# Resolving coordination challenges in distributed mobile service executions

Ramón Alcarria*, Tomás Robles, Augusto Morales and Edwin Cedeño

Department of Telematic Systems Engineering,
Technical University of Madrid,
28040 Madrid, Spain
Email: ralcarria@dit.upm.es
Email: trobles@dit.upm.es
Email: amorales@dit.upm.es
Email: edwinc@dit.upm.es
*Corresponding author

**Abstract:** The internet of things enables environments where objects are fully interconnected, allowing the execution of smart services and the consumption of functionalities provided by surrounding objects. This loose-coupled object interconnection demands improvements in the control and data planes for an optimum coordination between distributed services in mobile devices. There are several coordination challenges in these environments related to the interaction between services through communication channels, the support of node and link disconnections and the transmission of events at runtime. This paper defines a coordination model and proposes solutions to these challenges by developing a cooperative service execution model for mobile environments, integrating the publish/subscribe paradigm for communicating event messages and improving its performance by using a gossip-based solution. Subsequently, we evaluate this model and analyse the improvements of the designed optimisation mechanisms over the message queue telemetry transport protocol and the ns-3 network simulator.

**Keywords:** service coordination; internet of things; workflow patterns; publish-subscribe; gossip.

**Reference** to this paper should be made as follows: Alcarria, R., Robles, T., Morales, A. and Cedeño, E. (2014) 'Resolving coordination challenges in distributed mobile service executions', *Int. J. Web and Grid Services*, Vol. 10, Nos. 2/3, pp.168–191.

**Biographical notes:** Ramón Alcarria currently is an Assistant Professor at the E.T.S.I Topography in the Technical University of Madrid while he finishes his PhD studies. He received his MS degree in Telecommunication Engineering from the Technical University of Madrid in 2008. His research interests are service architectures, sensor networks, service composition and prosumer environments. He is a member of IEEE, IEEE Communication Society and ACM.

Tomás Robles is an Associate Professor of Telematics Engineering at the E.T.S.I. Telecommunication in the Technical University of Madrid since 1991. He received an MS and PhD degrees in Telecommunication Engineering from Technical University of Madrid in 1987 and 1991, respectively. His research interest is focused on advanced applications and services for broadband networks, both wired and wireless networks.

Augusto Morales received his Bachelor's degree in Electronic Engineering in 2007 from the University of Panama and his Master's degree in Computer Systems Engineering from the Technical University of Madrid in 2010. Since 2008 he has been working in several areas related mobile publish/subscribe systems, service architectures, and network security while he pursues his PhD. He holds several IT certifications such as CEH, Security+, Linux+ and CCSE.

Edwin Cedeño received his Bachelor's degree in 1997 from the Technological University of Panama and Master's degree in Computer Science (2009), Network and Communications (2009), and Distributed Systems Engineering (2011), from the Metropolitan University of Science and Technology of Panama, Technological University of Panama and Technical University of Madrid, respectively. Since 2001 he is a professor of Telematics Engineering at the University of Panama. Currently, he continues his studies as a PhD student at Technical University of Madrid.

*This paper is a revised and expanded version of a paper entitled 'Resolving coordination challenges in cooperative mobile services' presented at the 'International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)', Palermo, Italy, 4 July 2012.*

## 1 Introduction

The Internet of Things (IoT) envisions a world in which all objects are interconnected and interact. The emergence of the Web of Things (WoT) inspires these heterogeneous objects to be accessible in the digital world. The convergence at the network level should also be applied to the service level, where the infrastructure has to provide appropriate abstractions to describe objects by the functionality or the information they provide. This evolution reflects the current user behaviour, which is primarily interested in real-world entities (e.g. things, places, and people) and their high-level states (e.g. empty, free, and walking) rather than in individual sensors and their raw output data.

Data-driven services allow these digital objects to interact and interchange data, in order to be processed (e.g. transformed, filtered, merged, used). Grid and web services provide an abstraction to access these processes through well-defined interfaces and allow data-driven services to be designed as workflows that capture the invocation logic.

The communication between elements from the WoT is often delegated to orchestration processes using WS-BPEL (Web Service Business Process Execution Language) (Khalaf et al., 2007) for information control. To focus more into complex coordination between entities or devices, a distributed model based on choreography is needed. Moreover, an execution model in which the services are fully specified before runtime is less desirable than the ability of services to interact by sending events and operations in a choreography model. Users can appear and disappear, they may also require the same service with different properties, for example in a QoS-based service provision such as in the work of Wang et al. (2010), while services are being executed so, services must adapt to these situations. Thus, using a choreography-based model for managing distributed service coordination introduces the challenge of an emergent behaviour of the services.

In order to decouple executing services and, thus, avoid coordination problems when services experiment an emergent behaviour, this paper defines a service model based on an event-driven approach, consisting of interaction between services, sending

notifications on state changes, which are initiated either by user actions on the user interface or by requests from other services. This approach combines the advantages of the control-driven workflow execution (which allows the development of more complex services and more control in the management of these services by the execution environment) and the data-driven approach (which handles the content exchange between activities), since it enables the possibility for events to contain service data. This is critical for services accessing the IoT to share content with the so-called web objects, such as databases, restful web services, or even in external services domains and clouds (Anjum et al., 2012).

We consider a mobile distributed service execution scenario which enables inter-organisational collaboration and interoperability of heterogeneous hardware. The services described in this paper are executed in mobile and fixed terminals in a distributed way, since the ubiquitous access to the functionality of the IoT and WoT objects (which support standard application layer protocols and techniques, such as HTTP or REST) is decoupled from the invocation control in workflow diagrams, as described by WS-BPEL and Service Component Architecture (SCA). Due to natural mobility characteristics of execution devices, which often suffer from problems related to lack of connectivity (lack of coverage or low battery), operations such as, *device disconnections and reconnections or the addition and removal of workflow participants* are required to be managed.

In this work, we define a cooperative service model for resolving coordination problems in mobile service executions. We also propose an underlying communication model based on the publish/subscribe paradigm (Eugster et al., 2003) and an algorithm optimisation of a gossip protocol to alleviate the consequences of the mobile nature of service executions, resulting in the problems stated in the previous paragraph.

Regarding coordination problems, we propose solutions to the identification of each element participating in the service interaction, the correlation between execution instances and the communication between processes composing the distributed service.

This paper is organised as follows. Sections 2 and 3 describe the service model and the communication model for cooperative mobile services in distributed service executions, respectively. Section 4 defines the distributed architecture for mobile terminals and the interaction between different modules. Section 5 contributes to resolve coordination challenges in distributed processes and Section 6 describes the most appropriate underlying communication model depending on the executing scenario and evaluates the paper contributions through the simulation of a distributed mobile service execution model. The paper concludes with related work and some conclusions of the proposed solution.

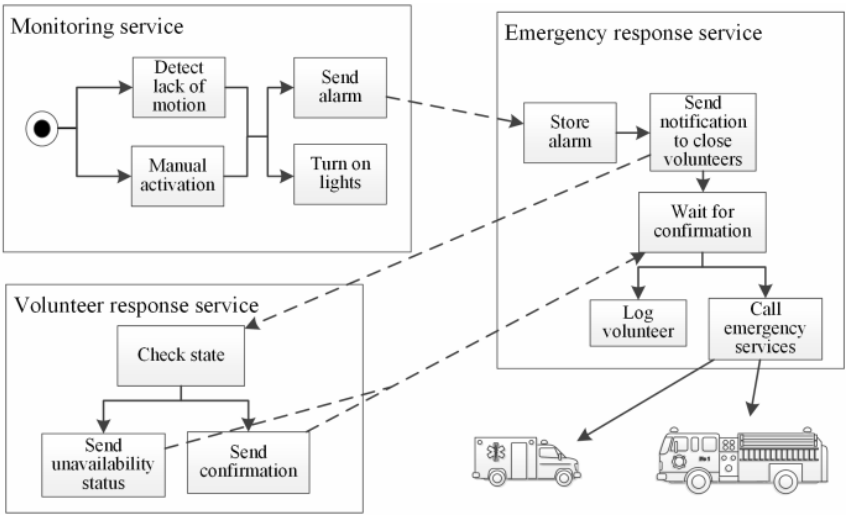
## 2 Service model

This section describes an example of a distributed mobile service that presents the coordination challenges discussed in the introduction and then defines a complex service model that takes into account the service internal logic and some abstractions to facilitate the understanding of service behaviour in mobile service execution environments. In the considered scenario, a set of services are coordinated in order to enable a distributed execution between different nodes from one or more networks. Each service is composed of activities and a logic that defines its execution. These activities exchange data and control events through a service execution engine and perform invocations to the so-called web objects through a communication middleware.

