);
}
while (group.get(number)) {
    number++;
}
group.set(number);
```

When a replica creates a bitset, it is automatically propagated to the other replicas that create their own bitset instance, with the same state.

There is a problem with the previous code. A replica can fall down while it is being executed, leaving the group in an inconsistent state. It is necessary to have a mechanism to discard the changes if the whole process is not terminated properly. Although we have used the term *transaction* to name this mechanism, the concept is slightly different from the traditional one used on databases. Here it does not protect against concurrent changes, just against uncompleted updates due to server crashes:

---

‡‡‡ This concept is not related to the *domains* concept on the CORBA Fault Tolerant specification, where domains are used to allow applications to scale to arbitrary sizes.
domain.startTransaction(replicatedMap);
ReplicatedBitSet group = replicatedMap.get(groupName);
if (group == null){
    group = new ReplicatedBitSet(domain);
    replicatedMap.put(groupName, group);
}
domain.endTransaction(replicatedMap);
group.lock();
while (group.get(number)) {
    number++;
}
group.set(number);
group.unlock();

A transaction is made around a replicated monitor, locking it first. This, in effect, protects against concurrent changes on different replicas, but two threads on a replica could still perform concurrent updates (replicated monitors do not synchronize threads), violating the isolation property of transactions. Additionally, rollbacks are not allowed, but transactions can be nested.

The relationship between transactions and the virtual synchrony model has been extensively studied [9, 10, 11, 12, 13, 14], as both are solutions to the same consensus problem. Although it is not possible to consider these transactions as full featured transactions, they still share some of the properties required by distributed transactions, like the atomicity. The implementation is based on delaying group communications during transactions, and the other replicas only receive them when the transaction has been completed. This assures that, if the member falls down, the other replicas perceive no changes at all. The transaction atomicity is provided by sending communications in one multicast message [11]. Nevertheless, the replica that processes the transaction must observe the changes, to avoid write-read dependencies; that is, this replica receives and processes the messages that it sends, but the other replicas only receive these messages when the transaction has been completed.

The implications of this behavior on the virtual synchrony model are dealt with in a subsequent section. The concept that we wish to highlight in this section is the possibility of working with replicated components, which encapsulate all the replication behaviour, and therefore the application designer can focus on the application logic itself, using algorithms that are very similar to those employed on standalone applications.

The use of replicated components is enabled in Sensei via SenseiDomains, which implements the concepts shown in this section: domains, monitors, transactions, as well as the state transfer mechanisms described in the following section. The implementation is done on top of SenseiGMS; Sensei also defines replicated components for the most used containers, based on the well-known java.util package.

5. State transfer

The use of defined components as replication units is convenient as it makes the development of replicated solutions easier. These components should be designed to minimize group communications, therefore giving a better performance. They are also
separate from designer issues like the state transfer: how to transfer the state to a new replica joining the group.

Group communication systems usually have some support on this transfer, based generally on a one step state transfer: a replica is requested to give its state, that is sent to the joining replica, using the same communication primitives as for the other group communications. Other systems, like Cactus [15], are based on logging mechanisms, where each replica must periodically return its state, which is logged together with the new messages arriving at that replica. In this case, a new replica can receive the state from those logging mechanisms, but it means that every replica is continuously spending processor time on that task. This is the same approach taken by CORBA, but while it is perfectly reasonable for non-active replicas, we believe it is less suitable for active replication. We have compared state transfer protocols [16] and the conditions that must be achieved on a virtual asynchronous system during the state transfer from other replicas [17], considering as well the case that a transfer is made in several steps. The generic solution is to block any group communications during the state transfer, stopping the transfer in case of view changes (changes on the group composition).

The virtual synchronous model states that every replica surviving two consecutive views must process the same messages between those views. This is the reason for the need to cancel any transfer while installing a view: as the messages are blocked, they must be unqueued and processed before installing the new view.

Sensei allows a special behaviour mode, which roughly means excluding the involved members during a transfer, which then must process any delayed message before re-joining the group. This mode means that the transfer is not interrupted due to view changes, but it clearly violates the virtual synchronous model, and applications strictly based on it would not work properly. The exclusion of the members is not real, just an abstraction; messages can be defined in Sensei as being not blocked during state transfers, and the members on transfer do also receive these messages. Examples of these messages are those used by the infrastructure to perform the state transfer. To use this feature, messages must be defined as DomainMessages, defined in IDL CORBA as:

```idl
class DomainMessage {
public:
  bool unqueuedOnST;
  bool untransactional;

...}
```

The first field indicates when a message is received even during a state transfer, and it is false by default. The second field is explained in the next section.

The support of state transfers in several steps is useful for big components, but its advantage is in its optimization possibilities. A container is blocked while it is transferring its state; if it can transfer it in several steps, it is blocked more times but for less time, giving probably a faster answer to its user when the state is big. The state transfer becomes
more complex, and the container must deal with messages received during the state transfer, probably modifying its state, but this is completely hidden to the final application.

*Sensei* extends the flexibility on state transfers by allowing the applications to choose the members that make the transfers. An application requiring fast answer times could allocate a replica to do this task exclusively; none of the other replicas servicing client requests would be blocked because of state transfers.

6. **Transactions**

In our model, a transaction is the mechanism used to avoid inconsistencies across multiple replicated components when a running process or its host falls down.

