Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2014) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 9th International Conference on Future Networks and Communications (FNC-2014)

Universal Connector Framework for Pervasive Computing using Cloud Technologies

Hamid Mcheick*, Louis Deladiennee, Mickael Wajnberg, Benoit Martin, and Marc Abi-Khalil

University of Quebec at Chiucoutimi, 555 Boul De l'Universite, Chicoutimi, G7H-2B1, Canada

Abstract

Today, software architecture has a vital role in achieving quality goals for large scale distributed software systems which is made up of components and connectors. Especially, software development mainly consists of composing re-usable components. The main issue in this approach resides in the difficulty to make these heterogeneous components communicate with each other smoothly, especially in mobile and pervasive computing area. These components can be replaced, dropped or added easily using injection of control mechanism (IoC). In this work, we propose and validate a universal connector framework that allows, through a web service interface using cloud technologies, diverse software components and mobile devices communicate with each other. For example, components and devices in pervasive systems can exchange context information, and analyze them to deduce new rules. As a proof of concept, we apply this connector in a mobile messenger context and shows our results.

Peer-review under responsibility of the Program Chairs of FNC-2014.

Keywords: Pervasive Computing; Design Connector; Mobile Applications; Cloud Computing

1. Introduction

Distributed Systems (DS) take advantage of the computing powers in components that exist on several networked devices. Today's electronic device market is no more restricted to a desktop computer but by many different mobile

* Hamid Mcheick. Tel.: +1-418-545-5011; fax: +1-4185455017.
E-mail address: Hamid_Mcheick@uqac.ca

Peer-review under responsibility of the Program Chairs of FNC-2014.

devices such as tablets, smartphones and even cars. In terms of architecture, connector and component are two key concepts to understand logical organization of software [1].

Connectors are regarded as software elements for delivering data and control in a software system [8]. They are the software module for describing the relationships of components and providing a channel to link components together [9] [10]. Especially, the connectors play a significant role in distributed architecture.

In general, components in DS are located on different devices and require sophisticated software connectors that can orchestrate the various distributed and complex activities. Although, there is an obvious need for these connectors that are hardware independent. This is especially important in the case of distributed system applications like pervasive computing, natural disaster management systems, email services and so forth. In the last case, every device has its own email client connecting to various email servers using various protocols (SMTP, MIME, POP3, IMAP, etc.). The goal is then to be able to create a software solution which does not involve any extra work from the wireless network operator. In pervasive computing area, it is necessary to adapt the communication of different components to context-aware [6]. The variety of potentially relevant contextual data and the various ways to operationalize pose major problems [7].

This paper proposes a universal connector framework, which is the result of a will to develop our own flavor of what should be a mobile messenger application. The main idea was to build a totally secured, anonymous communication tool that could not be hacked or spied by anyone. Soon, we realized a key technical point was the ability to push messages to a mobile device. After a period of research, it appears that the only available and admitted way to do this was to use a Google or yahoo API, or to rely on a complex third-party library. Both solutions didn't fit our requirements. Indeed, using a big company API implied the loss of control over the data we were sending. On the other hand, all the libraries we found over the Internet (either open-source or commercial ones) seemed to be difficult to use or were presented as a black-box solutions. So, we ended up developing our own push solution.

The framework described in this paper was designed to be as generic as possible, even considering its current use. Indeed, we can understand the current challenges that all software developers and architects are dealing with, especially in today's trend of distributed systems and connected devices. Considering the constantly growing bandwidth available on the internet, and the low cost of cloud computing, we designed our model to be deployed in the cloud.

This paper is organized as follows. Section 2 gives an overview of the background and related works. Section 3 describes the universal connector framework for pervasive computing. An implementation and validation of results are given in section 4. Conclude remarks and future works are described in section 5.

2. Background

To rely on efficient and robust technologies, and to facilitate the maintenance of our system, we decided to design our connector in the most modular way such as each component is independent of each other. In this section we describe briefly health care systems and two important technologies: Aspect-Oriented Programming, and Spring Framework. These technologies can be used to reach our goals.

