Design and Evaluation of an MSP-based Body Area Network Management Architecture using JMX

Eduardo Rubio Paniza

Thesis for a Master of Science degree in Telematics
from the University of Twente, Enschede, The Netherlands,
June 2009

Graduation Committee:

Dr.ir. B.J.F. van Beijnum
Dr.ir. I.W. Widya
Prof.dr.ir. H.J. Hermens

Abstract

The increasing availability of mobile devices and the ongoing improvement on their capabilities is leading to further explore their possibilities. As an example of this exploration, the use of mobile devices in Body Area Networks has gained in recent years the attention of the research community. But, although Body Area Networks are lately being subject of extensive research, there is one key aspect that has been marginally addressed: their operational management.

This thesis presents an architecture for the remote management of a Body Area Network that makes use of JMX as management technology and that is fully aligned with the Mobile Service Platform, which is in turn based on the Jini Surrogate Architecture specification. Furthermore a prototype implementation of the proposed architecture is presented and evaluated.

Table of Contents

List of Figures

List of Tables

Preface

This thesis is not just the end of my studies. This thesis marks as well the end of a personal quest. One that started the moment I declined a vacancy to continue my studies in Spain, packed a suitcase and headed to The Netherlands with no more than a bunch of dreams and the intention of being with the woman that is now my wife. This thesis closes a chapter, while the promising first lines of the next one are already being written.

I would like to thank my supervisor, Bert-Jan van Beijnum, not just for his valuable comments and suggestions, but also for his understanding during some difficult periods.

I especially want to express my gratitude for the never-ending and invaluable support from my family and friends, both in Spain and in Holland. I owe them more of who I am today than I will ever be able to repay.

But there is someone that helped me more than anyone making this research possible: my wife. Her love and affection, her support and understanding, her sense of humor and infinite patience kindly guided my steps for the past few months. *Te quiero infinito*.

Eduardo Rubio Paniza

Oldenzaal, 3rd of May 2009

1 Introduction

1.1 Motivation

Over the years, mobile devices have been improving their capabilities keeping up with the fast evolution of technology. Their memory and processing power hasn't ceased to increase, they have incorporated new connectivity options and many other new technologies that have set the potential of ultra portable devices such as mobile phones and Pocket PCs at a level unimaginable a few years ago, in many cases almost redefining their original purpose. Mobile phones are not anymore just a device to make phone calls, but a full featured portable computer with endless applications and possibilities.

As an example, and one that is key to the subject of this research, Pocket PCs have gained an important role in the area of telemedicine, a concept that is rapidly gaining popularity and used to describe the application of traditional medicine where information is exchanged over the telephone or some other communications network in order to deliver clinical care at a distance.

Being a bit more specific, Pocket PCs have become the core component to what is commonly known as Body Area Networks (BAN). A BAN is, simply put, a set of communicating devices that somebody is wearing on his/her body and although its purpose may vary it generally focuses on the collection of vital signs and parameters via a set of sensors to realize some real-time remote patient monitoring and to provide e-Health services. When data has been gathered, it may be transmitted over the internet to a different location, where a medical specialist can remotely analyze and interpret it in real-time and provide proper feedback to the patient.

But, although Body Area Networks are lately being subject of detailed study and extensive research, there is one key aspect that has been marginally addressed: the operational management of a BAN. Considering the key to Body Area Networks is to provide remote medical assistance, what happens if some settings have to be modified? Consider, for instance, that the medical specialist providing remote assistance decides to modify the sampling frequency of the sensors worn by the patient in order to provide a more detailed view of the data collected. If this can't be done remotely the main purpose of Body Area Networks is defeated. But there is more. If necessary, how can the software that controls the patient's BAN or one of its components be stopped, restarted or even updated? How can new components be installed to provide new functionality to the BAN?

It becomes clear that there needs to be a means to manage the BAN resources at a distance, to maximize the intrinsic usefulness of remote medical assistance and monitoring.

In this paper we propose a remote BAN management architecture based on Java and taking advantage of JMX as Java's management solution and MSP as supporting infrastructure for mobile service provisioning.

1.2 Objectives and scope

As we will see in more detail in section 2.1, a BAN is basically a set of resources (be them hardware or software) that are in principle subject to management. So developing a BAN management infrastructure is essential in order to monitor and control the BAN providing fault management support, configuration management, accounting management, performance management and security management [1].

Based on the motivation for this research given on the previous section, the two main objectives of this research are:

1. Design an architecture for the remote management of a BAN, its resources and services using JMX and fully aligned with the Mobile Service Platform (MSP) architecture.
2. Implement and test a prototype of the designed architecture.

It is important to emphasize the remote aspect of our design. Managing BAN resources locally using Java Management Extensions is a subject that has been already addressed in [2]. Our intention with this research is to extend the use of JMX for BAN resource management to do it remotely. This introduces some new challenges that we address in this thesis in order to develop a viable solution.

1.3 Approach

In order to gain sufficient knowledge on the different subjects that this research is based on, studying related literature was the first step. More specifically, literature on Java Management Extensions, the Jini Architecture, the Jini Surrogate Architecture and the Mobile Service Platform has been studied, as well as publications from the Awareness project and from MobiHealth on the related subjects.

Parallel to the study of relevant literature and due to the lack of programming experience, sufficient Java programming expertise was to be gained in order to be able to implement and test the prototype. This proved to be time-consuming but was foreseen while planning the research.

After gaining sufficient knowledge from the literature and gained some Java programming experience, the Mobile Service Platform implementation had to be understood. To accomplish this, the MSP was set up and the code documentation thoroughly studied.

The next step was to design and implement our remote management architecture. This was done by defining different phases:

- Develop a device service to serve as an MBean Server in order to turn our device manageable.
- Develop a first version of the surrogate to be uploaded to the surrogate host. This first version simply functioning as a client for the MBean Server on the device, and establish a connection between both ends using one of the standard connection mechanisms defined by JMX.
- Develop the JMXMSP Connector in order to make use of MSP Interconnect as transport protocol.
- Implement a fully functional MBean Server surrogate

- Implement a simple testing client

Once implemented, the resulting prototype had to be evaluated and carry out some performance tests.