Although transactions can be based on *undoable* components, allowing at the same time the possibility to abort transactions, this design would transfer to those components most of the transaction support. This would translate into a more difficult implementation of those components but also into a more difficult and less transparent use.

The implementation used in *Sensei*, based on delaying group communications during the transfer, implies two problems related to the virtual synchrony model. First, not every member in the group observes the same messages in the same order. Second, the group could install a new view during a transaction, and because it is not possible to cancel the transaction, not every member in the group would observe the same messages between two consecutive views. Therefore, the use of transactions under this second design violates the virtual synchronous model. However, as their use depends on the application, this can choose whether that violation is permissible. This argument is based on the following reasoning: the consistency of any component through different replicas is questioned if two different members in the group are able to perform concurrent updates on a component that has been built on the assumption that no such concurrency is allowed. The approach is then to avoid the possibility of such concurrent updates by using monitors.

Therefore, transactions are built in *Sensei* around distributed monitors, protecting the components inside the transaction against other external parallel updates, at least if the other updates are also carried out properly by first locking the same monitor. Two transactions can be done in parallel using different monitors, as far as the component updates to be carried out, they can be performed concurrently. If a component can be queried without first acquiring a lock, it means that the message associated to that query can be processed without affecting other component messages.

The second problem is that not every replica observes the same messages between two different views. This means, as happened with the state transfer support, that applications relying strictly on the virtual synchronous model will not behave properly using transactions. In the case of an application based on the certainty that fallen down replicas have been able to process every message up to a specific view, the security that the living replicas have no inconsistencies is not sufficient to maintain the application requirements.

Therefore, our proposal for transactions and their implementation in *Sensei* completely depends on the quality of the code being using. If a replica updates its components or group
of components guarded by a monitor or a transaction, the violations on the virtual synchronous model are not translated into inconsistencies on their states. The drawback, as stated before, is on applications with requirements on the states of replicas which have fallen down.

In order to respect the virtual synchrony model, Sensei forbids the use of transactions on applications where the model is strictly required. In the previous section, we described a work model for the state transfer where the members involved are excluded from the view, simplifying the transfer but not being totally compatible with the virtual synchrony model. Sensei only allows transactions when the group is working under the previous model.

There is another interaction between transactions and the state transfer; because a replica blocks its group communications during a transfer, its state is not shared with other members. It is therefore not permitted for a state transfer to be started until the transaction is completed; otherwise, it would first send a partially changed state and later the queued messages, making it difficult for the incoming replica to decide how the final state should be built. Note that if a member locks a monitor, it also prevents other replicas from doing their job until the monitor is unlocked. Following the previous logic, the state transfer should therefore come from a replica that is not locking any monitor, as it would delay its unlocking and therefore slow down the whole group.

When the application carries out the transaction, the queued messages are sent to the other members. The message queuing has an interesting effect: group communications across several components can be grouped into a single message, therefore achieving a better performance.

7. Transactions and monitors

The interface on domains to support transactions is very simple, based on the definitions of monitors, that in JavaRMI, without the specification of exceptions, is:

```java
public interface Monitor extends Remote{
    public void lock() throws ...
    public void unlock() throws ...
}
```

```java
public interface TransactionsHandler extends Remote {
    public void startTransaction(Monitor monitor) throws ...
    public void endTransaction() throws ...
}
```

The possibility to nest transactions is essential to keep the components abstraction. Otherwise, if a component uses a transaction internally, this component could not be used by others under another transaction, making the use of transactions inside components potentially impossible.

To start a transaction requires first locking a monitor, but if communications are not sent to the group, that lock would be totally useless, as other members would not be aware of it and could therefore lock the same monitor themselves, probably resulting in replica inconsistencies.
For this reason, not every group communication is blocked during a transaction. As seen in the previous section, messages in Sensei can include a flag that defines them as not being blocked on transactions. However, components using those messages become more complex, as they lose the transaction transparency maintained by the system; from the replicated containers used in Sensei, no one needs to set his flag.

Messages queued under a nested transaction are not sent to the group until the outer transaction is finished. Otherwise, another replica could start a transaction over a component that does not reflect its latest state. The same applies to the unlocking of a monitor: while locking a monitor is translated into a non-blocked message to the group, unlocking a monitor produces a message that is queued together with the other messages in the transaction.

8. Conclusions

The virtual synchrony model used to build replicated systems provides in its definition very basic communication primitives. The use of object oriented wrappers and serialization patterns giving way to typed messaging facilitates the application programming under the model. The approach taken by CORBA isolates the server from communicating with other replicas, therefore being very effective in the deployment of solutions. However, to define the whole server as the replication unit is not the most adequate approach in many cases, not only because of performance issues but also due to design problems.

Our proposal to facilitate the development of replicated applications is based on the use of replicated components, with the underlying system already providing the most necessary components (e.g., collections). To successfully create these replicated components, the system must provide a strong state transfer support. Moreover, to make possible the use of these components, the system must include support for grouping components, use of replicated monitors and, on a higher level, for transactions between the members in the group.

Sensei already includes these features: transactions, domains, and a strong state transfer support, for both CORBA and JavaRMI. Our current work is to complete a library of replicated components, which initially only includes containers based on the java.util package. Our goal is to be able to write replicated applications paying minimal attention to the group communications, and therefore focusing on the application logic.