2.1. Pervasive computing in Health Care Systems

Several research works aimed to use context-aware platform in order to improve the health care systems. Context-aware systems are emerging as an important class of applications. Such systems can respond intelligently to their physical world changes, acquired by using sensors, and according to the information about the computational environment and user's needs [5]. However, many of today's context-aware system architectures tend to treat the need of no-human intervention by a static design of artifacts that implement the variations of needs in the software features during the development cycle.

Contemporary systems must follow a flexible and adaptive architecture to perceive, sense, respond and, in consequence, to suit the particularity of their operation environments. A combination of runtime adaptability and

context features aware modeling offers best approach to dial with the need of reconfiguration during runtime. Therefore, a universal connector between different components is a key concept to make a flexible and dynamic communication between them.

2.2. Aspect-Oriented Programming (AOP)

Aspect-oriented programming is a paradigm that supports two fundamental goals: [2]

- Allow for the separation of concerns as appropriate for a host language.
- Provide a mechanism for the description of concerns that crosscut other components.

AOP does not replace OOP or other object-based methodologies. Instead, it supports the separation of components, typically using classes, and provides a way to separate aspects from the components [2].

We can use AOP to crosscut classes (components) to setup relationships between them without modifying functions in the code of original components. In other words, the code can be kept clean and independent by using Aspect-Oriented Programming.

2.3. Spring Framework

The Spring framework is a wide-ranging framework to develop enterprise Java applications. It provides a lightweight solution and a potential one-stop-shop for building enterprise-ready applications [3]. The Spring Framework is organized as modular. These modules are grouped into 6 types [3]: i) Core Container, ii) Data Access/Integration, iii) Web, iv) AOP (Aspect Oriented Programming) Instrumentation, and v) Test.

Core container is one of most important modules. Especially, in core container, Inversion of Control (IoC, it is also known as dependency injection) is key feature of Spring framework. Dependency Injection is based on Java language constructs, rather than the use of framework-specific interfaces. Instead of application code using framework APIs to resolve dependencies such as configuration parameters and collaborating objects, application classes expose their dependencies through methods or constructors that the framework can call with the appropriate values at runtime, based on configuration [4].

A connector situated in a distributed system must be based on certain transport mechanism (such as TCP/UDP socket, messaging system…) to connect components in network. According to some situation, the distributed connector needs to change the transport mechanism during the runtime. For example, when the quality of network becomes more stable, the system has the intention of dynamically changing the TCP socket to UDP socket. For reuse of design (and code) and flexible loading mode of java class (or bean), we apply IoC of Spring framework to achieving this objective.

3. Universal Connector Framework for Pervasive Computing using Cloud Technologies

This section presents our connector framework, adopting a top-down approach. We start by presenting the overall model, and then will discuss each component independently.

3.1. Overview of our Model

Our model relies on three simple components : a web service engine that handles incoming requests and transfers a message to the right client. It’s the frontend of our system. Then, the PushUnits, that manage connections to the registered clients. Finally, a mapping component that acts as the directory to our system, storing the communication channel associated to a given client.

Figure 1 presents an overview of the general architecture of our system, with the links between the components. As we can see, our system provides one simple entry point to communicate with several clients. Clients are packed in little groups called PushUnits. Our infrastructure can use multiple PushUnits to scale up (and down) to a very large (small) amount of clients. The exact numbers can be customized depending on the specific needs of the implementation context. Each client is registered in the mapping component individually so it can be accessed by the web service engine.

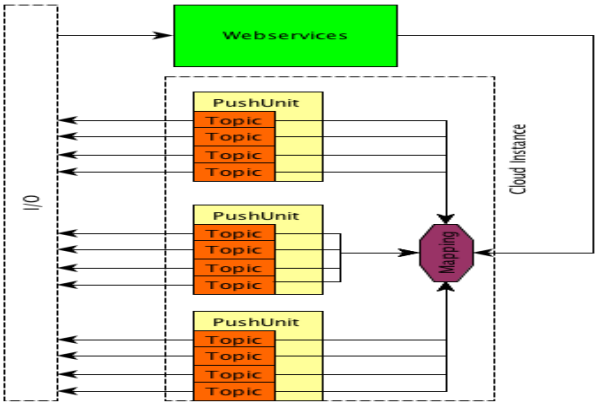

Fig. 1.Overview of the infrastructure of the framework.