1.4 Thesis structure

This thesis is organized as follows:

Chapter 2 discusses the backgrounds needed to understand the technical environment in which the remote management functionality is to be made possible. To this end, we discuss the architecture of the Body Area Network (BAN), the Jini Architecture, the Jini Surrogate Architecture, the Mobile Service Platform, and the Java Management Extensions (JMX) architecture.

Chapter 3 presents an overview of our architecture for remote BAN management using JMX and MSP.

In Chapter 4 the JMXMSP Connector is thoroughly described, from its design to its development.

Chapter 5 then focuses on the MBean Server Surrogate design and implementation.

In Chapter 6 an evaluation of the solution developed is given, together with a demonstration of the working prototype. Then some benchmarks and performance tests are presented.

Chapter 7 closes this research by presenting the conclusions.

2 Background information

This chapter gives the background information needed to understand the design issues and problems to be addressed and solved within the scope of this research. First we discuss the Body Area Network (BAN), the role it plays in e.g. the monitoring and treatment of patients, and the BAN architecture and technologies used. It also explains the Jini technology: first the Jini Architecture and then the Jini Surrogate Architecture. Then the Mobile Service Platform (MSP) is discussed followed by an overview of the Java Management Extensions (JMX).

2.1 Body Area Network (BAN)

A Body Area Network (BAN) is a network of communicating devices worn on the body that perform specific functions to provide mobile services to the user [3]. Those devices can range from small computing devices (such as PDA's, mobile phones or a GPS receiver) and other gadgets (such as cameras, audio players or other multimedia devices) to (smart) medical sensors and actuators used to monitor and control health related conditions of a person. Besides those devices it is possible to have a central device coordinating the BAN communications and performing diverse computational tasks.

The communication between the different devices within the BAN is called intra-BAN communication, whereas communications with external entities (such as other BAN's or networks) are referred as extra-BAN communications [3]. Typical intra-BAN wireless communication protocols that are used are Bluetooth and ZigBee. Wireless extra-BAN communication typically uses WLAN, GPRS, UMTS or HSDPA.

For the purpose of this research we use a simplified BAN model based on that described in [4], in order to be able to focus our attention on the management possibilities of a BAN. Figure 1 shows a UML diagram presenting this simplified BAN model. The different components of this model will be discussed in the following.

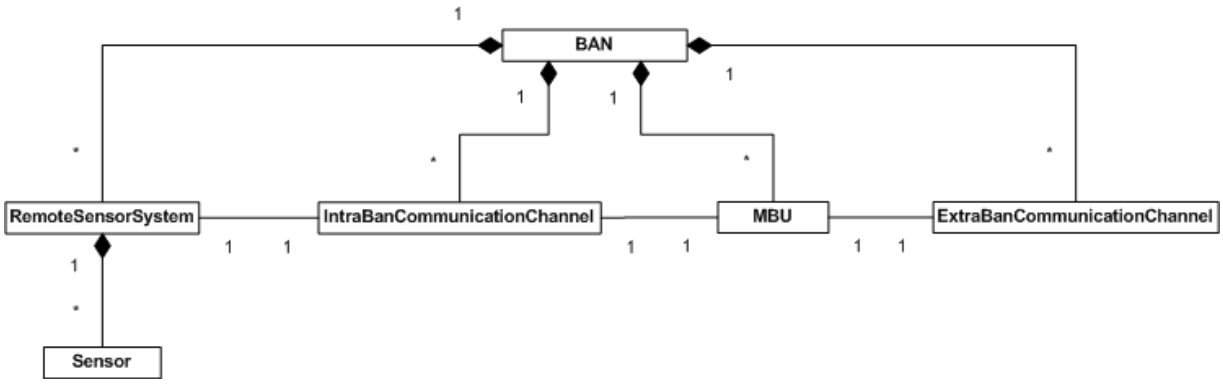

Figure 1. Conceptual model of a BAN [1]

The BAN uses a Mobile Base Unit (MBU) as a central coordinating device capable of performing computation, coordination and communication functions [4,1], as well as signal processing tasks on the

data obtained from the sensor systems. The MBU is typically implemented on a PDA, but could be any other device capable of performing those functions.

The MBU serves also as a gateway between the BAN and the Back End System, performing extra-BAN communications, and possibly hosting other applications to control and manage the BAN.

Intra-BAN communications support the information exchange between on-body sensor systems and the MBU. Each connection link between a single sensor system and the MBU is an Intra-BAN communication channel (see Figure 1).

Intra-BAN communications may be wired using, for instance, copper wire or optical fiber as medium, or wireless. As for wireless technologies, the two especially suitable options are Bluetooth, which offers typical coverage of 10 meters and speeds of up to 1 Mbps on its 1.2 version (and up to 3Mbps with later versions)[5], and ZigBee offering up to 250 Kbps and 100 meters [6]. Although the Bluetooth standard (IEEE 802.15) is a very popular short range communications technology for creating Personal Area Networks, its relatively high power consumption [3] position the later as a very promising technology regarding Intra-BAN communications.

Extra-BAN communications take care of the exchange of information between the BAN, and more specifically the MBU, and the Back End System (see section 3.2). For convenience, in order to support the mobility of the person wearing the BAN, this communication uses wireless technologies. There are plenty of possibilities regarding extra-BAN communication technologies, and there is not a single answer as for which one is the most suitable option. The answer depends on the specific requirements, and can be pinpointed considering certain criteria such as the communication range to be supported or the speed of the data exchanged required.

A technology like WLAN (Wireless Local Area Network) is especially suitable for short to medium range communications, offering varying speeds depending on the specific version of the standard but typically of 11 Mbps (and up to 54 Mbps), and a typical transmission range of about 100 meters.

