AN EXTENSIBLE, MULTI-PARADIGM MESSAGE-ORIENTED MOBILE MIDDLEWARE

Yuri Morais and Glêdson Elias
Component Oriented Software Engineering Group, COMPOSE, Informatics Department
Federal University of Paraiba, Paraiba, Brazil

Keywords: Message-oriented Middleware, Mobile Computing, Communication Paradigm, Software Product Line.

Abstract: Message-oriented middleware (MOM) platforms are usually based in asynchronous, peer-to-peer interaction styles, leading to more loosely coupled architectures. As a consequence, MOMs have the potential for supporting the development of networked mobile applications. However, MOM platforms have been implemented under a limited set of message-based communication paradigms, each one being specifically adapted to a given application domain or network model. In such a context, this paper proposes a mobile middleware solution which offers a comprehensive set of extensible, message-based communication paradigms, such as publish/subscribe, message queue and tuple spaces. Supported by a Software Product Line (SPL) approach, the proposed middleware is suitable for constrained devices as all supported communication paradigms share and reuse a reasonable number of software components that deal with common messaging features. Additionally, by means of an extensible design, new communication paradigms can be easily accommodated, as well as existing ones can be removed in order to better fit in more constrained devices.

1 INTRODUCTION

Ubiquitous computing, context-aware applications and mobile services form one of the most promising business opportunities in the near future. Ubiquitous applications, however, introduce great challenges to software developers, such as mobile nodes, scarce resources and fragile wireless links. In such a scenario, middleware plays a key role as it aims at facilitating communication and coordination of distributed components, concealing difficulties and complexity raised by mobility to software engineers as much as possible. However, the traditional client-server, synchronous model used by traditional middleware (e.g. RPC, RMI) is not adequate for mobile environments. Instead, an asynchronous, peer-to-peer style fits better for mobile computing, as it leads to a loosely coupled architecture. Such a communication style is provided by message-oriented middlewares (MOMs). In MOMs, instead of direct method invocations, communication is accomplished by producing and consuming messages, which are generic data structures created by applications in order to transmit data.

MOMs have been implemented under a limited set of message-based communication paradigms, each one being specifically adapted to a given application domain or network model. Examples of MOM paradigms include publish/subscribe, tuple spaces, message queues and notifications. In all of them, communication is always done by means of message exchanges. The main aspects that differentiate them are their respective message delivery semantics: (i) consumers are asynchronously notified or explicitly retrieve new messages; (ii) messages are handled by intermediate nodes or directly addressed to consumers; and (iii) messages are consumed according to a set of properties or in a predefined order.

Taking into account the diversity of communication paradigms, one of the main problems in developing middleware for distributed mobile applications is to decide which paradigm is the best option to be adopted. Each paradigm presents particularities and focuses on different scenarios. Therefore, according to (West, 2005) and (Costa, 2005), modern middleware platforms should not be limited to a specific communication and design paradigm, instead, they must be flexible

Morais Y. and Elias G. (2010).
AN EXTENSIBLE, MULTI-PARADIGM MESSAGE-ORIENTED MOBILE MIDDLEWARE.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 158-164
DOI: 10.5220/0003011501580164
Copyright © SciTePress

enough to attend a wide variety of application domains. Moreover, by supporting varied options of communication paradigms, middleware platforms are more likely to be reused in distributed applications than a solution solely based on a specific communication paradigm.

In such a context, this paper proposes a message-oriented middleware solution for mobile computing that offers a comprehensive and extensible set of communication paradigms. In order to optimize resource utilization in constrained devices, all supported communication paradigms share and reuse a reasonable number of software components that deal with common messaging features.

In addition, supported by a reusable and extensible architectural design, the proposed middleware has been conceived based on a Software Product Line (SPL) approach. The architectural design permits the selection of supported communication paradigms during deployment time, enabling paradigms to be included or removed, as well as creating new ones by filling in predefined extension points, without modifying existing code. Thereby, the middleware can be customized according to application requirements and device capabilities, and besides it is also able to accommodate new features.

The middleware was initially proposed in (Morais, 2010), by the definition of a preliminary high-level architecture. In this paper, however, we introduce the idea of an extensible design, along with implementation details.