## *2.1 Motivating example*

The *home care assistance* service is a good service example from the described environment. This service, described in Figure 1, notifies a group of volunteers and supervisors some alerts related to the monitoring of elderly people in their homes. If a monitored person (*patient*) has an accident, the activity that is responsible for collecting the information generated by the motion sensor will produce an event that will turn all the lights on in the patient home and send an alarm to the response services. These services consist of a group of volunteers, whose response time is very short (usually neighbours) and emergency professional services (ambulances, health workers, fire-fighters, etc). Depending on the type of the generated alarm the alert is notified to one of the two groups or both.

**Figure 1** Motivating example: home care assistance service




The service is composed of three individual services (monitoring service, volunteer response service and emergency response service) and can be modelled as a workflow diagram consisting of activities that invoke functionalities such as the connection to the motion detector (to detect a lack of motion problem of the monitored patient), the home illumination system and mechanisms to respond to alarms.

This example involves some requirements over the service model:

- First, service execution is distributed among several entities and atomic services perform invocations to environment capabilities and IoT objects. Service distribution favours a more decoupled service execution and also sensible information is sent only to the entity that is authorised to manage it.
- As each entity consists of a set of activities, it should be taken into account that these entities pass through different states and require coordination with other entities to make transitions between states.

- Service coordination must be dynamically adapted to include more participants that arrive during service execution, as new patients or volunteers may appear. Events produced by publisher activities should be notified to dependent activities, and this number may vary over time.

### 2.2 Internal service model

The coordination model defined in this paper meets the requirements proposed in the previous paragraph and relates the concepts of services, tasks and activities. A *composed service* consists of a distributed workflow that can be executed. We define *individual service* (from now we call them just *services*) as each one of these distributed workflows, which are individually created or are part of a more complex service, has been fragmented/partitioned (Fdhila et al., 2010). The fragmentation process covers the actions of computing, initialising and distributing a set of fragments needed for carrying out a service. The service should consider other aspects such as user interaction, life cycle management and security, which are out of the scope of the paper. We also assume that all the services are successfully placed in mobile devices and the information about service interaction is stored in the SDL (Service Description Language) document, which contains the necessary information to execute the composed service.

A *task* is the instantiation of a service that performs a work. Tasks are arranged and initialised in the service bootstrapping process, which will be explained later. A task is composed of at least one activity. For example, the control service is instantiated in as many tasks as patients are concurrently monitored.

An *activity* is an atomic unit of a task. It manages the communication with an object, which can be physical or digital, to perform an operation. We classify operations according to their ability to produce data (sensors), consume data (actuators) and process data (processors). Activities trigger data and control events that are consumed by other activities in their own task scope or external task scopes. In the motivating example, a control event is generated from the *motion detection* activity and is internally notified to the *set audible alarm* activity. In addition, in a distributed workflow scenario there are interactions between activities from different services. The *alert notifier* activity generates a data event that is consumed by the alert receiver activities in the voluntary and emergency response services. We define *limit activity* as any activity that communicates with other activities contained in a different service by using data or control events.

### 2.3 Abstraction for service coordination

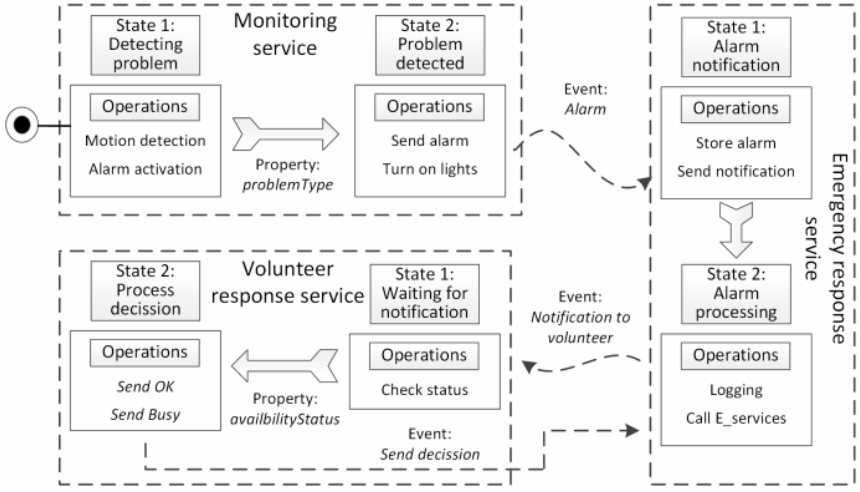
In order to manage the complexity of this service model, we define four abstractions: states, events, operations and properties.

- *State*: It represents a unique configuration of information in a service. It integrates a complete set of properties (name-value pairs) that represents something relevant and significant for the other services to know. A stateful service is something debatable as services are normally used in a stateless fashion. This is also why WSDL and WS-* standards omitted the notion of state, which some authors consider a mistake (Daniel et al., 2009), as stateful services are the natural way to bridge applications and data-oriented services.

- *Events*: Notify state changes. Events are initiated either by user actions on the UI, by requests from other services or by results from invocations to IoT objects. Events also can contain data as name-value pairs to be transferred to other dependent services. These services subscribe to events to enable coordination of executed services and state changing.
- *Operations*: Represent invocations made by service activities through their defined methods. The most common operations are related to invocation to IoT elements and local capabilities (functionalities provided by executing nodes or devices) and also to the provision of graphical elements such as user interfaces. Usually, operations cause state changes and the generation of the corresponding state change events.
- *Properties*: Represent configurable conditions at runtime or design time, which along with operation results determine the state to evolve.

Figure 2 shows the *home care assistance* service represented with these abstractions. Considering the first service, *monitoring service*, we note that it has two states: the first state expects to detect a problem and then the service evolves to a second state for invoking the necessary operations (*send alarm* and *turn on lights*). After the state change, the service produces an event that is consumed by the control service in order to start with the *alarm notification* state.

**Figure 2** Service model in home care assistance service




The reason for using this abstraction model for service communication is that the internal complexity of each service is difficult to coordinate and can be decoupled from the dependencies of this service. From the view point of service, coordination services are considered as event generators and the communication with them can be performed through a communication channel defined by the event type instead of through each of the communication points defined by limit activities. By decreasing the amount of communication points, the amount of communication ports and physical addresses is also reduced; so, performance in the communication layer is improved.

The use of this service model for distributed service execution scenarios involved some challenges. First, the interaction between services should be considered at the service and communication levels. In this work, we define logic gates between distributed services, corresponding to the most common workflow patterns (van der Aalst et al., 2003) and we enable communication establishment between limit activities by creating and optimising communication channels. Second, we describe how to integrate the runtime coordination based on states, events, operations and properties in the communication level. Finally, we show how the architecture is dynamically adapted to include more participants that arrive during service execution.

## 3 Communication models

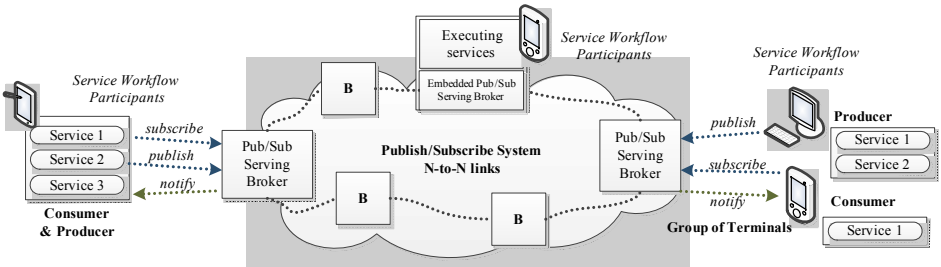
The proposed model for the communication level in order to support the service model is based on the publish/subscribe paradigm. Publish/Subscribe (Pub/Sub) systems are basically composed of three main components: publishers, which are the content producers; subscribers, which express their willingness to consume specific content; and brokers, which put in contact publishers and subscribers by storing and forwarding information.