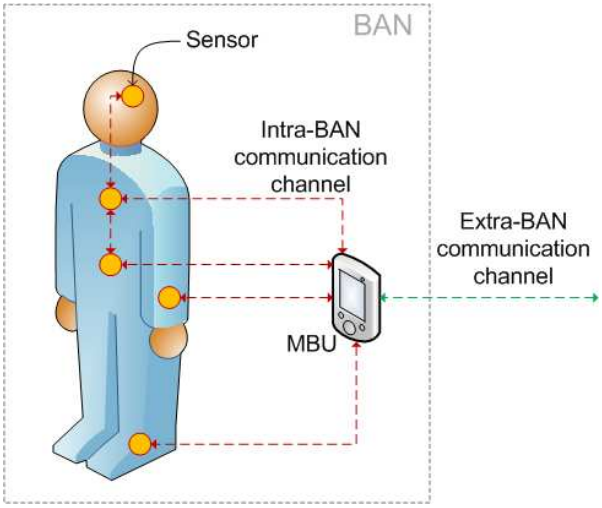

Figure 2. Generic architecture of a BAN

Wide area communications on the other hand are typically realized using GPRS, UMTS or HSDPA technology [3,1], providing the BAN with a mobility only limited by the coverage of these networks.

For the purpose of this research and with the simplified BAN model presented in Figure 1 in mind, we can consider the set of inter-communicating devices that compose it to be typically a set sensors gathering physiological data (such as blood pressure or heart beat rate) from the body of the person

wearing them. It is worth noting, though, that these devices are not limited to sensors, but could be any type of device. The data gathered by these sensors is transmitted to the MBU for eventual further processing.

As a remark, when the main purpose of a BAN is to use the devices that compose it to gather, process and transmit biosignals, or to offer other health-related services, the term BAN is generally understood as a healthcare BAN or health BAN [1]. In the sequel, when the term *BAN* is used, it should be understood as a health BAN.

When considering the applications and uses of a health BAN within the scope of telemedicine a clear application opens up for exploration: remote patient monitoring and teletreatment. This application is explored in [7]. In [8], homecare coordinated by home nursing services and supported by the patient's BAN in combination with an ambient intelligent home environment is explored. An approach for using context-aware BAN's for Telemedicine is presented in [9], aimed at applications in neurology.

2.2 Jini Service Architecture

The Jini technology is a service oriented architecture developed by Sun[1] and designed for the construction and deployment of flexible distributed systems in order to bring together users and the resources they might require. Any resource providing some service can announce its presence on the network through the Jini architecture so users can locate the resource and make use of it. And by resource here we understand anything from hardware devices (i.e. a printer or network hard disk) to software programs or even other users providing a service.

Simply put, the purpose of Jini is to provide a sort of plug and play network that allows users and services to participate on the network in a flexible and easy way, without the need of a central controlling authority. This makes Jini particularly suitable for dynamic environments where users and resources continuously change their network locations.

Considering a typical Jini scenario would basically involve clients, services and a Lookup Service to be able to locate each other. And supporting the communications between these three parties we have a network.

As an illustrative example, we can think for example of a user with a digital camera that wants to share some pictures. The user might want to upload the pictures to an online storage device for backup, use a printer to have a hard copy of them or share them online through some photo gallery service. In such a situation, Jini would provide the infrastructure for the user to easily locate and make use of those services.

---

[1] Originally developed by Sun, Jini was contributed to the Apache Software Foundation for its maintenance and further development under the project name Apache River [22,23].

Note that although Jini is written in Java and much of its simplicity derives from using Java to develop the different components involved, it is not required for neither clients nor services to be developed in Java. A service that wants to be able to participate on a Jini network would be able to do so by providing a proper Java wrapper.

2.2.1 Clients

A client is basically a user, a hardware device or even a software program, that wishes to make use of some resource, be it a printing service, an online calendar application or any other service. While talking about Jini, it is important to emphasize that a client is *anything* that wants to use a service.

2.2.2 Lookup Service

For a client to be able to use a service it first needs to be able to locate it on the network. For this Jini provides a Lookup Service (LUS), the role of which is to serve as a central service repository. When a new service becomes available and wishes to offer its services, it first needs to register itself with the LUS so clients are able to locate it. When a client needs to make use of some service it first queries the LUS asking for it by providing its description.

2.2.3 Services (service provider)

Services are the cornerstone of the Jini architecture. According to the Jini Architecture Specification provided by Sun [10] "a *service is an entity that can be used by a person, a program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user.*"

A Jini system consists of various services that can be running anywhere: on servers, desktops, laptops, mobile phones, etc. As we will see this flexibility is key for the foundations of this research.

Technically, services are objects written in the Java programming language (or at least providing suitable Java wrappers), with an interface defining the operations that can be performed by that object. Services can advertise their presence on the network to users and other services and make use of other services as well and they are free to define and make use of any appropriate communication protocol depending on their requirements.

Another concept worth mentioning is that of a Service Provider. A Service Provider creates the object for the service it wishes to make available to clients and registers it with a LUS. That registered object is called the Service Object and is the object that clients will have to obtain in order to interact with the service. The Service Object bundles the Java interface for the service and possibly a set of attributes [10] to further describe the service and make it easier for the service to be found. This movement of code ensures that the implementation the client will run is an up-to-date version of the Service Object, since

it is supplied by the Service Provider, and can be updated by registering a newer version with the Lookup Service [10].

Service Objects generally implement one or more well known interfaces that the clients will use in order to interact with the service located on the Service Provider. These Service Objects are basically *proxies* and they can be implemented following different approaches, depending on how much logic of the service is going to run on the client and how much on the so called service back-end [11].

So the Service Provider may, for instance, create a Service Object with a complete implementation of the service. This is commonly known as a 'fat' proxy, since the whole implementation of the service is registered as a Service Object with the Lookup Service and later downloaded and used by the client. This approach may be suitable in situations in which the service implementation does not require in any way to communicate back with the Service Provider, being able to run independently from it.