The remainder of this paper is structured as follows. Section 2 presents an overview of the proposed mobile middleware solution. In Section 3, the architecture and design to support the proposed features are presented. Section 4 describes implementation details. Section 5 discusses related work, and, finally, Section 6 presents concluding remarks and future directions.

2 PROPOSED MIDDLEWARE

MOM was initially proposed for dealing with communication issues in enterprise information systems due to their versatility and robustness (Eugster, 2003). Recently, their asynchronous, peer-to-peer and uncoupled interaction style, along with their fault-tolerance mechanisms showed to be the most adequate way of providing communication in mobile, wireless environments. MOM's main goal is to hide all the nasty communications from applications by providing a simple high-level API for distributed programming.

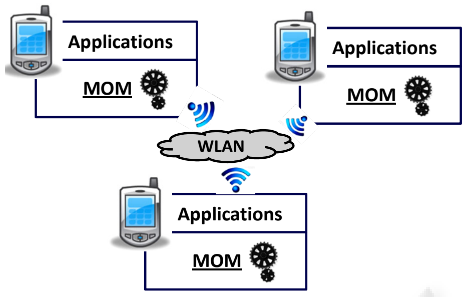

Figure 1: Middleware operating environment.

The services offered by the proposed middleware are intended to support mobile applications that need to communicate with other ones in a local wireless network, in an ad hoc fashion or connected by means of an access point. Examples of such applications include: cooperative games, file sharing, mobile indoor guide, military battlefield, and disseminating information (e.g., advertisement, share prices) on a trade floor. Figure 1 shows the environment in which the middleware must operate.

Such an environment is based on a peer-to-peer model, in which each node can act as message producer and/or consumer. There is no distinction between server and client nodes, some of the server functionalities (i.e. persistence, service advertisement) are embedded in each node.

Moreover, instead of a monolithic architecture, the middleware design is based on the product-line concept. According to (Clements, 2002), a software product line is a set of systems sharing a common, managed set of features that satisfy the specific needs of a market segment and that are developed from a common set of assets in a prescribed way.

In such a sense, the middleware architecture defines common messaging features that are shared among the communication paradigms, forming the product line kernel. On the other hand, messaging semantics specific for each communication paradigm are modelled as optional components. Thereby, such optional components can be part or not from each instance of the middleware.

The middleware message model has the following goals: (i) support several communication paradigms in a unified, extensible and customizable platform; (ii) enable the development of applications with different distributed interaction dynamics.

2.1 Messaging Common Features

Although the MOM interaction style suits well in mobile, wireless scenarios, according to (Mascolo, 2002), traditional monolithic MOM implementations are not enough. Messaging features must be adapted in order to deal with restrictions imposed by mobile devices and wireless networks. Thus, the proposed middleware adopts the following adaptations.

Sending Buffer. Taking into account the frequent disconnections of mobile networks, this mechanism temporarily stores messages in a local buffer while waiting to be transmitted when connection is again available. Such mechanism enables the sender application to 'fire-and-forget' messages, trusting the middleware to handle the delivery.

Message Time-to-Live (TTL). Despite local buffers being an efficient way of dealing with disconnections, long disconnection periods can overload the device memory, which is usually very constrained. Thus, applications must specify a message time-to-live (TTL), which defines the time that a message can be kept by the middleware while waiting to be transmitted.

Persistent Messages. Although the sending buffer provides an exactly-once delivery guarantee, in cases of abnormal conditions, such as resource-exhaustion, processing failure or device being turned off due to low battery, messages managed by a device may be lost. The proposed middleware mitigates such an issue by allowing messages to be marked as persistent and thus to be stored in a non-volatile memory.

Service Location. Traditional MOM providers, such as (Sun Microsystems, 2002), use a fixed, predefined lookup service in order to transparently find available services. However, in mobile networks, it cannot always be assumed the existence of a central entity to lookup services from. Due to that, the proposed middleware advertise its services in the network by means of a service discovery protocol, similar to (Rellermeyer, 2010). In order to look up services, the middleware sends a multicast message to the network. Devices providing services listen to that multicast address and reply in unicast indicating their service addresses.