### *3.1 Publish-subscribe communication model*

The use of a pub/sub-based communication model implies two advantages in these environments. First, the spatial and temporal decoupling provided by publish/subscribe enables the independence between event publishers and subscribers, which supports the arrival of new participants and the protection against connectivity losses. Second, a communication model based on notifications enables a straightforward transmission of events produced by services.

Figure 3 depicts our simplified model which is composed of a network of nodes that are interconnected through pub/sub serving brokers, which allows message event transmission from content publishers to subscribers. Serving brokers can be external entities, separated from workflow participants, or can be embedded into a client to avoid external infrastructure support. We define a client as an edge entity of the pub/sub network. From the subscriber/producer's side, our model leaves the typical primitives *publish*, *subscribe* and *notify* as simple as possible, and keeps the model complexity in services. Despite this, as brokers are location-independent they can also run together with producers and consumers, so it ensures the service execution even when there is no fixed pub/sub infrastructure.

**Figure 3** Communication model



### *3.2 Gossip-based protocols*

The problem of event transmission based on static routing is that, after a network fall, brokers wait for a network recovery to send the information packet. This is known as the busy waiting problem. To avoid this problem when recovering network operation after falls we have extended this communication model with the use of gossip protocols.

Gossip-based algorithms (Eugster et al., 2004) are a group of network protocols for information propagation in distributed systems. Gossip-based algorithm offers advantages (Birman, 2007) for environments where simplicity, scalability and convergent consistency are crucial, such as distributed services. Some gossip-based mechanisms exist (Song et al., 2009) for predicting service workflow; however, our solution not only limits to the workflow itself, but also how the whole event dissemination is adapted in runtime and the benefits of using cooperation mechanisms between distributed services and distributed pub/sub systems.

Event dissemination performed by the brokers is managed by the round time, which not only affects the dissemination rate but also the network saturation degree. Thus, a compromise is maintained depending on the network topology and the broker relationship with its neighbours.

The use of gossip protocols is justified for networks of unknown size in which we need to spread news robustly (Eugster et al., 2004); therefore, it fits perfectly in an environment with broker and node losses. It also supports a service model in which there are different versions of events. A gossip broker checks the version of the newly arrived event and if this version is equal or less that the processed event it discards this new event. Therefore, the information flowing through the network is always the most updated and it prevents that the arrival of outdated events causes unexpected behaviours in running services. In our model, we define that a service goes through a set of states. If the workflow has no loops (its representation graph is acyclic), the transition always occurs to new states and, therefore, each state can be associated with a version of events that are generated in the service. However, if the workflow returns to previous states the event version should be increased for each state change.

### *3.3 Node interoperability*

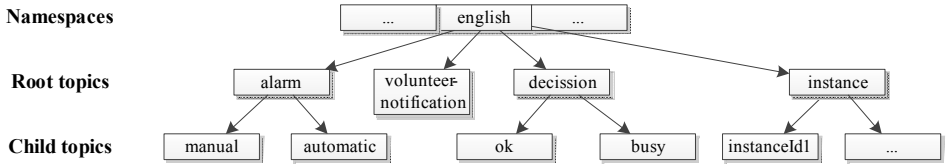
Interoperability between nodes is tackled in two levels. In the communication level, it is solved using the same pub/sub protocol. In our work, we use the Message Queue Telemetry Transport protocol (MQTT Protocol Specification, see http://public. dhe. ibm. com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html). Conceptually, the nodes must agree on the type of events that publish and consume. To do this we use a topic-based system (Eugster, 2007), in which messages are published to 'topics' and subscribers will receive all messages published to the topics to which they subscribe.

In order to agree on the used topics, we use a topic domain, shared by all entities, divided into namespaces. Topics are published in a namespace to ensure that notification messages are received in the appropriate language. In this way, we use the same topic structure to ensure that incompatible notifications are not received. Each topic in a topic namespace (*tns*) can have zero or more child topics, and a child topic can itself contain further child topics. A topic without a parent is termed as a root topic. We use the forward slash (/) character to indicate a 'child of' relationship. For example, the *tns1:alarm/manual* refers to the subtopic *manual*, subset of the parent topic *alarm*, in the namespace *tns1*.

This approach supports transformation and aggregation of topics, that is, it is possible to construct configurations (using intermediary brokers) where the topic subscribed by the subscriber differs from the topic published by the publisher, yet notifications from the publisher are routed to the subscriber by a broker that is acting according to administratively defined rules. For example, a subscriber to the topic *tns1:alarm* also receives notifications from topic *tns1:alarm/*automatic. In addition, it is possible for actors to define additional topics based on existing topics without requiring coordination with the actor responsible for creating the topics that are being built on. Our solution is compatible with the WS-topics OASIS standard (Vambenepe et al., 2006), which present a set of 'items of interest for subscription' in web service environments, and it has been extended to be aligned to a non-WS environment.

An example of a topic hierarchy for *home care assistance* service is shown in Figure 4. The *tns* chosen corresponds to the *English* language, to avoid language incompatibilities. A root node has been added that contains the identifiers of the service instances that are being executed, to avoid correlation problems, as described in Section 4.

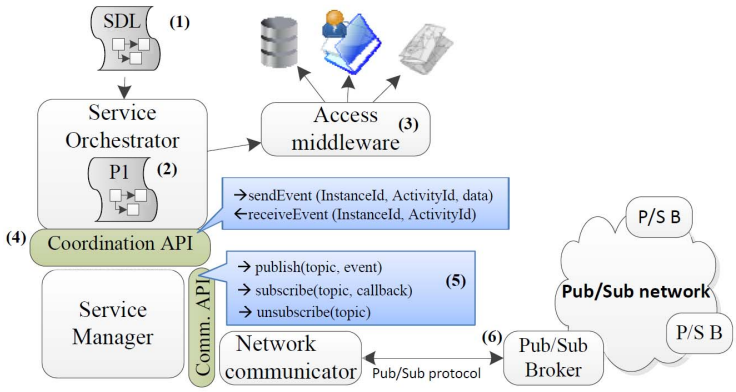
**Figure 4** Home care assistance service topic hierarchy




## 4 Architecture for cooperative mobile services

The proposed system is supported by the defined the terminal architecture, depicted in Figure 5, which is applied to each of the mobile devices participating in the service execution. We describe the process and the interaction between the main elements.

**Figure 5** Distributed architecture (see online version for colours)



Once the system receives the (1) SDL document, which contains information about event dependencies and agreed topics for communicating with other services, the (2) *Service Orchestrator* (SO) starts the service execution by invoking the functionality of the activities in the workflow. The middleware (3) is responsible for accessing the functionality described by the running activity (see Figure 5). It includes support libraries for local invocations to device capabilities (camera, contacts, etc.) and to remote objects (objects in the WoT, databases, web services, etc.).

When the orchestrator detects a limit activity means that a relation between the execution service and another one exists through this limit activity. This relationship is based on the exchange of events that are control mechanisms (used to start or resume the execution in other services) and also information between activities, such as properties or result from previous operations. The service orchestrator uses the *coordination API* (4) to invoke the sendEvent (*InstanceId*, *ActivityId*, *data*) method so that the *Service Manager* (SM) carries out the process of communicating the limit activity with their neighbours. The service orchestrator waits until the service manager asks to execute a new limit activity with the *receiveEvent* (*InstanceId*, *ActivityId*) method. To address these coordination challenges, the SM uses the *Network Communicator* (NC), which initiates the exchange of events between mobile devices at the network level, following the publish/subscribe paradigm. Using this paradigm is justified by the need for time decoupling (wherein the sender and receiver of a message do not need to be involved in the interaction at the same time) and space decoupling (wherein the messages are directed to a particular symbolic address or channel and not directly to the address of an endpoint), which enable the publication of data and control events to an unknown number of nodes in an unknown location.