This focus on supporting data containers as the main replication unit is based on our experience, where the state of a server can usually be captured using these containers. Following this logic to an extreme, JavaSpaces [18], a product based on JavaRMI, works using a paradigm of distributed programming based on access to a distributed data structure instead of the normal client/server paradigm based on messages. This data structure is essentially a hashmap, where operations have atomic semantics and full transaction support. Applications communicate with others by updating the distributed data structure, which could be replicated to achieve fault tolerance, although it is not required. Sensei only makes the use of replicated data easier, the JavaSpaces programming model is not directly supported, due to its transaction requirements.
References


Building Test Constraints for Testing Distributed Systems with Middleware

Jessica Chen
School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
xjchen@cs.uwindsor.ca

Abstract
Reproducible testing is one of the effective ways to perform or repeat a desired test scenario in concurrent systems. When we apply it to distributed systems where remote calls are used as communication facilities, new deadlocks may be introduced when we incorporate the test control mechanism to the execution of the program under test. A static analysis technique has been thus explored to solve this problem. Using the static analysis, we can derive a test model from a given test constraint. This test model is then used in the test control procedure to help the test controller to avoid deadlocks. The test constraint carries information of not only the constraint from the test users, but also the program structure related to the remote calls. In this paper, we present our work on the automated construction of such test constraints from user’s input and the program source code.

Keywords: Distributed Systems, Nondeterminism, Test Constraints, Middleware.

1 Introduction
With the advances of modern computers and computer networks, distributed and/or concurrent software systems are becoming more and more popular. We are concerned about the quality of these systems just as we are for sequential ones. Testing is still our primary device to determine the correctness of a software system nowadays. However, testing a distributed and/or concurrent system is very often much harder than testing a sequential one, mainly due to the involved nondeterminism. Unlike in traditional sequential systems, a given input sequence to a distributed and/or concurrent system may have several different execution paths depending on the interactions among different threads and/or different processes possibly running on different machines across the network.
Reproducible testing [2, 3, 12, 13] is one of the possible approaches to performing testing in a concurrent environment. In this approach, for a given test case, some additional information such as partial or total order of some statements in the program is provided and the program is forced somehow to take certain execution path based on the additional information. Then the external observations are compared with the desired ones. However, forcing a concurrent system to take a particular execution path manually during a test may be fairly difficult and tedious. Some kind of automated control mechanism needs to be developed and integrated into the system during the test. Such test control mechanism will interact with the Program Under Test (PUT) and force the system to take the desired execution path based on the given input and some additional information. The control is usually done by artificially blocking some threads/processes at certain points and letting other threads/processes proceed. The execution of the PUT is augmented by additional communications between the control mechanism and all the threads in the PUT, possibly in different processes. Each thread communicates with such control mechanism whenever it needs to coordinate with other threads in its own process (via monitors etc.) or in other processes (via remote method calls, etc.). This communication can be introduced either via automatic code intrusion or by altering the execution of the underlying execution environment (such as Java Virtual Machine for java programs). We consider the former approach, as discussed in [2, 3, 4, 5, 12]. With the added communication, a controller is able to decide whether a thread should proceed, wait for other threads, or resume from waiting state, based on the overall concurrency constraint on the system and the current status information of other threads.

For a given input, we need to define the additional information to identify the desired execution path, such as when and where to block which threads/processes during the system execution. Usually, the nondeterminism in concurrent systems are caused by synchronization among different threads and processes, hence, as discussed in [1, 4, 5, 9, 11], the additional information is about the order of the threads and processes at the synchronization points. The test control mechanisms very often just block certain threads/processes at some of these points. In the following, we consider the additional information (called input constraint) as happen-before relation among some synchronization events [4], i.e. accesses to shared objects. As we mentioned in [3], in order to keep the code intrusion unique to different test scenarios, we make the PUT contact the test controller whenever a thread/process accesses a shared object, while the test controller will pay attention only to those defined in a particular scenario.

In a distributed system, communications among different processes can be realized via Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), stream sockets, (virtual) distributed shared memory, etc. When middleware layers like CORBA, DCOM, Java RMI etc. are used, we can actually consider remote procedural/method calls virtually as local calls. Thus, lots of discussions on testing concurrent systems in certain sense can be applied to distributed systems. However, as pointed out in [6], the soundness of a test control mechanism may depend on the underlying implementation of process
communications in the distributed architecture. An effective technique to avoid introducing new deadlocks was also presented there. Basically, we assume that the test constraint provides the information about the ordering among the starting and completing of some synchronization events and all related remote call events, i.e. the calls for remote procedure/method. A test model is constructed in terms of finite automata, according to such a test constraint and the thread model used in the underlying implementation of process communications. This test model is then used in the test control procedure to help the test controller to avoid deadlocks.

The test constraint used in this approach contains essential information for the test model construction. To provide such information, the test users are required to have a good understanding of the way the test controller works, and to search out lots of information on the program structure with respect to the remote calls in addition to the original input constraint in their mind, that is, the ordering among some synchronization events. This is of course too demanding. In this paper, we present our work on how to automatically construct the test constraint according to the user’s input constraint, and the available source code of the PUT.

The rest of the paper is organized as follows. In Section 2, we use CORBA middleware as an example to show the deadlock problem introduced by incorporating test control mechanism into the PUT. Our test tool architecture with the solution to solve the deadlock problem is presented in Section 3. In Section 4, we explain in details how to obtain the test constraint from a given input constraint. Related work is summarized in Section 5 and we conclude our work in Section 6.