2.3 Messaging Styles

The messaging features presented in the previous section consist in common facilities for dealing with issues in mobility scenarios. Such features form the basis for providing messaging exchanges in a variety of message-based communication paradigms. In order to provide multiple communication paradigms, only the code specifically related to message delivery semantics must be added. This subsection specifies the communication paradigms considered in this proposal and discusses design decisions.

Message Queue Paradigm. Message queuing is based on the adoption of distributed queues, where several producers send messages to a queue and then various consumers retrieve them from the queue. In contrast to traditional MOM platforms, in which queues are created only by the system administrator, queues in the proposed middleware are created dynamically and on demand by applications.

By following the peer-to-peer style, message queues can be hosted in any device of the network and applications can access local or remote queues. Additionally, the time interval that messages are held is queues is also controlled in order to avoid resource exhaustion. Traditional enterprise message queue systems usually store messages indefinitely. However, mobile devices have storage resources very limited. Therefore, the message TTL defines how long a message can spend in a queue until being read or, otherwise, discarded.

Tuple Space Paradigm. The notion of tuple space was originally proposed in Linda (Gelernter, 1985) as a coordination language for concurrent programming, however, its uncoupled and opportunistic style of communication has been shown to provide many useful facilities for communication in mobile environments. Tuple spaces provide a simple and powerful abstraction for accessing a shared memory, which acts as a repository of data structures for distributed applications communicating by means of the insertion and removal of tuples.

Each tuple has a set of application defined properties which are used in retrieval operations and can differentiate each one from the others. The retrieval of tuples is by means of content association, that is, by comparing a set of properties informed as parameter and the properties of the tuples in the space. A tuple space service can be created in any device and serve as a mean to allow distributed applications to communicate. Furthermore, similar to message queues, tuple spaces services are created on demand, by applications.

Publish/Subscribe Paradigm. In this paradigm, communication is achieved by publishing events and subscribing in certain types of events. An intermediate service, called event channel, records subscriptions and forwards events produced by publishers for all interested subscribers. Each event channel is identified by a topic service name, which specifies a group of similar applications exchanging messages.

When an application creates an event channel, the middleware takes care of advertising it, keeping subscriptions and forwarding messages as they are published. Applications can send or subscribe to event channels hosted by their own device or in other devices of the network. The location is transparent, that is, applications obtain a reference to the service by means of the service locator. In order to publish messages, applications may create its own topic or locate an existing one in the network. It is important to highlight that if an event channel has no connection with a given subscriber, the middleware takes care of retransmitting the message later on.

Synchronous Paradigm. Differently from the previous paradigms, in the synchronous paradigm, each message is explicitly addressed to a specific application, without using an intermediate service, such as an event channel or tuple space. In addition, this paradigm adopts the client/server model, in which the client application sends a request message to the server application, and then, the latter sends back a response message to the former. Similarly to RPC, the client side blocks while waiting for a response. The difference is that, instead of directly invoking a remote operation, a message object is used to exchange information.

Point-to-Point Notification Paradigm. Similarly to the synchronous model, this paradigm has no intermediate service. It differentiates from the previous model in that instead of a request-response model, it uses one-way invocations. Moreover, it may be used to split a synchronous remote invocation into two asynchronous invocations: the first one triggered by the client to the server application – accompanied by the invocation arguments and a callback reference to the client – and the second one triggered by the server to return the response message to the client.

3 EXTENSIBLE DESIGN

The architecture of the proposed middleware exploits the common aspects of the communication paradigm, previously explained, and represents as mandatory, shared components all features that are common among the messaging styles. On the other hand, the code related to each message delivery semantic, and so specific for each communication paradigm, is modelled as variable, optional. Figure 2 illustrates the architecture designed to support the proposed solution. The optional components are those that implement the communication paradigms, represented in Figure 2 inside the dot-dashed box.