We define *communication point* as each of the input and output information ports of each activity. Each communication point is associated with the event type it transmits and, according to the defined service model, with a single topic. This association is defined in the service's SDL and is interpreted by the SM. The communication between the SM and the NC is via the *communication API* (5), which includes the *publish* (*topic*, *event*) and *subscribe* (*topic*, *callback*) methods, used to publish (control or data) events that generates a given communication point and to receive events published by other devices in a callback method. The method *unsubscribe* (*topic*) is used to notify to the communication infrastructure that the node wants to leave the service execution and no longer wish to receive events.

The NC solves the correlation problem by adding a topic that identifies the service running instance to the basic topic. It uses the Full TopicExpression Dialect (Vambenepe et al., 2006), which contains XPath (Clark and DeRose, 1999) expressions to identify more than one Topic. We use the conjunction operator (&) to bind each event topic with an *instanceId*. The NC uses this composed topic for the pub/sub messages and transmits them to the *pub/sub broker* (6), which can be an internal or an external entity (not implemented in the mobile phone) that manages the subscription information necessary to deliver publish-subscribe messages. If the broker is embedded into the terminal architecture, the pub/sub communication protocol uses the *localhost* interface for communicating with the broker. Thus, we favour a low coupling and allow other modules to use the broker, such as the middleware module, in order to publish sensor data as in the work of Alcarria et al. (2012a), which describes a middleware that incorporates various communication paradigms, including pub/sub, for communication with external resources.

This architecture is replicated in each participant. When a connection to a participant is lost, the network communicator detects the disconnection and the serving pub/sub broker stores the packets until the node reconnects. Then, the broker sends to the node all the late packets. When the connection to a broker serving a set of nodes is lost, these nodes are out of service, but the messages they want to send are stored in the network communicator module. Also, messages directed to the offline broker are stored in the neighbour brokers, until an eventual reconnection. In the event that the offline broker belongs to a message route, the simplest approaches are to let the neighbour brokers wait until a broker reconnection to continue with the message routing or to manually redefine the route to avoid including this broker. Our solution considers a more complex approach, based on the use of gossip protocols, which take advantage of the dissemination mechanisms to find an alternative path by a controlled event flooding to avoid network congestion.

## 5 Managing distributed processes

The main contribution of this paper is how the SM resolves three coordination challenges. The first one is *service interaction and communication establishment*, assuming in this paper that communication between activities in the same service is resolved by the service orchestrator. To resolve the service interaction problem we define logic gates, corresponding to the most common workflow patterns (van der Aalst et al., 2003) and create optimised communication channels between logic gates. The second challenge is related to the *runtime coordination*, in which a contribution is performed by defining interactions between logic gates. Finally, a support for *arrival of new participants at runtime* is described by defining two participant aggregation modes.

### *5.1 Service interaction concepts and definitions*

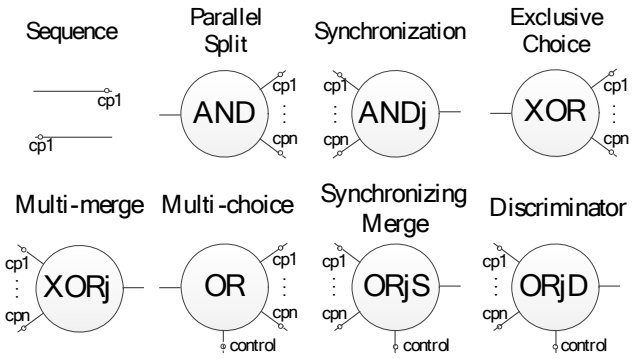
We use logic gates to enable communication between services, which can be seen as structured workflows. These logic gates follow the *workflow patterns model*, defined by van der Aalst et al., corresponding to basic control flow patterns and advanced branching and merging.

A logic gate LG is a tuple ($\Pi$, *type*, $\pi_{ctrl}$), where $\Pi$ is a set of publication ($\Pi_P$) or subscription ($\Pi_S$) *communication points*, *type* $\in$ {*publication*, *subscription*} indicates whether the logic gate is publisher (its communication points send a message to other services) or subscriber (its communication points receive a message from other services) of data and $\pi_{ctrl}$ represents a control point present only in some LG.

Figure 6 shows the existing communication points in each logic gate. The *sequence* (SEQ) pattern is modelled with a single communication point and an OR gate with *n* outputs is defined as $\Pi_{OR} \equiv \{\pi_1, \pi_2, \ldots, \pi_n, \pi_{ctrl}\}$.

We consider the *AND*, *XOR* and *OR* (activates all the branches, only one, or an empty or non-empty set of them, respectively) as publication gates and *ANDj*, *XORj*, *ORjS* and *ORjD* as subscription gates. *ANDj* transmits the execution when all branches have been activated and *XORj* for any activated branch. We define the *ORjS* and *ORjD* as logic gates with a control communication point connected to a previous *OR* gate to support the *structured synchronising merge* and the *structured discriminator* workflow patterns.

**Figure 6** Logic gates and associated patterns

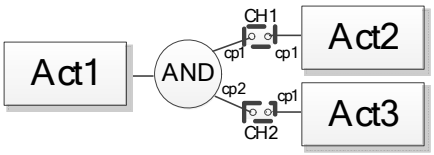



The structure between the *OR* and *ORjS/ORjD* gates is blocked until all the active branches are processed. *ORjS* transmits the execution when it receives the first branch activation and *ORjD* delays the transmission until all branches have been activated. The *SEQ* gate (transmits the branch activation) can be used for publication and subscription.

## *5.2 Channel creation*

Let $\alpha_1$ and $\alpha_2$ be two producer and consumer limit activities, respectively. We define the predecessor and successor functions such that $\alpha_1 = pre(\alpha_2)$ and $\alpha_2 = suc(\alpha_1)$. In order to connect these activities, it is needed to introduce a logic gate between them and create channels between the communication points of the activities and the logic gate, as shown in Figure 7. Thus, we associate each limit activity with a logic gate. We define channel as the tuple $(\pi_p, \pi_s)$, where $\pi_p$ and $\pi_s$ belong to the communication point set from a publication and subscription gate, respectively. Activities *Act*1, *Act*2 and *Act*3 are associated with the *AND* logic gate through channels *CH*1 and *CH*2 in Figure 7.

**Figure 7** Channel creation




At this stage, channel creation occurs by following the process illustrated in the pseudo-code of Algorithm 1. Let $A_{OR} \equiv \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be the set of consumer limit activities of a service. After scanning all the communication points of the subscriber logic gates of each activity (1) the identifiers of each communication point are retrieved (2) and used to look up the point of the assigned publication gate into the SDL document (3). After that, a callback address is created and bounded to the communication point (4), a channel is generated (9) and, finally, after retrieving the *topicId* from the share topic hierarchy (10), the subscribe method, from the communication API, is invoked (11).

**Algorithm 1** Channel creation and optimisation

Channel creation: $\forall \alpha \in A$

1: $\forall\ \pi_j \in \Pi_S$ from LG($\alpha$)

2: commPointId = getId($\pi_j$)

3: search in SDL associated $\pi_j \in \Pi_P$ from LG(suc($\alpha$))

4: create new *callback* = *f*($\pi_j$)

5: if: multiple $\Pi_i = \{\pi_{i1}, \pi_{i2}, \ldots, \pi_{in}\}$

6: set *ch*($\Pi_i$, $\pi_j$)

7: resolve *commPointId[]* in *topicId[]*

8: invoke *subscribe(topicId[], callback)*

9: else: set *ch*($\pi_i$, $\pi_j$)

10 resolve *commPointId* in *topicId*

11: invoke *subscribe(topicId, callback)*

By the channel creation process, all events generated by a service are linked to one or more services, according to the communication patterns defined by the considered logic gates.

However, the channel creation process is not so simple in the communication plane. By having a distributed architecture following a P2P model, in which each node knows only to communicate with its neighbour nodes, the discovery of publisher nodes becomes a difficult task. For this discovery process, the communication model based on a gossip solution introduced in Section 3 is used. Section 6 compares this solution with a standard technique based on recursive search of unvisited neighbours for the channel creation phase.