In summary, this infrastructure can be seen as a software routing system which could route messages/requests from a component to another, all of these go through the cloud. Note that in our idea, we designed our system for a unicast purpose, but it can be simply converted to broadcast messages by configuring an appropriate Mapping component.

The next sections will develop the details of each components, starting by the pushUnits, then the Web services and finally the Mapping.

3.2. The pushUnit component

A PushUnit is a container that manages several connections to the client components. Basically, it is a TCP/IP socket server that will receive connection requests from clients and will instantiate an encapsulated object for each received connection. Figure 2 is a high level representation of a PushUnit. The red block, i.e. the server socket, listen on a given port for client connection request. When a new connection is received, the PushUnit will instanciate a *Topic* object.

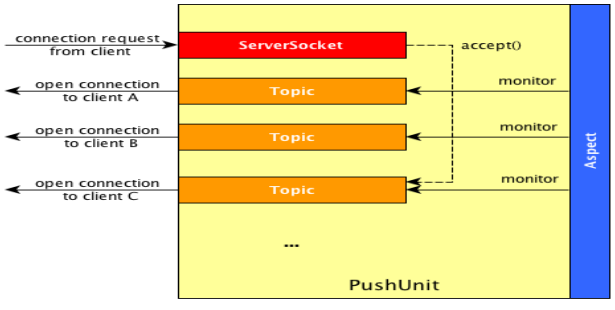

Fig. 2.PushUnit component with aspect unit to monitor all the events.

In order to simplify the mental representation of our system, and to avoid a deep technical description, we introduced the concept of Topic. A Topic is an embedded component that encapsulates both the connection to the user and the first logic layer of a push component. Basically, a Topic instance will be the lowest level of granularity of our system. It represents a link to a client. Figure 3 presents the Topic interface we rely on.

Fig. 3.Topic UML diagram.

It is obvious that, depending on the implementation context, a Topic can be configured with more advanced features, or even use a different connection technology than the one we present here. For instance, you can imagine a UDP based topic if your goal is lightweight fast communication between your server and your client, and if reliability is not your primary concern.

However, as we can see, the specification is quite simple. Considering the fact that a Topic is an encapsulated socket, all you can do with it is to set a message to send to the client and push it. The getter is only here as a convenient way to check the internal status of the component.

So, when a client registers to our platform, it connects to a PushUnit entry point. This PushUnit will create a Topic instance which contains a TCP/IP socket. This socket is maintained active as long as possible so the server can use it to push messages to the client.

We use an aspect-oriented programming to monitor the state of all the instantiated Topics. Indeed, each PushUnit instantiates an aspect-object. When a call to the setMessage method is performed on a Topic, this aspect will trigger the push method automatically on the concerned object. This choice has been made so we only use one thread to monitor a group of Topics, and so, a group of client connections. This idea avoids the handling of one thread per client while maintaining rather good performances. We realize that this AOP idea might create a sort of bottleneck if it is not tweaked properly. One may set the number of topics, and hence clients, a single aspect will monitor depending on its particular performance requirements. If you plan to implement this model on your own, we recommend that you pass the maximum number of topic a pushUnit can handle as an instance parameter. This number must be established empirically as we don't have any specific advice to give on this matter.

Given the above description, there are some issues that are still to address. If your goal is to manage a large amount of clients, it is clear that you may have to use several PushUnit instances. These instances may be created dynamically, which necessitate an hypervisor component. This hypervisor would also act as a load-balancer for the different PushUnits. However, this component will not be discussed in this paper. Another issue is the handling of disconnected clients. Indeed, it is not a good idea to maintain inactive Topic instances (i.e. which broken connections to their client) in your system as we work with a limited amount of allowed connections. The PushUnit must monitor the state of each Topic and be able to detect any lost of connection so it can update its number of remaining connections. This can be achieved by the AOP component.

3.3. Webservices and Mapping Components

Webservices.

The front end of the system is composed of a RESTful webservice. To make them developer-friendly, we adopted the RESTful specification. To explicit how we see this component, we will rely on our messenger app use

case, but keep in mind that this component can be adapted to any application you want. We will discuss other use-cases in the next section.