1.1 New deadlocks in the test control procedure

As we know, when a process makes a remote call, an implicit separate thread on the server site may be used to handle it. Using CORBA middleware, there are a few underlying thread models to realize this. For example, we have pool of threads model, where a pool of threads is created to handle incoming remote call requests. The size of the pool is fixed. We also have thread-per-object model where one thread is created per remote object. Each of these threads accepts remote calls on one object only. The concurrent threads on the server site for the remote calls usually are limited. With only limited threads available, remote calls will become the contention resources and thus, new deadlocks may be introduced when we execute a system under the control mechanism.

Now we use an example to show a scenario when the test control introduces new deadlocks into the execution of the PUT. This example will be used later on for the discussion on the automated construction of test constraints.

Let us consider an application of on-line conference control. With the use of Internet and multimedia, it is possible to host an on-line conference. Now let us consider such an on-line conference application that allows only one person to speak at a time. We can use a central server to realize the control. Each participant is a client who must request the permission from the server each time he/she wants to speak and must inform
the server of its completion of his/her talk. Using CORBA, these two activities can be accomplished by calling remote methods e.g. \texttt{request()} and \texttt{finished()} respectively. Figure 1 shows the interface definition of these two remote methods.

The skeleton of the control part of the server implementation is given in Figure 2. The bold lines are inserted code for the purpose of testing and will be explained later on.

The server is in charge of granting permissions to clients to speak and it needs to guarantee that only one client can speak at a time. This is achieved through an implicitly used token. When the token is available (i.e. \texttt{token=ture}), the server knows that currently nobody is speaking.

When the token is not available, all incoming calls for \texttt{request()} are put into a queue. The server keeps checking the status of the token and the client's waiting queue. When the token becomes available, the server wakes up the first client in the queue, if there is any, to allow the client to speak.

As we know, the Java language and runtime system support thread synchronization through the use of \texttt{monitors} originally introduced in [7]. Generally, the critical sections in Java programs are defined as a statement or a method (identified by the \texttt{synchronized} keyword), and Java platform uses monitors to synchronize the access to this statement/method on an object: each object with \texttt{synchronized} statement/method is a monitor that allows only one thread at a time to execute a \texttt{synchronized} statement/method of that object. This is accomplished by locking the object when a \texttt{synchronized} method is invoked so that no other thread can invoke any \texttt{synchronized} method on this object at the same time. A \texttt{synchronized} method automatically performs a \texttt{lock} action when it is invoked; its body is not executed until the \texttt{lock} action has successfully completed. When the execution of the method's body is ever completed, either normally or abruptly, an \texttt{unlock} action is automatically performed on that same lock. In addition to having an associated lock, every object with \texttt{synchronized} method has an associated waiting queue of threads. A thread executing in a \texttt{synchronized} method may voluntarily \texttt{call wait()} to release the lock on the monitor object and put itself into the waiting queue of this object. When \texttt{notify()} is called and the waiting queue is not empty, the first thread in the queue is removed from the waiting queue and re-enabled for thread scheduling. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on

```java
module CAE {
    interface ConfControl {
        void request();
        void finished();
    };
}
```

Figure 1: The CORBA interface definition in on-line conference example
public class ConfControlImpl extends ...... {
    ......
    private boolean token = true;
    private int waitings;
    public void confControlManager {
        while (true) {
            try {
                cT.message(pName, tName, 1, "-", "synReq");
                // && 1 && //
                synchronized (this) {
                    if (token == true && waitings > 0) {
                        token = false;
                        waitings = waitings - 1;
                        notify();
                    }
                }
                cT.message(pName, tName, 1, "-", "synCom");
            } catch (Exception e) { e.printStackTrace(); }
        }
    }
    public void request() throws RemoteException {
        cT.message(pName, tName, 2, "-", "synReq");
        // && 2 && //
        counter.inc();
        cT.message(pName, tName, 2, "-", "synCom");
        cT.message(pName, tName, 3, "-", "synReq");
        // && 3 && //
        acquire();
        cT.message(pName, tName, 3, "-", "synCom");
    }
    public synchronized void acquire() {
        try {
            waitings = waitings + 1;
            cT.message(pName, tName, 4, "-", "synReq");
            // && 4 && //
            wait();
            cT.message(pName, tName, 4, "-", "synCom");
        } catch (Exception e) { e.printStackTrace(); }
    }
    public synchronized void finished() throws RemoteException { token = true; }
}

Figure 2: The skeleton of the server program for the on-line conference control example
this object.
In the implementation of the server program of the above example, we make use of such a waiting queue of Java monitors. In each request, \textit{wait()} is called so that the current thread is put into the waiting queue of the monitor of the remote object. Correspondingly, the server uses \textit{notify()} to wake up one of the client, as defined in \textit{confControlManager()}. In this setting, the server only needs to check the number of clients who are currently waiting for the token. This number is kept in integer \textit{waitings}.

In order to collect statistic data, we maintain the number of client’s requests to talk. This number is kept in integer variable \textit{count} in object \textit{counter} of class \textit{Counter}. For simplicity, class \textit{Counter} is not shown in the figure. Corresponding to variable \textit{counter}, we have defined in class \textit{Counter} synchronized methods \textit{inc()} and \textit{get()}. Method \textit{inc()} increases the current counter by 1, and method \textit{get()} returns the current counter value. Whenever there is a remote call for \textit{request()}, \textit{count} is increased by one.