### *5.3 Channel optimisation*

If the output events of some communication points of a logic gate are equal (they share trigger conditions), it is possible to integrate multiple communication points in the same channel, avoiding generating additional channels (see lines 6, 7 and 8 in Algorithm 1). We define optimised channel as a tuple *ch*($\Pi_i$, $\pi_j$), where $\Pi_i$ is a subset of the whole communication point set of a publisher logic gate and $\pi_j$ is a communication point that belongs to the set of a subscriber logic gate. The degree of optimisation of a channel *O*(*ch*) is given by card($\Pi_i$), i.e. the number of communication points that compose the optimised channel. For example, for an AND = ($\Pi_{AND}$, *publicator*) gate, *O*(*ch*) is equal to the total number of communication points of the gate, as this gate replicates the same events in each output. This way, using a logic gate with an optimisation level of *O*(*ch*) means that the number of published events is reduced by *O*(*ch*)-1 (since all the communication points share the same pub/sub topic the network broker can use the multicast technique to forward a single event to all subscribers).

### *5.4 Runtime coordination*

At runtime, control events are transmitted through the created channels. Depending on the type of the logic gate involved in the channel formation the procedure varies.

For the subscription gates, in the case of *SEQ*, once the data are received from the established channel, the SM invokes the *startLimitActivity* method from the coordination API so that the service orchestrator executes the limit activity associated with the gate.

In the case of *ANDj*, the SC waits until all its branches receive events to contact the orchestrator. Regarding the *XORj* gate, the SC invokes *startLimitActivity* for each event received from the established channels. In the case of *ORjS*, to implement the *structure synchronising merge* pattern, the information from the $\pi_{ctrl}$ of a previous *OR* gate is used to determine how many branches the *OR* gate has activated. The SM waits for the control events in all activated branches and, when the last event arrives, asks the orchestrator to start the execution. If the previous gate is an AND, the SM knows that all branches are activated and waits for the arrival of the control event in all branches.

In the case of the *ORjD*, to implement the *structured discriminator*, the SM, using the information received from $\pi_{ctrl}$, routes the first control event and filters the events from the rest active branches.

For publication gates, in the case of *SEQ* and *AND*, the orchestrator invokes the *endActivity* (*InstanceId*, *ActivityId*, *result*) method from the coordination API when a limit activity completion event arrives; and the SM publishes the control event by all the communication points. In the case of *XOR* and *OR* gates, a decision is required to activate the branches, depending on the result values. Furthermore, the *OR* gate publishes the branch activation decision through the control port.

### *5.5 Managing participant arrivals at runtime*

The service model and the proposed architecture support the arrival and disconnection of new participants, even for running services. We define two aggregation modes, depending on the scope in which they occur: static scope and instance-based.

In the *static scope* aggregation, the participant and their services are added to the whole workflow for all instances of the composed service. Knowing the set of topics to be subscribed and the identification of executing service instances the network communicator is capable of establishing channels with deployed logic gates, by changing all *AND*, *OR* and *XOR* logic gates so that they acquire a new output branch, corresponding to this new service. These modified gates will be of the same type as before, except in the case of *SEQ* gate, which will become *AND*, as the transformation of a *SEQ* to an *OR* or *XOR* gates requires a branch activation condition that has not been specified. Regarding publication processes, a service arrival involves adding a new branch to the *ANDj*, *XORj*, *ORjD* and *ORjS* gates, and transforming a *SEQ* gate to a *XORj*, which transmits the execution for any activated branch. An example of participant aggregation in the static scope is when a participant executes a service with a different workflow from the workflows specified in the composed service. Figure 1 shows how the *send alarm* activity is linked to the *store alarm* activity through an *SEQ* gate, which will become a *XORj* gate with a number of input branches equivalent to the number of existing residences, so that this *XORj* gate transmits each of the alarms received, regardless of the instance in which they are located.

For the *instance-based* aggregation, the participant is integrated in a single instance of an executing service or his/her terminal creates the instance if this participant starts the service. In order to include this participant in the composed service execution, the other participants should be aware or the definition of new instances and the aggregation of these instances in the shared topic tree, since subscription messages should contain the

instance identifier to solve the correlation problem. We can find an example of instance-based aggregation also in the *home care assistance* service, but this time in the aggregation of volunteers. A new participant executing the *volunteer response* service requires creating a new instance, as volunteer responses are managed individually by the *control* service, waiting for confirmation that the volunteer can take care of the patient with problems.

Using the novel approach of managing workflow coordination by pub/sub models the publication or subscription to new topic is not needed when new participants arrive.

## 6 Prototype evaluation

We validate the service and communication models defined in this work by a qualitative and quantitative analysis of a set of services. We have developed the service architecture in a runtime environment in which we test the performance of our solution in the channel creation and runtime coordination phases. Besides, we define a situation of lack of network connectivity to compare the performance of the solutions that we call *standard pub/sub* and *gossip-based*.

Our goal is to check whether these two solutions for distributed service executions are able to detect changes in the pub/sub network topology, such as link failures and converge on a new loop-free routing path. We focus on robustness against link losses because it is a recurring problem in environments for the execution of mobile services, and the solution of this problem enables the transmission of event-based information between various participants executing services in mobility.

In these two solutions, we use an MQTT implementation in which we apply two levels of optimisation. The first one, *channel optimisation*, is related to the pub/sub model and specifically to the broker's capability of using the multicast technique to send a single publish to multiple subscribers of the same topic, i.e. the same channel (we have explained this in Section 5.3). The second optimisation level is called *multiple topic subscription* and enables the establishment of all channels associated with a service with a single MQTT *subscribe* message. MQTT makes possible the use of this technique.

### *6.1 Environment preparation*

We have implemented the described model and architecture using the MQTT (Message Queue Telemetry Transport) protocol, which is currently in process of standardisation. Three different environments have been defined.

*RealEnv* is a real-simple environment for coordination of distributed service in mobile devices. This environment consists of three android mobile phones (one Samsung Galaxy Note and two Google Nexus S), an MQTT client for android and a single open source message broker based on java called Moquette (Moquette Website) installed on a server with Core i7 2.80 GHz, 8 GB of RAM and Ubuntu 12.04 64 bits; all the devices connected through a WiFi 802.11g network. In *RealEnv* we performed a proof of concept, implementing the *home care assistance* service and distributing it among terminals. The *RealEnv* environment is considered to provide the most accurate values concerning execution delays. The gossip-based solution was programmed in Java 1.6. It has around 8000 lines of code, including the MQTT libraries, so it is enough suitable for current machines.

*SimulatedEnv* decouples the service and communication levels through the ns-3 simulation environment (Ns-3 simulator, see http://www.nsnamp.org). We consider a client and a broker with an MQTT support library that we have implemented (MQTT for ns-3 SourceForge Project, see https://sourceforge.net/projects/mqttforns3/). We use TCP as the transport level protocol. Communication between service and communication levels is achieved through the communication API, defined in the ns-3 environment to invoke pub/sub messages from an external application to ns-3, written in C. The advantage of *SimulatedEnv* is that it supports topologies with a large amount of nodes, so that it is used for scalability tests in the channel creation phase.

*VirtualEnv* is a scenario that combines the advantages of *RealEnv* and *SimulatedEnv*. It consists of two main parts: the first is the simulation within the ns-3 simulator (version 12.1) and the second is a set of *N* nodes implemented using a virtualisation technique called containers. Containers are created using the standard Linux utilities (LXC). In the configuration phase, we define parameters such as: ip address, operation mode (e.g. physical or virtual) and bridging. In the particular scenarios, it is necessary to define the communication interface in 'phys' (physical), as this allow us to execute pub/sub brokers with the same configuration as a real scenario but at the same time isolated at the kernel level.

We define all the network topology parameters using a script that defines the nodes, network interfaces, connections and applications. Even when this approach is more complex, it is also more flexible than hard-coding the network topology. It is necessary to define *TapBridge*-type devices in order to ensure that ns-3 and containers will communicate back and forth. In this case, we also set the corresponding physical interfaces for the respective containers. The operation mode of the *TapBridge* is ConfigureLocal. This operation mode allows to automatically create interfaces so *TabBridges* will inherit the same configuration. These devices are installed on ns-3 nodes, as ghost nodes, so they can communicate, through the same network interfaces, with ns-3 or applications that can run inside the container. Finally, in runtime the ns-3 creates the *TapBridge* and containers connect to them.