The first service is the registration service. Indeed, a client application which needs to use our infrastructure must first register itself and so, open a push channel. The process is simple : the client connects to the /register service using a HTTP POST request. The post argument contains the id of the client. The service engine will check in the mapping if this id is valid (which means it is not already in use by the system). Depending on the case in which you want to use the infrastructure, the system can/must check if the request is legit, using a certificate system for example. Finally, the service will return to the client the ip address and port of a PushUnit. The client then has to connect to the given address to open a push channel from the system. And that's it.

Now, you can add whatever service you may need to communicate to a given client. For example, in our messenger context, we added a sendmsg service that receives an encrypted message and the id of the recipient. The webservice will access the mapping component to get a stub to the recipient's topic, and will set the message in it. Once again, you can imagine many more communication possibilities.

The Mapping Component.

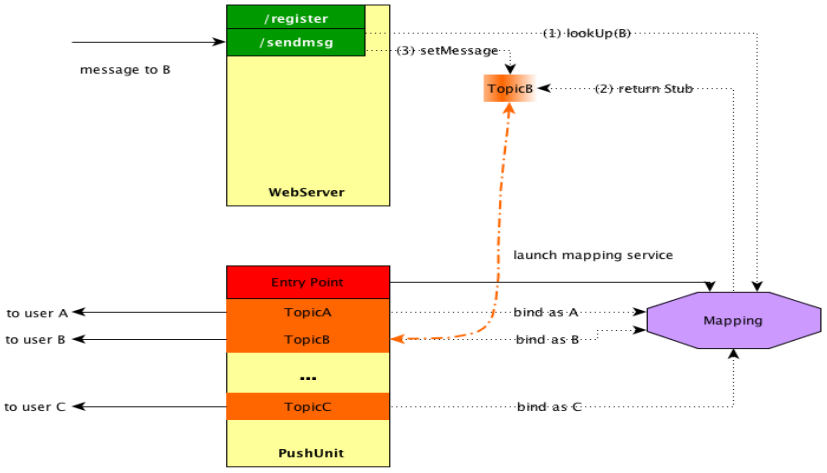

Fig. 4.Mapping System Overview.

The mapping component is a broker system that allows access to remote objects stored in the PushUnits from the WebServices. In our case, the remote objects are Topic instances.

Figure 4 presents how the mapping component is supposed to work. We detail here the process of sending a message to the client named B.

0) Some user send an HTTP Post request to the */sendmsg* service containing the id of the user B, and the content of the message.

1) The */sendmsg* service will perform a lookUp request on the mapping component to obtain a remote interface (a stub) linked to B Topic instance. If B is not mapped in the system, the webservice might send an error answer to the sender.

2) If B is referenced in our mapping, the */sendmsg* service receive a stub linked to B Topic. This stub allows the service to manipulate the Topic instance of B remotely as if it was a local object.

3) The service uses the setMessage method provided by the Topic interface to remotely pass the message content to B Topic.

4) Using the AOP mechanism described in the previous sub-section, the content is automatically pushed to B.

It is interesting to note that a PushUnits component must actually register its Topic instance to the mapping, so they can be accessible by our webservice. The topic registration in the mapping can be performed by an additional

AOP component which role will be to monitor Topic instantiation [11]. This allows a loosely coupled architecture as the logic of each component and the links between them are strictly separated.

3.4. Assembling all the components

Combining the three components described in the previous sections, we obtain a fully functional infrastructure that matches a simple objective : the routing of a message from one component to another through the cloud.

At this time, we use this model for a messenger application. So we spiced up the model with security features including asymmetric encryption. Moreover, to allow total reliability, it would be a great idea to use a database system as a buffer to save messages in case of a unconnected recipient. But this is not the purpose of this paper.

4. Implementation and Results

This model has been implemented and partially deployed at the time this paper is being written. We only use already existing technologies such as Java, RMI and JAX-RS to create our proof of concept. Using an auto-reconnection module, we are able to push messages to a mobile device connected to the internet.