A common alternative approach is to use a 'thin' proxy. A Service Provider may create a Service Object that simply forwards method calls from the client back to the service back-end, possibly using Remote Method Invocation (RMI).

Any intermediate combinations of the previously mentioned approaches, called *smart proxies*, are possible, where part of the implementation is handed in to the client and the rest is run by the service back-end [10]. However, having a back-end service allows the service to use resources local to the Service Provider host.

In any case the client will receive a Service Object in order to make use of the service. If the service has been designed so that the Service Object has to communicate back with the service back-end, both client and server sides have to use a common communication protocol. But, since the proxy used by the client has been provided by the Service Provider itself the client is not required to know anything about the communication protocol, since it is something kept between the Service Object and the Service Provider [10]. This provides convenient transparency for the client regarding how the communications are handled and allows updating the communications protocol whenever necessary without affecting the way a client is implemented.

2.2.4 Discovery, Join and Lookup protocols

The Discovery, Join and Lookup protocols provide the means for a service to announce its presence on the network and for users to be able to find it and make use of it. These protocols are the core functionality of Jini.

A Service Provider will make use of the Discovery protocol in order to locate a Lookup Service with which to register the service it wishes to offer so it can be advertised on the network (Stage 1 in Figure 3). This can be done in two ways. If the location of the Lookup Service is known to the Service Provider, the Discovery process consists of a unicast request directed to a well-known port on the Lookup Service host. Otherwise, if the Service Provider is not aware of the exact location of any Lookup Service the request will be broadcasted on the network directed to the same well-known port.

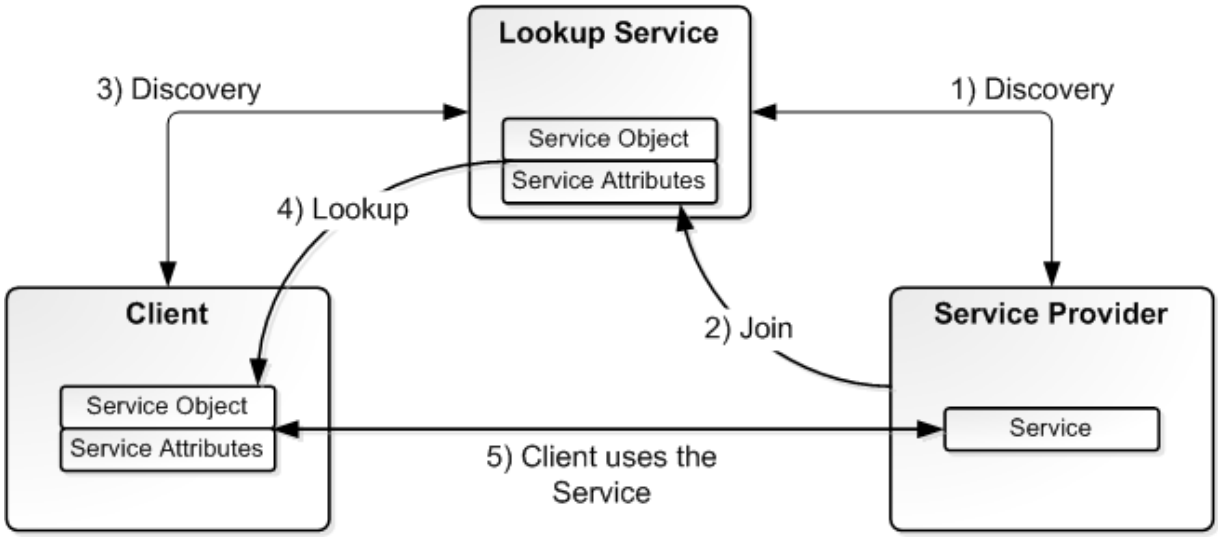

Figure 3. Jini Discovery, Join and Lookup protocols.

Any Lookup Services listening for and receiving one of such requests will return a *service registrar* object. The registrar object is a proxy that allows the interaction with the Lookup Service. At this point, the Service Provider is ready to perform the Join using the *service registrar* object to register the service with the Lookup Service by providing a suitable Service Object (Stage 2 in Figure 3).

When a client wishes to make use of some service, it first has to use the Discovery protocol to find a Lookup Service and obtain a *service registrar* object in the exact same way the Service Provider does (Stage 3 in Figure 3). The client, however, will use the *service registrar* object received in a different way, in order to query the Lookup Service for a specific service through the process called Lookup.

During the Lookup process, the client will create a *service template* object specifying the service it is looking for, which will serve as the search criteria for the query sent to the Lookup Service. Since the client only knows the service it is looking for through some sort of specification (typically a Java interface), in general this object will contain one or more class objects for service interfaces and possibly other attributes [10].

When the Lookup Service receives the query from the client it will use the *service template* object to see if any of the Service Objects registered matches the criteria. If there is a match, a Service Object (an instance of a class for the desired service) will be returned to the client (Stage 4 in Figure 3). If the client has the class definitions for the object obtained, it can be used straightaway. Otherwise the client has to somehow obtain the proper class definitions for the Service Object [11], which typically can be downloaded from an HTTP server (see 2.2.7).

Now the client is ready to use the Service Object to communicate directly with the Service (Stage 5 in Figure 3). The client invokes the methods defined by the service interface directly on the Service Object. These method calls will then be forwarded (possibly using RMI) to the Service, which in turn will, if necessary, perform some operations and send a reply back to the client.

2.2.5 Events

Like many other Java technologies, Jini provides a mechanism to support distributed events. With it, objects can register interest for being notified of events triggered by other objects. When one of such events is triggered, the object originator of the event will check if there are any event listeners in order to notify them. To further understand the Distributed Events model used by Jini we refer the reader to chapter 16 in [11].