The performance of the pub/sub nodes in *VirtualEnv* is equivalent to *RealEnv* with the advantage of providing a large amount of containers. In our server, we were able to open 96 containers. Another advantage of this scenario is the possibility of using the monitoring and configuration features of ns-3 to simulate link losses. Figure 8 shows the architecture of this scenario in which we have two containers, which host a broker and a pub/sub node with the full architecture, similarly to *RealEnv*.

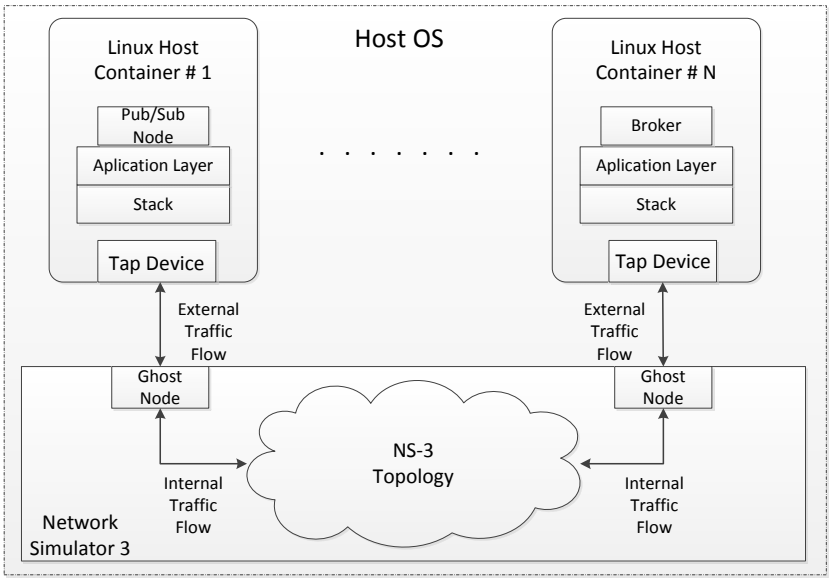
## *6.2 Qualitative analysis*

The decoupling between the service level (execution of distributed services) and the communication level allows us to separate the establishment of coordination mechanisms between services from the communications performance for the channel creation and runtime coordination processes. A communication model that supports both of these protocols is important to cover all situations in which each protocol offers their advantages. However, these situations and the improvement of the protocol in a given situation must be defined.

For an event-based service coordination environment with mobile terminals the most important factors to consider in a system are as follow:

- Using communication models that support event-based communication. Event-based systems differ markedly from request-based systems, since they are based on asynchronous 'push' messages, in a fire-and-forget, unidirectional pattern. Also, events are transmitted and communicated in the form of autonomous messages and they do not often require additional context and dependencies.
- Robust execution against link and device falls and capability to resume execution, as the work environment considers coverage problems and network link losses.
- Management of the arrival of new participants to the *composed* service execution and integration of new *individual* services.
- Level of decoupling between service and communication levels, which encapsulates internal details of distributed systems. This facilitates that the functional processes in the service layer, where the business event took place is less dependent on the availability and completion of communication processes in the distributed network.

**Figure 8** VirtualEnv scenario architecture




The contributions of this work in terms the service and communication models consider all these factors. Our proposal uses a pub/sub model to support event-based communication. The ability to resume execution after several situations of link, broker and node failures is described in Section 4 and the arrival of new participants is tackled in Section 5.5 and is validated in Section 6.3 by introducing link losses in the communication network. Finally, the decoupling between service and communication levels is achieved by the API defined in the architecture and is considered for the validation scenarios presented in the *environment preparation* section.

However, depending on the communication solution some of these aspects are best supported. Table 1 describes the suitability of each solution for each feature.

**Table 1** Qualitative comparison summary

| *Factors* | *Standard Pub/Sub solution* | *Gossip-based solution* |
|---|---|---|
| Event-based communication | Pub-sub based | Pub-sub based |
| Performance against falls | Active wait until recovery | Dissemination overcomes fall |
| New participants management | Low-performance path discovery for channel creation | High-performance path discovery for channel creation |
| Level of decoupling | Architecture based | Architecture based |

Regarding the performance against falls and the capability to resume the execution when the communication has been recovered, the defined standard pub/sub solution does not consider dynamic path rerouting, unless it is implemented over a routing protocol supporting changes in the network topology, such as OSPF (Open Shortest Path First). Thus, when a link or a broker that handles communications between two nodes suffers a disconnection the standard pub/sub solution remains waiting for the link or broker to reconnect. Instead, the gossip-based solution that we defined in this paper uses the dissemination mechanism to reach the termination point and always find the alternative route if it exists.

Regarding the support for the arrival of new participants, our solutions differ in how to perform channel creation between services of this new participant and those who are being executed on the other nodes. The gossip dissemination mechanism converges before a technique based on recursive search of unvisited neighbours implemented in the standard pub/sub solution for environments with a priori unknown network topology. Once the address and the serving broker of all network nodes are discovered, the pub/sub solution performs much better in terms of publication delay.

In summary, our gossip solution should be used to enable information sharing in networks of unknown size and also when a robust transmission is needed in environments with node and link disconnections. However, the standard pub/sub solution we defined offers more performance when the network topology is known and once the channel creation phase is produced.

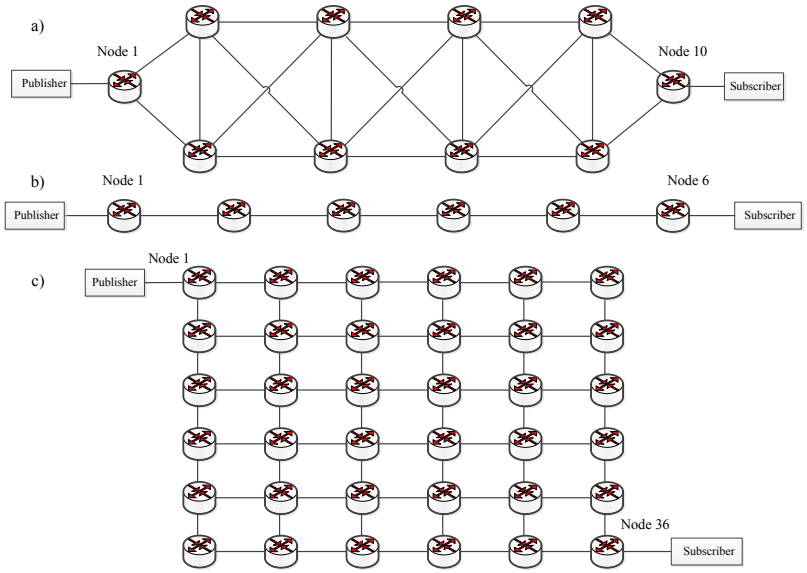
### *6.3 Performance evaluation*

The goal of this section is to measure the feasibility of our solution in our working environment. In previous section, we stated that the gossip mechanism performs better than simple MQTT for the channel creation phase in unknown networks. However, it also offered less performance once brokers determined the information regarding the best path from publishers to subscribers. To evaluate this, we consider the *SimulatedEnv* which is composed of a network of 36 connected brokers, which is depicted in Figure 9c. We consider node 1 runs a service that requires information produced by the node 36 and therefore needs to establish a channel thought the network.

We measured the time from the moment that node 1 invokes a receiveEvent, through its coordination API, until the *subscribe* message reaches the serving broker of node 36, and therefore updates the publication path. The MQTT specification does not define how to set the pub/sub paths among brokers, so we have set a simple recursive search and subscription mechanism for previously unvisited neighbours. This mechanism produces a 258 ms delay (~14.3 ms per hop) to the whole path, since a single message needs 18 hops to reach its destination. On the other hand, the gossip mechanism offers network

convergence at an average of 11 hops. In this case, the gossip has used 177 ms (~16 ms per hop). The variation over time between each jump is due to the complexity of spread mechanism in gossip. However, when the number of hops is fixed, there is only one path between the publisher and the subscriber, so the MQTT simple solution is 17% faster than gossip MQTT solution. We have verified this by using a topology of 6 nodes in line, each node with an integrated pub/sub broker, as shown in Figure 9b. In this case, we measure the time since the node 1 sends an event through its coordination API until receiveEvent method of node 6 returns its result. The results showed a delay of 72 ms in simple MQTT and 84 ms in gossip.