Assume that in our test, there are three clients with processes \( p_1, p_2 \) and \( p_3 \) respectively. Each process has one thread \( t \). The server process is \( s \).

As we mentioned in the Introduction, an input constraint is expressed as a happen-before relation on synchronization events. In this example, we have two shared objects. One is monitor \( r \) of the remote object, and another is the monitor of object \textit{counter}. Suppose we want to test a case when each of the three client processes \( (p_1, p_2 \) and \( p_3 \)) requires to speak once. In the input constraint, we require that \( p_2 \) call \textit{method acquire()} and access shared object \( r \) before \( p_1 \) and \( p_3 \). In the real execution, \( p_1 \) and/or \( p_3 \) may call \textit{acquire()} and access \( r \) before \( p_2 \). The traditional way to handle this is to let \( p_i \) (for \( i = 1, 2, 3 \)) communicate with the test controller for permission to access \( r \). In the case that the request to access \( r \) arrives first from \( p_1 \) or \( p_3 \), the test controller will delay its response until it knows that \( p_2 \)’s access to \( r \) is completed.

Now, assume that \textit{pool of threads} model is used, and that the maximum number of threads in the thread pool of the server process \( s \) is 2. It is possible that during the execution, \( p_1 \) followed by \( p_3 \) requests the controller to access \( r \) before \( p_2 \) calls the remote method \textit{request()}. In this case, both requests will be suspended by the controller. On the other hand, before sending requests to the controller for permissions to access \( r \), \( p_1 \) and \( p_3 \) have already occupied the two threads in the thread pool of \( s \) when they called remote method \textit{request()}. When \( p_1 \) and \( p_3 \) are suspended by the controller, they will not release these implicitly used threads. Then we are in a deadlock state because: (i) the controller will resume \( p_1 \) and \( p_3 \) only after \( p_2 \) has accessed \( r \); (ii) \( p_2 \) will need a thread available in the thread pool of \( s \) in order to call the remote method \textit{request()} before it can access \( r \), yet there will be no thread available in the thread pool until \( p_1 \) or \( p_3 \) finishes the remote call.

Such a deadlock state is obviously introduced by our test control mechanism.

The underlying CORBA implementation of process communications can be user defined. Apparently, if for each request or each client, it is guaranteed that a separate thread on the server site for remote calls will be available to handle it, and then the implicit threads used on the server site will not be part of the contention resource. As a direct consequence, we do not have the above-mentioned problem if \textit{thread-per-request}
model or thread-per-client model is used. The problem typically arises when pool of threads model or thread-per-object model are adopted.

2 Test control architecture

The solution used in [6] to handle the above-mentioned deadlock problem is to control the order of remote calls together with the order of synchronization events. Figure 3 illustrates the structure of our test control architecture for reproducible testing in distributed systems.

The automated code insertion part augments the program with the communication with the test controller. This communication refers not only to synchronization events, but also to remote calls. In order to achieve the unique code insertion with
respect to various test scenarios that may refer to different synchronization points in the source code, all synchronization points and remote calls are identified and actually enumerated. The enumeration is inserted into the source code as special comment line like

```
// && 1 && //
```

where \( l \) indicates the location of a synchronization point or a remote call (cf. Figure 2).

For each test scenario under test, apart from the test data, the test user also provides the desired input constraint which specifies the ordering among synchronization events. These synchronization events are identified by the location number inserted into the source code.

An input constraint is further refined into a test constraint and we present in the next section the details on the automated construction of it.

From the test constraint, we obtain an abstract test model according to the thread model used in the underlying implementation of process communications. The test model is given in terms of finite automata. It expresses the control structure that the test controller can use in order to satisfy the expected test constraints. The edges of the automaton are decorated with the synchronized events and the remote call events referred to in the test constraint. An edge with event \( e \) from state \( s \) expresses the fact that when the test controller is in state \( s \), it will allow event \( e \) to happen. The constructed automaton contains all possible control paths that can (i) lead to a successful execution; and (ii) satisfy the test constraint. Technically, the automaton is constructed in two steps:

1. First, we construct an automaton that contains all possible paths according to the given test constraint. Note that such an automaton may contain deadlock states i.e. those non-final states from which there is no outgoing edge.

2. Second, we provide an operation on this automaton to prune those unwanted states that will lead to deadlocks. Note that some nodes that are initially not deadlock states may become deadlock ones when their outgoing edges are removed during the pruning procedure, so the pruning procedure needs to recursively remove all deadlock states. Thus obtained test model is deadlock-free in the sense that along any existing path we will eventually arrive at the final state.

With the statically obtained deadlock-free test model, upon receipt of a request from a process to make a remote call, the test controller will be able to decide whether the permission should be granted, taking into account both the current execution status and the underlying thread model adopted, in order to avoid leading the execution into a deadlock state. If pool of threads model is adopted, where the limit of the number of threads in a pool for process \( p \) is \( n \), the controller will dynamically decide which (maximum \( n \)) remote calls should be allowed at each moment to be executed within \( p \). If thread-per-object model is adopted, the controller will decide for each remote object and at each moment, which remote call should be allowed to execute.