2.2.6 Leasing

Leasing is the mechanism by which Jini avoids the presence of outdated and unnecessary resources on the system. It is safe to assume that failures can occur on the network or network components. In such situations a service could terminate unexpectedly with the Lookup Service being unaware of this. The Service Object registered with the Lookup Service would be still offered to interested clients that would unsuccessfully try to make use of the service.

Jini deals with this type of situations providing a leasing solution [10]. For instance, when a service registers itself with a Lookup Service it requests a lease for a certain period of time. If the lease is granted, the service is responsible for periodically requesting its renewal. If at some point the service fails to renew its lease, whatever the reason, the Lookup Service will remove its corresponding Service Object from its resources. At that point, the service would have to register again with the Lookup Service and ask for a new lease.

2.2.7 Jini and RMI

A basic concept behind Jini is that of moving code around between clients, services and LUS's. This involves things such as serializing the objects to be moved from one Java Virtual Machine (JVM) to a different one, and remotely invoking methods on them. Jini builds on top of Java Remote Method Invocation (RMI) to accomplish this.

With RMI a server application (within Jini, some Service Provider) creates a remote object (an object extending the Remote interface) providing some service. Then the server creates a special type of proxy object to the service and makes it available for clients to find it. This can be done by exporting the proxy object to an RMI registry, a Jini Lookup Service or some other type of network service directory.

A client obtaining a reference to a remote object can invoke methods on it, and the details of the invocation will be handled by RMI, making the method calls completely transparent for the client. Since the communication between the client and the server may involve passing objects back and forth it is possible that the class definitions for such objects are not available within the client's JVM in which case RMI handles the download of the necessary class definitions previously made available by the server by placing them (typically) on an HTTP server. The location of these class definitions will be specified by providing a codebase property when running the server.

2.3 Jini Surrogate Architecture

For a device to be able to participate on a Jini network and offer any type of service it has to fulfill certain requirements: it has to be able to execute Java code and be able to perform the Discovery, Join and Lookup protocols described on section 2.2.4. As seen, these protocols involve, among other things, uploading and downloading Java objects or performing common RMI procedures (like downloading class definitions or creating and registering proxy objects).

But, some limited devices may not be able to meet these requirements (due, for instance, to the lack of resources necessary to run a JVM) and thus are unable to participate directly on a Jini network. In order to overcome this, the Jini Surrogate Architecture extends Jini by providing the means to allow devices unable to participate directly on a Jini network to do so aided by a third party.

Using the Jini Surrogate Architecture a limited device can provide a surrogate, a Java object that represents it and acts on its behalf. This surrogate will be uploaded (directly or indirectly) into a host-capable machine that is free of the device limitations, and running the Surrogate Host.

The Surrogate Host provides an application environment, consisting of one or more JVM's, specially designed to host surrogates. It takes care of managing the resources used by the surrogates and controls when they are loaded, started, stopped or removed (see 2.3.1 for more details).

2.3.1 Interconnect Protocol

In order to communicate, a limited device and a Surrogate Host use the *interconnect*, which determines the logical and physical connection between them. Note that more than one interconnect can be defined for a single physical connection [12]. The mechanism that describes how the device and the Surrogate Host use a specific interconnect is the Interconnect protocol. Although [12] does not provide a specific implementation, it does state the minimum requirements that must be met by any interconnect protocol in order to comply with [12]. These minimum requirements are described here.

In order to establish a connection, a device and a Surrogate Host must first find each other. This is accomplished through the discovery[2] phase. This mechanism is interconnect specific and different implementations are possible depending on the capabilities and requirements of the device and the interconnect itself [12]. As an example, in a possible implementation a device could broadcast its presence on the network by sending discovery packets until a receiving Surrogate Host would respond announcing its availability.

Once they've found one another, the surrogate retrieval phase begins, in order to load the compiled class files of the surrogate representing the device into the Surrogate Host. This can be done in different ways. The device can push the surrogate to the Surrogate Host, the Surrogate host can download it from the device or it can even be retrieved by the Surrogate Host from a different location (i.e. a third party

[2] Not to be confused with the Jini Discovery protocol described in section 2.2.4

FTP or HTTP server on the network). In any case, at the end of the surrogate retrieval phase the Surrogate Host is in possession of a surrogate object, which must then be activated by instantiating it and then calling the method to begin its execution. These procedures, as well as assigning the necessary resources for the surrogate to run, are handled by the Surrogate Host and the details depend on the Surrogate Host implementation [13].

At this point the surrogate is running on the Surrogate Host, free of the limitations of the device they represent. The surrogate instance is now capable of joining the Jini network, advertising its presence on Lookup Services to offer the services provided by the device by uploading a Service Object (as described in section 2.2.4), or even directly make use of other Jini services. And whenever necessary, the surrogate may communicate with the device it represents.

Once the surrogate has been activated, either the surrogate or the Surrogate Host itself must monitor the device for *liveness*, making sure that the communication between the device and its surrogate it's possible and that both are active. If at any point the communication link is broken for whatever circumstances the surrogate will be deactivated and the resources reallocated by the Surrogate Host [12]. The device must somehow be able to determine that the connection with its surrogate has been broken and should resume discovery or be ready to be re-discovered by the Surrogate Host. Note that it is the surrogate's task to take care of liberating possible Lookup Service leases or other resources [12].

For further details on a specific interconnect we refer the reader to the IP Interconnect specification provided by Sun [14]. And for a specific implementation of the Surrogate Host a good place to start is *Madison*, the Surrogate Host implementation made available by Sun, that conforms both to the Jini Surrogate Architecture [12], and the Jini IP Interconnect specifications [14], and which can be found at [15].

2.4 The Mobile Service Platform

MSP is a software infrastructure originally developed in Java by the University of Twente and based on the Jini Surrogate Architecture that aims to extend the Service Oriented Architecture to the mobile device and provide a supporting infrastructure for the development of Nomadic Mobile Services (NMS). A Nomadic Mobile Service is a type of service offered by a mobile device (like a mobile phone or a PDA), capable of participating in a service discovery network and of roaming from one wireless network to another (i.e. switching from a Wi-Fi network to UMTS, GSM, etc.).