4.1. pushUnit implementation

The PushUnit component has been implemented using Java SE technologies. It uses only Java Socket to assure the connection to a client. For the AOP part, we used AspectJ, which is a Java implementation of AOP. Its role is to trigger the push method when a call to the setMessage() method is performed on a Topic instance. So, this aspect component is really simple, with only one pointcut to monitor. We also used AspectJ for the mapping registration of our Topic objects, and for logging purpose, so we kept our code clean and really task focused. We use the spring framework to allow runtime configuration of the server ports.

4.2. Webservice implementation

The web service is a JAX-RS implementation, relying on the Jersey library. Note that this library is a really simple component. The service is deployed in a Tomcat application server running on its own server.

4.3. Mapping implementation

To implement the mapping component, we decided to use Java RMI as it seemed to match all our requirements. So, all the topic instances are registered in a RMI registry, hosted on a distant server. The web service can perform a lookup on this registry to obtain a topic stub. On the other hand, each topic instance has to be registered in the registry.

4.4. Current results

At the time, we were able to demonstrate the ability of a mobile application to auto connect to our system and to receive push notifications sent by it. We deployed a one PushUnit version of the infrastructure on a dedicated server hosted on the internet. We are still performing our tests and tweaks (e.g. stress-tests, reliability tests, firewall bypassing) and we hope to publish our results in a upcoming paper. But the current system match our first and principle requirement : it delivers messages sent using a web service interface to a mobile device.

5. Beyond the messenger application

This paper uses a messenger context to explain our architecture, but we believe this model can be used in many different domain.

For example, if you consider the domain of health-care, today trend seems the use of small wearable sensors to monitor all kind of vital data. If you consider each sensor as a basic computer with the ability to connect (wirelessly) to a network, using our connector will only require a light embedded http client to send in real-time the collected data to remote computing units. These data could be of any format, or maybe encrypted if necessary, but a connector based on our model would be able to transfer them to a remote analysis unit.

5. Conclusion and Future Works

This paper describes an universal connector model between different devices in term of software and hardware component. This model offers many advantages such as modularity of components, security of message and components, and reliability using a Database system to keep a messages during a period of time. As a proof of concept, we have tested the model using push email system between pushUnit component and smart devices.

Even if a lot of works remains to do to fully demonstrate our model in real-life condition, we are optimistic about the results we may obtain. Considering the current state of development, and the simplicity of the implementation, we hope to have a fully functional system by summer 2014.

But more important, the current implementation seems to reveal the full potential of our model. Indeed, if you imagine a sensor/agent environment, our connector could be the bridge between the hardware, with respecting the modular concept between the components of the systems.

Acknowledgements

This work is supported by the Department of computer science at the University of Quebec at Chicoutimi (QC), Canada.

References

1. R.N. Taylor, N. Medvidovic, and E.M. Dashofy. Software Architecture: Foundations, Theory and Practice. Wiley, 2009.
2. Joseph D. Gradecki and Nicholas Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, Wiley, 2003.
3. Reference Documentation of Spring Framework, Available at http://www.springsource.org/
4. Rod Johnsonet al., Professional Java development with the Spring Framework, John Wiley & Sons, 2005
5. C. Perera, A. Zaslavsky, P. Christen, and D. Geogakopoulos, “Context Aware Computing for The Internet of Things : A Survey,” pp. 1–41, 2013.
6. A., Benamar, N., Belkhatir, & F., Bendimerad. “An Aspect-Oriented Middleware for Adaptation of Pervasive systems,” International Journal of Computer Sciences Issues, Vol. 9, Issue 3, No. 2, May 2012.
7. C., Hoareau and I., Satoh. “Modeling and processing information for context-aware computing : A survey,” New Generation Computing 27 (2009), No. 3, 177–196.
8. Len Bass, Paul Clements, and Rick Kazman (2003). Software Architecture in Practice, Second Edition. Addison-Wesley.
9. Shown and Garlan (1996). Software Architecture: perspectives on an emerging discipline. Prentice Hall.
10. James McGovern, and Scott W. Ambler (2003). A practical guide to enterprise architecture. Prentice Hall PTR.
11. Inderjit Singh Dhanoa et al, Aspect Oriented Distributed System using AspectJ