3 Test Constraint Construction

Now we present the procedure to build up test constraints from given input constraints. This is realized in two steps and we present them in Section 3.2 and Section 3.3 respectively.

3.1 Input constraint

In our context, a synchronization event is represented by a quadruple \((p, t, l, n)\) where

- \(p\) is the name of the process this event is from.
- \(t\) is the name of the thread this event is from.
- \(l\) is the place of the statement to be controlled. This place is indicated by the location number in the extended source code as provided by the code insertion tool.
- \(n\) is the number of appearances of this synchronization event at location \(l\) in thread \(t\) and process \(p\). We need this number because a statement can be executed several times by a same thread in a same process.

An input constraint expresses the desired partial order among some synchronization events. Figure 4 (a) illustrates a possible input constraint in the conference control example. Here the nodes denote synchronization events and arrows denote the happen-before relationships on these events. Recall that in Figure 2, location 3 represents the call for synchronized method \(acquire()\). So event \((p_1, t, 3, 1)\) denotes the synchronization event of the first call for synchronized method \(acquire\) from thread \(t\) of process \(p_1\). Similarly, event \((p_2, t, 3, 1)\) and \((p_3, t, 3, 1)\) denote the synchronization events of the first call for synchronized method \(acquire\) from thread \(t\) of process \(p_2\) and \(p_3\) respectively. Thus, the input constraint in Figure 4 (a) expresses that \(p_1\) and \(p_3\) cannot access synchronized method \(acquire\) before process \(p_2\). In other words, \(p_1\) and \(p_3\) cannot obtain monitor \(r\) before process \(p_2\).
3.2 Test constraint without remote call events

To realize the control so that a synchronization event \( e_1 \) happens before another synchronization event \( e_2 \), the test controller actually controls the execution so that \( e_2 \) cannot start until \( e_1 \) is completed. In order to do that, a process should communicate with its test controller twice for each synchronization event: (i) to request for permission to start it; (ii) to acknowledge the completion of it so that some other requests may be granted to continue.

While synchronization events are from the test user’s viewpoint, we use test events to denote the communications happened between the program under test and the test controller. Test events are classified into different types. For example, for each synchronization event, we have two corresponding internal test events between the PUT and the test controller:

1. a synchronization request event to denote a request sent to the test controller to access a shared resource;

2. a synchronization completion event to denote the acknowledgement sent to the test controller for the completion of the corresponding synchronization event.

We use type \( \text{synReq} \) for requests to access shared objects, and type \( \text{synCom} \) for acknowledgements of having obtained a shared object.

A test event contains more information than an input event. Generally, a test event is a 6-tuple \((p, t, l, n, o, ty)\) where

1. \( p, t, l, n \) are as defined in input events.

2. \( o \) is the remote object name if this is a remote call event. We will discuss this element later on. For synchronization request event and synchronization completion event, this element is ignored and we use “-” to indicate it.

3. \( ty \) shows the type of the event.

For example, in the conference control example, test event \((p_1, t, 3, 1, -, \text{synReq})\) denotes a request for the first call for synchronized method \texttt{acquire} from thread \( t \) of process \( p_1 \).

A test constraint expresses a partial order among test events. Figure 4 (b) shows an example of a test constraint. Each time the test controller receives a message from the PUT, it will check the test constraint to see if this event appears. If this event does not appear in the test constraint, the message will be ignored, and the PUT receives permission immediately if the event is a request event. Otherwise, the test controller will control the execution according to the happen-before relationships given in the test constraint.

The test controller identifies a received message with a test event in the given constraint by comparing the information contained in the message with the elements in the test event. Note that, a test event contains six elements. However, when a
process communicates with its controller, it only provides five pieces of information: the process name, thread name, location, remote object name, and event type. Examples can be found in the inserted code in the conference control example (Figure 2). The missing information for the comparison is about the number of appearances of the statement. This piece of information is dynamically counted by the test controller and then associated with the other information in a received message.

Without loss of generality, we assume that the process name is given as the program is started and it can be accessed via a special method call \texttt{getPName()}. In Figure 2, we used \texttt{pName} as an abbreviation of it. Similarly, we used \texttt{tName} as an abbreviation of

\begin{verbatim}
Thread.currentThread().getName()
\end{verbatim}

to retrieve the name of the current thread. We have assumed that each thread is explicitly given a name that is unique in its process.

Now we explain the automated transformation from a given input constraint into the corresponding test constraint. For the moment, we do not consider remote call events. Note that in a test constraint, a happen-before relationship is always between a synchronization completion event and a synchronization request event. Thus, a happen-before relationship

\[(p_1, t_1, l_1, n_1) \rightarrow (p_2, t_2, l_2, n_2)\]

in the input constraint should be replace by

\[(p_1, t_1, l_1, n_1, -, symCom) \rightarrow (p_2, t_2, l_2, n_2, -, synReq)\]

Let \(\langle E, R \rangle\) represent an input constraint where \(E\) is the set of synchronization events appeared in the constraint and \(R\) a binary relation showing the partial order among these events. Table 1 shows the pseudocode of the transformation.