Due to their characteristics, to provide Nomadic Mobile Services certain requirements must be considered [16], such as:

- The communication between the service and the client making use of it must be seamless even while roaming from network to network. This involves dealing with things like IP address changes and Network Address Translation (NAT) problems.
- The limitations of the execution environment must be taken into account. Providing a service on a mobile device changes its role from client to server. In general (as it is the case in 2.5G and 3G

networks) the upload link is lower than the download link, so data sent should be optimized to take this into account. Other considerations involve intermittent network connectivity, limited battery life, etc. For further details we refer the reader to [16].

- Scalability. Scalability is not much of an issue on the fixed network since additional resources can be easily added. For Nomadic Mobile Services this is not the case, limiting the number of potential clients for the service to the mobile device power and its network capabilities.

MSP provides a middleware platform to address these requirements and ease the development of NMS. There are two versions of the platform: MSP and Context Aware MSP.

2.4.1 MSP

With MSP a Nomadic Mobile Service consists of two components: a *device service* (a service hosted by a mobile device) and a *surrogate* that represents it running on a Surrogate Host in the fixed network, with all the advantages that this represents (see 2.3). This idea is based on the Jini Surrogate Architecture, and allows clients to communicate with the service using the surrogate as a proxy to address the second and third requirements mentioned above.

The MSP itself can be divided in three main parts: 1) the MSP-IO, which is the part of the interconnect implementation (see 2.3.1) that resides on the mobile device and interacts with the device service in order to handle all the messages sent to and from it. 2) The MSP-Interconnect, which is the part of the interconnect implementation that resides on the Surrogate Host and handles the activation and deactivation of the device surrogate as well as the messages sent to and from the surrogate. 3) The MSP-Messages package, which defines the structure of the messages exchanged between the device service and its surrogate on the Surrogate Host.

The MSP interconnect implementation conforms to the requirements specified by [12] and extends the means by which device service and surrogate communicate by providing three types of interactions:

- *One-Way*: this type of interaction allows for unacknowledged message exchange between the device service and the surrogate without a reply being sent by the receiving side.
- *Request-Response*: the type of interaction that allows for a request message to be sent that must have a corresponding reply message, providing reliable message exchange.
- *Streaming*: supporting continuous exchange of data streams.

2.4.2 Context Aware MSP

The goal of the context aware MSP is to exploit the potential advantages of multi-homing for Nomadic Mobiles Service provisioning by taking a context-aware computing approach. Multi-homing refers to the capabilities of mobile devices to connect to the Internet using multiple network interfaces (i.e. Wi-Fi, GSM, UMTS…).

The fact that Nomadic Mobile Services invert the role of the hosting device from being a client to that of being service provider (with the execution environment and upload link limitations that this typically brings) and the fact that the service roams along with the device it is hosted on makes multi-homing support a desirable feature for NMS's.

The context-aware MSP states in [17] some multi-homing requirements for NMS's. These requirements are here briefly discussed:

- *Optimal Network Selection*: Considering the multi-homing capabilities of mobile devices and the fact that the reduced uplink bandwidth that typically characterize existing cellular networks compared to the download link (e.g. the GPRS uplink theoretically offers 20 kbps while the download link theoretically offers 40 kbps), it is important that MSP selects the best available network based on the uplink bandwidth and the service bandwidth requirements.
- *Low Latency*: In some critical environments (e.g. within telemedicine applications) a network with low latency is desired.
- *Session Handling*: MSP should be able handle ongoing data transfers during the network handover (i.e. when switching from one network to another) buffering data in case the new network provides less bandwidth than the previous one.
- *Economic Internet Availability*: Additionally, selecting the network providing the lower connectivity cost can be of interest in non-critical environments.
- *Reasoning Support*: MSP should be aware of the capabilities of the mobile device (e.g. the network interfaces available), their properties in order to determine the most suitable network depending on a number of factors, such as QoS of the available network interfaces, their actual throughput, cost, energy requirements or the criticality of the NMS.
- *Context Sensing and Processing*: Context sources should provide real-time context information (e.g. new network availability) in order for the reasoning support to be able to determine the suitability of a network at any moment.

2.5 Java Management Extensions

The Java Management Extensions (JMX) is a Java technology that provides a specification as well as an implementation to manage and monitor Java applications, devices (e.g. a printer), service implementations and so forth. Such resources are exposed for management as objects called MBeans. In this section an overview of the JMX architecture is given. For further details we refer the reader to [18].

The JMX architecture consists of three different levels: the Agent level, the Distributed Services level and the Instrumentation level. Each of these levels will be briefly discussed in the following in order to provide proper background knowledge for the reader to understand the basics of JMX and how it is used in the subject of this research.

A picture showing an overview of this architecture is shown in Figure 4.

2.5.1 Agent level

The JMX agent consists of an MBean server and a set of agent services useful to handle MBeans. The task of the agent is to control the resources and make them available for management. This is done by means of the MBean server, which serves as a registry for MBeans. Through the MBean server, a management application or any other object, can access the attributes exposed for management by a registered MBean, invoke operations on it, receive any notifications it might emit and even create new instances.

Although additional agent services can be developed, the JMX specification provides a minimum set of services like a *monitoring service* which enables a listener to be notified under certain circumstances, a *timer service* that allows scheduling for sending notifications, a *relation service* that defines and maintains the associations between MBeans and a *dynamic loading* that allows instantiating MBeans using Java classes downloaded from the network.

2.5.2 Distributed Services Level

This level provides the means by which a management application can have access to the JMX agent in order to manage an MBean. This communication is realized through connectors and protocol adaptors. Connectors are the components used when the remote client is a JMX-capable application (i.e. sees the JMX API in the same way a local client would). Adaptors are components designed to provide JMX capabilities to clients using other management technologies such as SNMP or even a web browser.