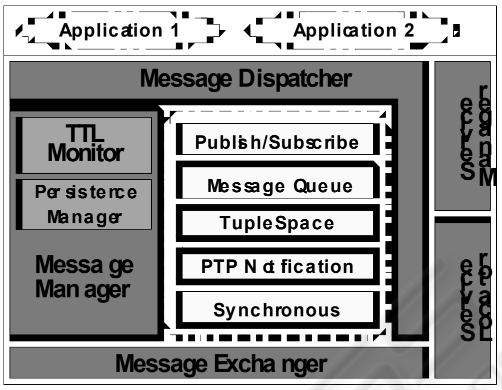

Figure 2: Middleware architecture.

The *message dispatcher* component provides the main API (Application Programming Interface) for applications to interact with the middleware. It is responsible for directing the called operations for the specific components that implement the requested communication paradigms. The *message dispatcher* is further explained in the next sections.

The creation and discovery of services is accomplished by the *service manager* and *service locator* components. While the former offers and API to applications and registers the services internally with communication paradigm components (e.g. the *tuple space* component), the former is responsible for dealing with network related issues, such as sending and replying multicast messages.

The central functional entity of the architecture is the *message manager* component. Internally, it implements and manages a single data structure that stores all messages handled by the middleware. For each message, the *message manager* knows the associated communication paradigm. Thus, any component that implements a given communication paradigm (e.g. the *message queue* component) does not need to implement a data structure for managing its handled messages. Instead, such messages are directly retrieved from the *message manager*.

Such a central functional entity prevents having to manage different data structures for each paradigm or maybe for each application using the middleware. Thus, each component that supports a given communication paradigm has only executable code for dealing with its specific delivery semantic, and not code related to message storage.

As illustrated in Figure 2, the *message manager* is a composite component. Whenever it receives a message marked as persistent, it calls the *persistence manager* component for storing the message in a non-volatile storage. Besides, it also instructs the *TTL monitor* component for constantly monitoring if such a message is still valid or should be discarded.

The *message exchanger* implements the communication among devices by means of TCP sockets. Both the *message exchanger* and the *service locator* are the unique components that interact directly with network specific functionalities, such as sockets connections.

3.1 Structural Design

The proposed middleware exploits the open-closed design principle (Clements, 2002). By following such a principle, a set of base classes deals with commonalities among paradigms, while the particularities of each paradigm are modelled as extensions from the set of base classes. Therefore, the design is closed for modifications but open for extensions. That means, in one hand, that we can make the middleware behave in new and different ways (add new paradigms), just by extending the base classes. On the other hand, future extensions do not cause modification in the existing code. In such a direction, a set of extension points are defined in the middleware design. In order to add a new paradigm, the extension points must be fulfilled, but no other part of the code must be changed.

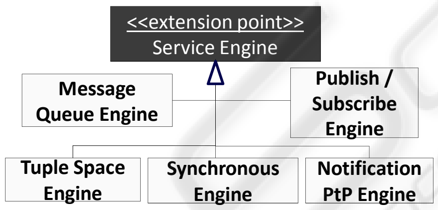

Figure 3: ServiceEngine extensions.

Service Engine. The *ServiceEngine* class actually implements the delivery semantic of the respective communication paradigm. By inheritance, it acts as an extension point, as shown in Figure 3. Each extended class defines specific operations for the respective paradigm. For example, the *PublishSubscribeEngine* registers the topics created by applications and subscribes applications in each topic. In the case of the *TupleSpaceEngine*, the semantics for retrieving messages based on comparing properties are defined, while the *MessageQueueEngine* defines semantics for retrieving on a first-in, first-out basis. Engine classes are not visible at the application-side, instead, they interact with internal middleware components only.

Service Session. The *ServiceSession* class is an application-side class that defines operations for interacting with middleware services. Also acting as an extension point, each extended class defines specific operations for the respective paradigm. For example, to take messages from a tuple space, the *TupleSpaceSession* defines the *take()* method passing as parameter the set of properties. Differently, the *PublishSubscribeSession* has the *subscribe()* method for registering in a given topic.