If a synchronization event appears in the given input constraint without any other synchronization event that should happen before it, then we change the event into a synchronization completion event, and make related changes in the happen-before relationships. This situation is illustrated in Figure 5. Analogously, if a synchronization event appears in the given input constraint without any other synchronization event that depends on its occurrence, then we change the event into a synchronization request.
\[ E : \text{set of synchronization events in the given input constraint} \]
\[ R \subseteq (E, E) \text{ is a binary relation} \]

for each \((p, t, l, n) \in E\) do
  if \(\exists e \in E \text{ s.t. } e \rightarrow (p, t, l, n) \in R\) then
    replace \((p, t, l, n)\) by \((p, t, l, n, -, \text{synCom})\) in \(E\)
    for all \(e \in E \text{ s.t. } (p, t, l, n) \rightarrow e \in R\)
      replace \((p, t, l, n) \rightarrow e\) by \((p, t, l, n, -, \text{synCom}) \rightarrow e\) in \(R\)
  else
    if \(\exists e \in E \text{ s.t. } (p, t, l, n) \rightarrow e \in R\) then
      replace \((p, t, l, n)\) by \((p, t, l, n, -, \text{synReq})\) in \(E\)
      for all \(e \in E \text{ s.t. } (p, t, l, n) \rightarrow e \in R\)
        replace \(e \rightarrow (p, t, l, n)\) by \(e \rightarrow (p, t, l, n, -, \text{synReq})\) in \(R\)
    else
      remove \((p, t, l, n)\) from \(E\)
      add \((p, t, l, n, -, \text{synReq})\) and \((p, t, l, n, -, \text{synCom})\) to \(E\)
      add \((p, t, l, n, -, \text{synReq}) \rightarrow (p, t, l, n, -, \text{synCom})\) to \(R\)
      for all \(e \in E \text{ s.t. } e \rightarrow (p, t, l, n) \in R\)
        replace \(e \rightarrow (p, t, l, n)\) by \(e \rightarrow (p, t, l, n, -, \text{synReq})\) in \(R\)
      for all \(e \in E \text{ s.t. } (p, t, l, n) \rightarrow e \in R\)
        replace \((p, t, l, n) \rightarrow e\) by \((p, t, l, n, -, \text{synCom}) \rightarrow e\) in \(R\)
  end_for

Table 1: Algorithm to construct the test constraint (without remote call events)

![Diagram](attachment:image.png)

Figure 6: Moving from synchronization events to test events (case b)
Figure 7: Moving from synchronization events to test events (case c)

event, and make related changes in the happen-before relationships. This situation is illustrated in Figure 6. Finally, if a synchronization event appears with both an event that should happen before it, and an event that depends on its occurrence, we split the event into a synchronization request event and a synchronization completion event, and make the related changes. This situation is illustrated in Figure 7.

Figure 4 (b) is derived from Figure 4 (a) according to this procedure.

3.3 Test constraints

As discussed in [6], to avoid introducing new deadlocks, the control mechanism should also control the execution order of some related remote call events. When a process makes a remote call, it needs to send to its controller (i) a remote request event to get permission from its controller before the call; and (ii) a remote completion event to inform its controller that it has finished the execution of the body of the method. These are two other types of test events. We use remReq for the type of a request to call a remote method, and remCom for the type of a completion message of executing a remote method. According to the procedure of the test model generation, for remote call events, we also need the name of the remote object used to make the call and the name of the server being called. For simplicity, we assume that the latter can be retrieved via the former. Thus, for instance, \((p_1, t, 5, 1, remobj, remReq)\) denotes a request for the first remote call for method request by remote object remobj from thread \(t\) in process \(p_1\). Here location 5 is the place where the client makes a remote call for request. This is not shown in Figure 2.

A remote call is related to an input constraint if it invokes (directly or indirectly) certain synchronization events in the input constraint. For each of these remote calls, both the related remote request event and remote completion event are essential in the test model generation. They are used during the construction of a test model to statically determine the place where a remote call is needed or completed and thus the
related thread on the server site is allocated or released respectively. Now we discuss how to enrich the test constraints derived from Section 3.2 with the related remote request and remote completion events.

First of all, we need to find out the chain of remote calls that can invoke the synchronization events in the given input constraint. We can obtain such information via the method invocation graph that we retrieve using ordinary compiler techniques. The method invocation graph is a directed graph where each node represents a method, and each edge from node a to node b represents that the method in b is called within the method in node a. In particular, there are some nodes that do not have incoming edges. These represent the starting points of the programs. We will call these nodes roots. Typically, a client/server application has two roots that represent the starting points of the server program and client program respectively. In our setting, the normal method invocation graph is modified in the following ways:

a. Each synchronized block is implicitly given a unique name and it is treated just like a synchronized method.

b. All nodes corresponding to remote methods are marked as “remote” so that we can identify all the remote methods in the graph. Moreover, these nodes carry the names of the remote objects that we use to call the corresponding remote methods.

c. If an edge corresponds to a synchronization event in the extended source code, we put the location number of that event on the edge.

Item c. guarantees that for each location given in the extended program, we can find in the graph the edge marked with that location.