This research focuses on the connectors defined by the JMX Remote API specification, and so we provide some more information about them. For more information on protocol adaptors we refer the reader to [18].

A connector consists of a connector client and a connector server. The connector server is attached to an MBean server and waits for incoming connection requests from clients. The connector client is at the other end, attached to a (possibly remote) client and is responsible for finding a suitable connection server and trying to establish a connection.

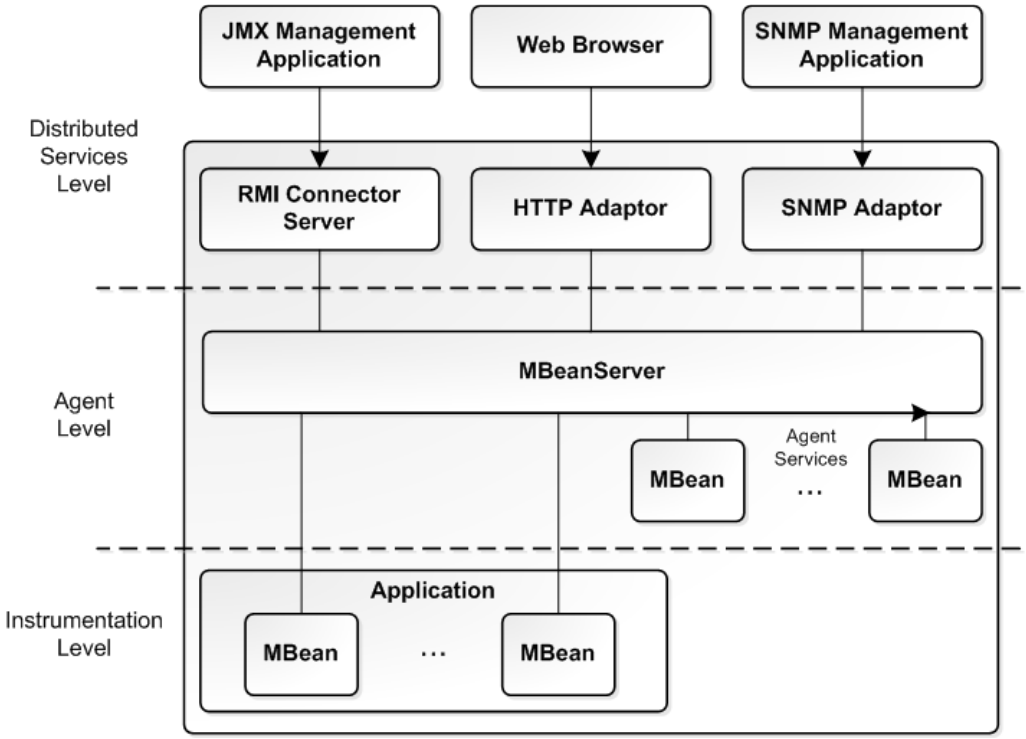

Figure 4. Overview of the JMX Architecture.

The JMX Remote API specification describes three different connectors:

- The RMI connector, which uses Remote Method Invocation (RMI) for the communication between the client and the server, and which must be supported by any conformant implementation of the JMX specification.
- The Generic Connector, which is designed to allow the creation of new connectors by simply plugging in a *transport protocol* and an *object wrapping* modules.
- The JMXMP connector, a particular implementation of the Generic Connector that uses JMX Messaging Protocol which is based on TCP as client-server communication protocol and Java serialization as *object wrapping*. An implementation of the JMXMP Connector can be found in[19].

2.5.3 Instrumentation level

The instrumentation level contains the resources to be managed and specifies the way they can be instrumented for management using JMX technology. Such resources must be implemented in Java (or provide an appropriate Java wrapper) and instrumented by means of one or more MBeans [jmx spec 1.4]. This level is composed of three main components: the MBeans, the notification model and the MBean metadata classes.

An MBean is a Java concrete class that conforms to a specific management interface that determines the way the resource can be managed. There are four different types of MBeans to cover different

instrumentation requirements: Standard, Dynamic, Open and Model MBeans. In the scope of this research we will limit ourselves to Standard MBeans. For more information on the different types of MBeans see [18].

Standard MBeans are the easiest and most straightforward way to instrument a resource. A Standard MBean is instrumented by implementing its own MBean interface and optionally the NotificationBroadcaster interface. Their management interface is composed of:

- The attributes that can be accessed (exposed through getter and setter methods). Defined statically in the MBean interface.
- The operations that can be invoked on the resource. Defined statically in the MBean interface.
- The notifications that it can emit (see Notification model).
- The constructors of the MBean Java class. Only public constructors are exposed.

The instrumentation level defines as well a notification model. This model specifies the way JMX Agents and MBeans can broadcast notifications with important information to other interested parties such as management applications and other MBeans.

Whenever a management application (or any other object, for that matter) is interested in receiving a specific type of *notification* it has to register as a *listener* with the MBean broadcasting them.

The last main component defined by the instrumentation level is the MBean metadata classes. These classes contain a description of all the components of an MBean's management interface including its attributes, operations, notifications and constructors. For each of these components there is metadata with the name, the description and other information such as parameter signatures for operations and access permissions for attributes.

These classes make possible to provide a representation for an MBean that can be presented in different ways to a management application, such as for instance through a graphical user interface like the Java Monitoring and Management Console (JConsole).

3 Remote BAN Management with JMX and MSP

This chapter provides a high level overview of a remote BAN management architecture that both builds on and extends JMX and MSP. First the BAN management requirements that this architecture must satisfy are formulated. Then, based on those requirements, the different parts of the proposed architecture are discussed.

3.1 BAN Management Requirements

From a technical point of view we identify three main categories for the requirements that must be satisfied by the remote BAN management architecture we propose. These categories are *BAN management capabilities* (i.e. what should be manageable), *integration requirements* (i.e. requirements addressing the use of JMX together with Jini and the Jini Surrogate Architecture to provide remote BAN management) and *instrumentation requirements* [1].