**Figure 9** Evaluated network topologies (see online version for colours)

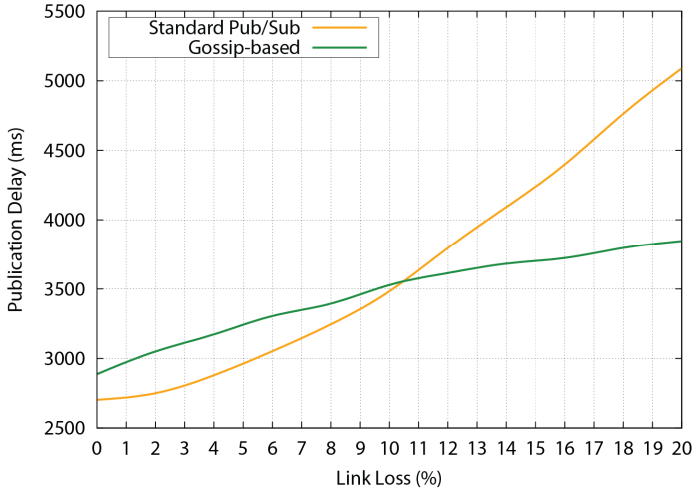



Once limit activities establish the communication channels, they start to exchange messages in runtime. We are interested in demonstrating that using gossip solutions is an accurate approach in environments with a high rate of failing links. For this task, we make use of the *VirtualEnv* environment where we have deployed 12 virtual containers, ten for brokers and two for two clients; clients are connected to broker 1 and broker 2, respectively, as shown in Figure 9a. Using the NS-3, we simulated that channels are likely to remain fallen over 1500 ms, so we send ten publications messages (with an interval of 500 ms) that will reach node 1 and finally node 10. We have also introduced a 500 ms delay of processing time in each broker, in order to make the failing link time proportional to the time publication message packets make through the network. Test results are shown in Figure 10. The percentage of link loss runs every 500 ms.

Our results show that the gossip dissemination protocol slowly improves the publication delay in comparison with MQTT simple. With a link loss around 10% the first solution offers an equal delay; however with a 20% of losses, gossip clearly outperforms MQTT simple. This is because, in these sort of topologies, even when there is a link failure in a broker that uses an optimal path, gossip takes advantage by sending the publication to a random peer, which could avoid current and successive link failures.

On the other hand, MQTT continuously verifies until the broker, that makes part of the optimal path, is reachable again so messages will finally get to subscribers. Despite this, we want to highlight the fact that the MQTT simple still performs better than gossip with low percentage of link loss.

**Figure 10** Publication delay comparison in environment with link losses (see online version for colours)




## 7 Related work

Related work tries to solve the problem of communications between mobile workflows from the IoT. Some works are related to the field of user/prosumer participation (Alcarria et al., 2012b). Generally, service communication is based on a data-driven approach, so that services can be created easily, with some composition (Gao et al., 2011) or mashup tools. Although there are some studies that combine data-driven composition with control flow specification (Rosenberg et al., 2008), we consider that the coordination between services based on the transmission control events (control-driven service composition) allows the execution of more complex cooperative services. The Presto framework (Giner et al., 2010) provides a service development platform for user participation in smart workflows, based on business processes. Our work also relies on user interaction with distributed elements and infrastructure of the WoT through their mobile devices, as it is proven their capability (Rodriguez et al., 2011) of being part of these scenarios. The work of Gómez-Goiri et al. (2011) shares similarities with ours, as they focus on the adoption of the triple spaces coordination language by heterogeneous and resource-constraint devices; and we focus on the problems of service coordination in distributed mobile service executions. We also find similarities in the field of decentralised service orchestrations (Jayaprakash et al., 2010) or choreographies (Fahland, et al., 2011a).

To manage this coordination, some authors (Tut and Edmon, 2002) propose the use of design patterns as reusable parts to compose services. In our work, we based on workflow patterns, specifically in the patterns defined by van der Aalst et al., to model

the connections between services (van der Aalst et al., 2005). van der Aalst et al. also point out the importance of unique identification of the elements of the process and the correlation problem (Fahland et al., 2011b), which we described in Section 3.

The interaction between service activities is often described in an SDL document, expressed in a standard language like BPEL or BPMN, or some other languages adapted from the service logic. Even though, the SDL should follow good design patterns (Mateos et al., 2011) because of the constrains of the mobile environment. In our work, we leave the door open to the possibility of using any service definition language compatible with the used workflow patterns for our SDL document. However, orchestration languages such as BPEL are not intended for distributed executions. Some distributed BPEL solutions exist (Khalaf et al., 2007) but they are based on providing WSDL interfaces to all services and they do not support the pub/sub communication pattern, especially important in pervasive and mobile applications.

The information exchange between coordinated services has been less addressed in related work. However, some proposals related to workflow decentralisation (Ranjan et al., 2008), task communication (Narayanan et al., 2011), and distributed orchestrations (Yildiz and Godart, 2007) have been found. Some authors (Yildiz and Godart, 2007) choose to solve the activity wiring using WSDL interfaces and SOAP messages. Other solutions use a tuple space (Ranjan et al., 2008) to manage the execution of scientific workflow applications by subscription/ notification methods. In other works (Nrayanan et al., 2011), virtual channels are used between sending and receiving tasks to ensure data communication.

In our work, we use the publish/subscribe communication paradigm (Eugster et al., 2003) as alternative to de-synchronise producers and consumers of information, and ensure functional decoupling in time and space (Costa et al., 2008). Pub-sub-based models can provide advantages (Fiege et al., 2006) over classic *polling*, which can overuse services and networks' resources by continuously querying information.

With regard to the communication support level, we highlight the works in reconfigurable communication middleware. The PLA middleware (Apel and Böhm, 2005) has been designed as a flexible and lightweight middleware for ubiquitous computing, aimed for mobile terminals. The main difference with our proposal, from the point of view of software engineering, is that they combine minimal fine-grained components and use a mixin layer approach (Smaragdakis and Batory, 2002) to tailor the architecture to fit in a specific scenario. MUSIC (Rouvoy et al., 2009) also extends a generic middleware, which seamlessly supports component-based and service-based configurations. The functionality provided by a component can be dynamically configured to adapt the framework to different environments. In our work, we develop a complete solution which focuses on distributed service execution environments and is designed specifically for it.

Integrating communication paradigms in access middleware has been tackled in the work of Morais and Elias (2010), proposing an architecture which supports the traditional synchronous model and different variations of the so-called asynchronous models. Other works, such as GREEN (Sivaharan et al., 2005), focus on the concept of reconfiguration in continuous execution environments, and provide a reconfigurable middleware (according to application requirements and context information) that supports pub/sub interaction types (topic-based, content-based and location-based) but only for one communication paradigm. Our previous work (Alcarria et al., 2012c) also focuses on

reconfiguration to access to unknown devices but it uses the dynamic bundle provision mechanism present in the OSGi (Open Services Gateway Initiative) platform to provide continuous service execution.

## 8 Conclusion and future work

This work defines a cooperative service execution model for mobile environments in scenarios from the IoT and WoT. In this model, user mobile and fixed devices execute services that access external resources and interact with the user. The need to coordinate these elements at the data plane (transfer of information produced by users or web objects to other terminals) and the control plane (synchronisation and management of the execution flow of tasks and activities) has been detected. This paper contributes to solve three coordination challenges detected in such environments. The interaction between services is resolved by introducing logic gates between limit activities, based on well-known workflow patterns. The channel creation and optimisation contribute to the communication establishment between limit activities, taking into account the arrival of new participants and the problems it arises in the communication layer. Finally, the runtime coordination is described by the interactions between the different modules of the defined architecture, in an environment with possible node and link loses. The validation of this work in real, simulated and virtual environments shows the acceptable performance of the *pub/sub standard* solution in the runtime coordination phase and the *gossip-based* solution for the channel creation phase and also for a scenario with a significant percentage of link failures (>10% of losses for the studied conditions).