Service Administration. The *ServiceAdmin* class is also an application-side class that allows managing the associated service. For instance, in a publish/subscribe service, its instance can be called to inactivate the service or to identify the current subscribers. Additionally, in a messaging queue service, the number of messages in the queue can be monitored. Note that an instance of such class is only obtained by the application that creates the service, that is, applications that are only clients of a service cannot call administration operations. Like the *ServiceEngine* and *ServiceSession* classes, *ServiceAdmin* is also an extension point. Thus, each paradigm must define a subclass of ServiceAdmin.

To sum up, supporting a new communication paradigm is a matter of extending the aforementioned classes. Similarly, if an existing paradigm must be removed for optimizing middleware footprint, removing its related classes from the build is enough. Such flexibility is achieved by adopting a naming convention for all classes that implements extension points. Each class of a given paradigm must be named as follows: *paradigm name + extension point name*. For example, tuple space related classes must be *TupleSpaceSession*, *TupleSpaceEngine* and *TupleSpaceAdmin*.

3.2 Behavioural Design

In order to support several communication paradigms in an extensible way, the common, shared components of the proposed middleware must not be coupled to the particularities of any paradigm. Instead, reflection mechanisms are used to dynamically discover which extension class must be called. Figure 4 shows an UML sequence diagram for illustrating the interactions for creating a session with a communication service.

First, the application discovers the existing services by invoking *findServices()* in the *service manager,* which in turn calls the same method of the *service locator*, returning the set of references to currently available services (*ServiceReference[]*).

Service references make transparent to applications all information required to access a service, such as host, port, service type and name. Then, the application calls *createSession()* in the selected service, which in turn invokes the same method of the *message dispatcher*, returning the *service session* instance, according to the type of the service.

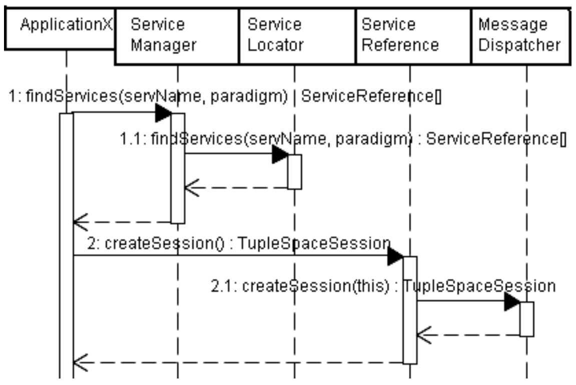
ApplicationX
Service Manager
Service Locator
Service Reference
Message Dispatcher
1: findServices(servName, paradigm) : ServiceReference[]
1.1: findServices(servName, paradigm) : ServiceReference[]
2: createSession() : TupleSpaceSession
2.1: createSession(this) : TupleSpaceSession

Figure 4: Obtaining references to communication services.

The type of the *service session* instance is discovered dynamically by reflection. For example, the *message dispatcher* identifies that the calling service reference represents a tuple space service, and thus, using the naming convention adopted for extension points, it knows that a *ServiceSession* subtype named *TupleSpaceSession* must be returned.

Similar to an HTTP cookie, internally, a session ID is attributed to that session by the *message dispatcher.* Now, in the following invocations to that tuple space service, the middleware knows that the message comes from a *TupleSpaceSession* and therefore must be treated by the *TupleSpaceEngine*.

Although reflection causes additional overhead, the use of sessions tends to soften such issue. That is, reflection is performed only when an application obtains the reference to a service at the first time. The subsequent calls to that service are identified by a session ID, which does not adopts reflection.

3.3 Implementation

In order to validate the proposed architecture, the prototype has been implemented on top of the Android platform (Android, 2010). Android is optimized to allow running one virtual machine per application. In addition, Android defines a component model that provides background processing as “services”. In such a context, the middleware runs as a background service, which can be accessed by applications through the supported inter-process communication (IPC) technology. Thus, it isolates the code of the middleware and applications, enabling several user-interactive applications to share a single middleware instance.

The source code of the middleware prototype totalizes 2094 lines, in which 644 lines are related to the extension classes that implement the supported communication paradigms. That is, in a middleware instance providing 5 communication paradigms, 69% of code is shared and reused among such paradigms. The middleware deployment unit is bundled into an Android package totalizing 92 KB. Considering that Android phones have non-volatile storage measured in gigabytes and average RAM of 256 MB (GSM Arena, 2010), it can be considered an excellent footprint.