Now for each event \((p, t, l, n)\) in the input constraint, we can find in the method invocation graph the edge with the same location number \(l\). For simplicity, we assume that from the starting point of each program, there is at most one sequence of remote calls (possibly with local calls in between) that leads to each synchronization event. Thus, for each edge we found, we can derive from each root at most one sequence of remote calls that leads to this edge. The sequence we are interested in is from the root of program \(p\). This sequence expresses the order of remote calls in order for \(p\) to reach location \(l\). If the length of the sequence is zero, we do not need to do anything for this event. Otherwise, let the location numbers on the incoming edges of the remote methods along the sequence be \(l_1, \ldots, l_k\) and the corresponding remote object names in the nodes of remote methods be \(o_1, \ldots, o_k\) \((k \geq 1)\). We enrich the test constraint derived from Section 3.2 by adding test events

\[
(p, t, l_1, n, o_1, remReq), \ldots, (p, t, l_k, n, o_k, remReq),
(p, t, l_1, n, o_1, remCom), \ldots, (p, t, l_k, n, o_k, remCom),
\]

if they do not exist yet in the current constraint under construction. Together with them, we add their relationships

\[
(p, t, l_i, n, o_i, remReq) \rightarrow (p, t, l_{i+1}, n, o_{i+1}, remReq),
\]
Figure 8: Test constraint in the conference control example

\[(p, t, l_{i+1}, n, o_{i+1}, remCom) \rightarrow (p, t, l_i, n, o_i, remCom).\]
for \(1 \leq i \leq k - 1\). Furthermore, if both \((p, t, l, n, -, synReq)\) and \((p, t, l, n, -, synCom)\) appear in the test constraint derived from Section 3.2, we add happen-before relationships

\[(p, t, l, n, o_k, remReq) \rightarrow (p, t, l, n, -, synReq),\]
\[(p, t, l, n, -, synCom) \rightarrow (p, t, l, n, o_k, remCom).\]

If \((p, t, l, n, -, synReq)\) or \((p, t, l, n, -, synCom)\) does not appear in the test constraint derived from Section 3.2, we add happen-before relationships

\[(p, t, l, n, o_k, remReq) \rightarrow (p, t, l, n, -, synCom),\]
\[(p, t, l, n, -, synCom) \rightarrow (p, t, l, n, o_k, remCom)\]
or

\[(p, t, l, n, o_k, remReq) \rightarrow (p, t, l, n, -, synReq),\]
\[(p, t, l, n, -, synReq) \rightarrow (p, t, l, n, o_k, remCom)\]
respectively.

In the above conference control example, for input event \((p_1, t, 3, 1)\), the sequence of remote calls to reach synchronized method call \textit{acquire()} (location 3) contains only one element, i.e. remote call \textit{request} (location 5) with remote object \textit{remobj}. Thus we add events

\[(p_1, t, 5, 1, remobj, remReq), (p_1, t, 5, 1, remobj, remCom)\]
and happen-before relationships

\[(p_1, t, 5, 1, remobj, remReq) \rightarrow (p_1, t, 3, 1, -, synReq),\]
\[(p_1, t, 3, 1, -, synReq) \rightarrow (p_1, t, 5, 1, remobj, remCom),\]

Similar events and relationships are added for \((p_2, t, 3, 1)\) and \((p_3, t, 3, 1)\). Figure 8 shows the test constraint derived from the one in Figure 4 (b).
4 Related Work

Some research work has been done in the past to deal with the nondeterminism in testing concurrent systems [1, 4, 5, 8, 9, 10, 11, 12, 13]. Generally, there are two basic approaches: random execution and controlled testing. Along the random execution approach (see e.g. [9]), we execute the program many times with the same input and then examine the result. Along the controlled testing approach, we try to force a concurrent system to take a particular execution path. This is very useful also for debugging. People have studied various automated control mechanisms to force the system to take the desired execution path. Typically, there exist in the literature various replay control techniques [4, 11] and reproducible testing techniques [2, 3, 12, 13].

In replay control techniques, we record during the first run, the internal choices among the nondeterministic behavior and then re-run it with the same inputs together with some control mechanism to force the program to run with the same choices. Replay control techniques are especially important for regression testing. Reproducible testing differs from replay control mainly in that the controlled execution sequence can either come from the record of the previous run or from other sources, e.g. requirement documents, design documents or even the program code. A possible combination of these two approaches is proposed in [5], as a specification-based methodology for testing concurrent programs.

The problem of newly introduced deadlocks in reproducible testing in distributed systems was pointed out and resolved in [6]. The solution is based on a given test constraint. The present work discussed the way to obtain such test constraint from user’s input on the given ordering among synchronization events.

5 Conclusions and Final Remark

Undoubtedly, it is of great importance to apply existing testing techniques to distributed system applications. [6] has shown a potential problem in doing so and proposed a possible solution. This solution assumes that a test constraint is given. However, such test constraint has considerable distance from the constraint in the test user’s mind: the test users should not be required to know the details of the theory behind the testing procedure in order to provide a correct test constraint. In the present work, we have explicitly distinguished different types of events in the constraints from the user’s viewpoint and those internally used for testing. Correspondingly, we have explicitly distinguished the constraint from the test user’s viewpoint and the one used to generate test model. The input constraint we introduced here is purely from the user’s viewpoint. We have introduced an automated tool to assist the test users to obtain test constraints. The clarification on the events and constraints, and the automated construction of test constraint are obviously a great help for us to avoid adding extra-burden to test users to understand the test constraint and go through the source code for more details about remote method calls. Thus, the presented work forms a
significant and important part of our overall testing technique for distributed systems with middleware.

As a final remark, we would like to mention that we used in this paper location numbers instead of method names (as used in [6]) in the test events. This is a more accurate way to identify the places of the statements in the program especially for synchronized block and wait() statement that may appear in different places without an explicit name.

6 Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References