As future work, in the field of coordination of distributed services, we will investigate automatic workflow partitioning mechanisms and user participation in the design or personalisation of the execution process of workflow activities, as an evolution of the concept of the prosumer user (Alcarria et al., 2012b). In the communication layer, we will investigate on pub-sub broker federation protocols to support service deployment in real environments with higher performance requirements.

## References

Alcarria, R., Robles, T., Morales, A., López-de-Ipiña, D. and Aguilera, U. (2012a) 'Enabling flexible and continuous capability invocation in mobile prosumer environments', *Sensors*, Vol. 12, No. 7, pp.8930–8954.

Alcarria, R., Robles, T., Morales, A. and González-Miranda S. (2012b) 'New service development method for prosumer environments', *Proceedings of the 6th International Conference on Digital Society*, Valencia, Spain, pp.86–91.

Alcarria, R., Robles, T., Morales, A. and González-Miranda S. (2012c) 'Flexible service composition based on bundle communication in OSGi', *TIIS*, Vol. 6, No. 3, pp.116–130.

Anjum, A., Hill, R., McClatchey, R., Bessis, N. and Branson, A. (2012) 'Gluing grids and clouds together: a service-oriented approach', *International Journal of Web and Grid Services*, Vol. 8, No. 3, pp.248–265.

Apel, S. and Böhm, K. (2005) 'Towards the development of ubiquitous middleware product lines', *Lecture Notes in Computer Science*, Vol. 3437, pp.137–153.

Birman, K. (2007) 'The promise, and limitations, of gossip protocols', *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 5, pp.8–13.

Clark, J. and DeRose, S. (1999) *Xml Path Language (xpath*), W3C standard. Available online at: http://www.w3.org/tr/xpath (accessed 16 October 2013).

Costa, P., Mascolo, C., Musolesi, M. and Picco, G.P. (2008) 'Socially-aware routing for publish-subscribe in delay-tolerant mobile ad hoc networks', *IEEE Journal on Selected Areas in Communications*, Vol. 26, No. 5, pp.748–760.

Daniel, F., Soi, S. and Casati, F. (2009) 'From mashup technologies to universal integration: search computing the imperative way', in Ceri, S. and Brambilla, M. (Eds): *Search Computing – Challenges and Directions*, Springer, Berlin Heidelberg, pp.72–93.

Eugster, P. (2007) 'Type-based publish/subscribe: concepts and experiences', *ACM Transaction on Programming Language Systems*, Vol. 29, No. 1, Article 6.

Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.M. (2003) 'The many faces of publish/subscribe', *ACM Computing Surveys*, Vol. 35, No. 2, pp.114–131.

Eugster, P.T., Guerraoui, R., Kermarrec, A-M. and Massoulie, L. (2004) 'Epidemic information dissemination in distributed systems', *Computer*, Vol. 37, No. 5, pp.60–67.

Fahland, D., de Leoni, M., van Dongen, B.F. and van der Aalst, W.M.P. (2011a) 'Many-to-many: some observations on interactions in artifact choreographies', *Proceedings of the ZEUS Conference*, Karlsruhe, Germany, pp.9–15.

Fahland, D., de Leoni, M., van Dongen, B.F. and van der Aalst, W.M.P. (2011b) 'Conformance checking of interacting processes with overlapping instances', *Proceedings of the 9th International Conference on Business Process Management*, Clermont-Ferrand, France, pp.345–361.

Fdhila, W., Dumas, M. and Godart, C. (2010) 'Optimized decentralization of composite web services', *6th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Luxembourg, Luxembourg, pp.1–10.

Fiege, L., Cilia, M., Muhl, G. and Buchmann, A. (2006) 'Publish-subscribe grows up: support for management, visibility control, and heterogeneity', *IEEE Internet Computing*, Vol. 10, No. 1, pp.48–55.

Gao, L., Urban, S. and Ramachandran, J. (2011) 'A survey of transactional issues for web service composition and recovery', *International Journal of Web and Grid Services*, Vol. 7, No. 4, pp.331–356.

Giner, P., Cetina, C., Fons, J. and Pelechano, V. (2010) 'Developing mobile workflow support in the internet of things', *IEEE Pervasive Computing*, Vol. 9, No. 2, pp.18–26.

Gómez-Goiri, A., Orduña, P., Ausín, D., Emaldi, M. and López-de-Ipiña, D. (2011) 'Collaboration of sensors and actuators through triple spaces', *IEEE Sensors 2011*, pp.651–654.

Jayaprakash, M., Shanmugam, M., Manikandan, P. and Shivaraj, S. (2010) 'Decentralized service orchestration by continuous message passing', *International Journal on Computer Science and Engineering*, Vol. 2, No. 5, pp.1627–1632.

Khalaf, R., Kopp, O. and Leymann, F. (2007) 'Maintaining data dependencies across BPEL process fragments', *Proceedings of the 5th International Conference on Service-Oriented Computing*, Vienna, Austria, 17–20 September, pp.207–219.

Mateos, C., Crasso, M., Zunino, A. and Ordiales, J. (2011) 'Detecting WSDL bad practices in code-first web services', *International Journal of Web and Grid Services*, Vol. 7, No. 4, pp.357–387.

Moquette Website (2013) *Java Small MQTT Broker Implementation*. Available online at: http://code.google.com/p/moquette-mqtt/ (accessed on 16 October 2013).

Morais, Y. and Elias, G. (2010) 'Integrating communication paradigms in a mobile middleware product line', *Proceedings of the 9th International Conference on Networks*, Menuires, France, 11–16 April, pp.255–261.

Narayanan, S., Devaux, L., Chillet, D., Pillement, S. and Sourdis, I. (2011) 'Communication service for hardware tasks executed on dynamic and partial reconfigurable resources', *Proceedings of the IEEE/IFIP 19th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, Kowloon, Hong Kong, pp.196–199.

Ranjan, R., Rahman, M. and Buyya R. (2008) 'A decentralized and cooperative workflow scheduling algorithm', *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, Lyon, France pp.1–8.

Rodriguez, J., Zunino, A. and Campo, M. (2011) 'Introducing mobile devices into grid systems: a survey', *International Journal of Web and Grid Services*, Vol. 7, No. 1, pp.1–40.

Rosenberg, F., Curbera, F., Duftler, M.J. and Khalaf, R. (2008) 'Composing RESTful services and collaborative workflows: a lightweight approach', *IEEE Internet Computing*, Vol. 12, No. 5, pp.24–31.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A. and Scholz, U. (2009) 'MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments', *Software Engineering for Self-Adaptive Systems*, Vol. 5525, pp.164–182.

Sivaharan, T., Blair, G. and Coulson, G. (2005) 'GREEN: a configurable and re-configurable publish-subscribe middleware for pervasive computing', *Lecture Notes in Computer Science*, Vol. 3760, pp.732–749.

Smaragdakis, Y. and Batory, D. (2002) 'Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs', *ACM Transactions on Software Engineering*, Vol. 11, No. 2, pp.215–225.

Song, W., Jiang, D., Chi, C., Jia, P., Zhou, X. and Zou, G. (2009) 'Gossip-based workload prediction and process model for composite workflow service', *World Conference on Services*, Los Angeles, CA, USA, pp.607–614.

Tut, M.T. and Edmond, D. (2002) 'The use of patterns in service composition', *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*, Toronto, Canada, pp.28–40.

Vambenepe, W., Graham, S. and Niblett, P. (2006) *Web Services Topics 1.3 (WS-Topics)*, OASIS Standard. Available online at: http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf (accessed on 16 October 2013).

van der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B. and Barros, A.P. (2003) 'Workflow patterns', *Distributed Parallel Databases*, Vol. 14, No. 1, pp.5–51.

van der Aalst, W.M.P. and Ter Hofstede, A.H.M. (2005) 'YAWL: yet another workflow language', *Information Systems*, Vol. 30, No. 4, pp.245–275.

Wang, S., Sun, Q. and Yang, F. (2010) 'Towards web service selection based on QoS estimation', *International Journal of Web and Grid Services*, Vol. 6, No. 4, pp.424–443.

Yildiz, U. and Godart, C. (2007) 'Centralized versus decentralized conversation-based orchestrations', 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, Hangzhou, China, pp.289–296.