4 RELATED WORK

MobileMom (Jung, 1999) provides asynchronous communication for mobile applications by means of distributed message queues. However, MobileMom does not support ad hoc scenarios, as it relies on fixed hosts. Additionally, it supports only a single paradigm (i.e. message queue), while the proposed middleware supports multiple paradigms.

Java Message Service (JMS) (Sun Microsystems, 2002) is a standard for enterprise-messaging systems which provides two messaging models: message queuing and publish/subscribe. However, its implementations are based on a central, resource full server, which is not always available on mobile networks. In addition, JMS queues and topics are created administratively, which is not adequate for dynamic scenarios. Oppositely, the proposed middleware enables such services to be dynamically created by applications.

On the other hand, (Vollset, 2003) describes the design and implementation of a JMS solution for mobile ad hoc networks (MANETs). However, it requires all clients to have a local copy of an identical configuration file in order to access remote topics and queues. In the proposed middleware, however, it is adopted a service discovery protocol, enabling devices to lookup messaging services in any unknown network.

RUNES (Costa, 2005) is a middleware for embedded systems that supports an extensible set of communication paradigms, such as tuple spaces, publish-subscribe and remote procedure call. Whereas RUNES enables communication paradigms to be plugged, common facilities to be reused among different paradigms are not defined.

5 CONCLUDING REMARKS

This paper has presented a MOM platform to support communication among distributed mobile applications. As an important contribution, the middleware supports multiple paradigms in an integrated architecture, in which common messaging features are shared among distinct communication paradigms. Such integration enables constrained devices to support various communication paradigms by demanding much less resources.

Additionally, the product-line extensible design enables features customization, that is, existing paradigms can be removed in order to better fit in more constrained devices, as well as future extensions and new behaviours can be easily accommodated. It is important to note that such customization does not require existing code modification, instead, it is done by adding/removing a set of classes on predefined extension points.

As a future work, it is planned a performance evaluation for measuring the cost of using reflection and IPC, and so, comparing their benefits against the added overhead. In addition, it is also important to measure the runtime memory footprint.

ACKNOWLEDGEMENTS

This work was supported by the National Institute of Science and Technology for Software Engineering (INES)[1], funded by CNPq, grants 573964/2008-4.

REFERENCES

Rellermeyer, J.S., 2010. jSLP project, Java Service Location Protocol. http://jslp.sourceforge.net.

GSM Arena, 2010. http://www.gsmarena.com.

Costa, P. *et al*., 2005. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In: *Proc. of IEEE 16th Int. Symp. on Personal, Indoor and Mobile Radio Communications*.

Wuest, B., 2005. Framework for Middleware Executed on Mobile Devices. Master's Thesis. Informatik der Universität Kassel.

Jung, D., Paek, K., and Kim, T., 1999. Design of MOBILE MOM: Message Oriented Middleware Service for Mobile Computing. In: Proc. of the International Workshops on Parallel Processing.

Morais, Y., Elias, G., 2010. Integrating Communication Paradigms in a Mobile Middleware Product Line. In: *9th Int. Conf. on Networks.*

Clements, P., and Northrop, L., 2002. *Software Product Lines: Practices and Patterns*. SEI Series.

Sun Microsystems, 2002. Java Message Service Specification. Version 1.1.

Mascolo, C., Capra, L., and Emmerich, W., 2002. Middleware for Mobile Computing. Tutorial. In: *Proc. of the Int. Conf. on Networking*.

Gelernter, D. 1985. Generative Communication in Linda. In: *ACM Trans. on Prog. Languages and Systems*.

Eugster, P. *et al*., 2003. The Many Faces Of Publish/Subscribe. In: *ACM Computing Surveys*, Vol. 35, No. 2, June 2003, pp. 114–131.

Android, 2010. http://developer.android.com.

Vollset, E., Ingham, D., and Ezhilchelvan, P., 2003. JMS on Mobile Ad-hoc Networks, LNCS 2775, pp. 40-52, Springer.

[1] wwww.ines.org.br