BAN management capabilities: Here we consider the usual capabilities commonly found in management technologies, namely:

- *Management attributes*: read and write management attribute values of a resource.
- *Management operations*: invoke methods of a resource which are exposed for management.
- *Management notifications*: notifications informing of events occurring to the manageable resource (e.g. an attribute value changed).
- *Management meta-data*: information describing the management interface of a resource (i.e. management attributes, operations, notifications, etc.)

Besides, the BAN architecture and its different components (as discussed in section 2.1) have to be considered in order to identify what should be exposed for management. Examples of interesting components to be provided with management instrumentation are the MSP-IO component, device services, BAN sensors or the JMXMSPConnectorServer described in the following section.

Integration requirements: These requirements take into account different aspects of bringing together technologies such as MSP (and thus Jini and the Jini Surrogate Architecture) and JMX to provide remote BAN management capabilities.

- Management applications (e.g. JConsole) must be able to remotely access the BAN for management.
- The BAN management architecture proposed must extend the Jini and Jini surrogate architectures, and make use of the Mobile Service Platform in order to handle the communications between a device service (an application running on a mobile device) and its surrogate (running on a Surrogate Host), taking advantage of the MSP interconnect protocol (see sections 2.4.1 and 2.4.2).

- Remote management applications must communicate with the surrogate object representing the BAN management service, instead of communicating with the BAN management service directly.
- A BAN management service must be able to participate in a Jini network (i.e. a BAN management service must be able to announce itself by registering with a Lookup Service).
- The reuse of existing management transport protocols (preferably standardized) must be considered to minimize design and implementation effort.

Instrumentation requirements: These mainly derive from the underlying technologies used by MSP.

- Device services are developed in Java, and run in a JVM on a Mobile Device, thus using Java as management instrumentation technology is the natural choice. This allows for minimal changes on the current service design and development process and encourages the design of management functionality as an integral part of such process.
- The BAN management instrumentation must be efficient in terms of memory and CPU usage.

This last requirement is beyond the scope of this research, although in an effort to determine the impact of the overhead added to the BAN by this management architecture, some simple benchmarks have been carried out and are discussed in chapter 6.

3.2 Remote BAN Management Architecture Overview

Based on the requirements described on the previous section a remote BAN management architecture is proposed, that both is based on and extends JMX and MSP. Figure 5 shows the different components of this architecture. This section gives an overview of these components, which are discussed in detail in subsequent chapters of this research.

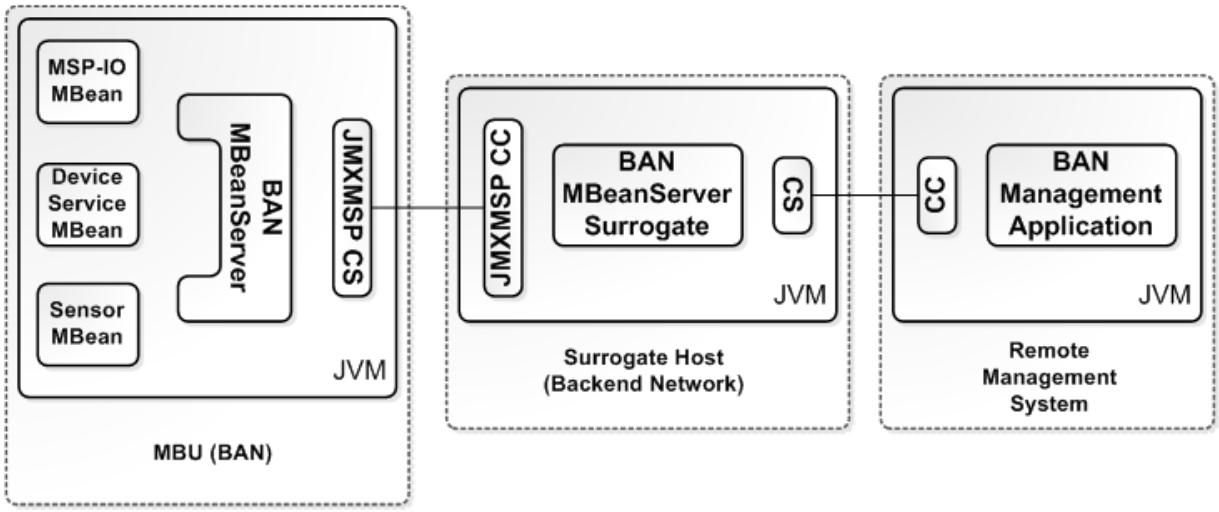

Figure 5. Proposed BAN Management Architecture.

- *BANManagementService*: This is the device service running on a JVM in the MBU of a Body Area Network and providing a BAN management service. The BANManagementService is responsible for creating the BAN MBeanServer that will be used to manage the BAN resources.
- *BAN MBeanServer*: This is a regular JMX MBean server running on a JVM in the MBU of a Body Area Network and created by the BANManagementService. Its main purpose is to serve as a local repository for the BAN resources that are exposed for management. These are the resources that we want to be able to manage remotely. Both locally and remotely, accessing resources for management is done via this MBean server.

  The MSP-IO component, device services residing on the MBU or BAN sensors collecting biosignal data are all examples of resources that could be exposed for management by registering an MBean with the BAN MBeanServer.

- BANManagementSurrogate: This component is the surrogate object running on the Surrogate Host that represents and acts on behalf of the BANManagementService. The BANManagementSurrogate is responsible for creating the MBeanServerSurrogate.
- *MBeanServerSurrogate*: This object acts as a proxy to the MBeanServer on the MBU, so any client (e.g. a management application) that wants to remotely access the BAN manageable resources must communicate with this object instead.

  This provides great benefits over directly accessing the MBU resources for management. For instance, the MBeanServerSurrogate object is in this way freed of the resource limitations (i.e. memory and processing power) of the MBU by running on a Surrogate Host. Furthermore, being a proxy between clients and the MBU manageable resources, the MBeanServerSurrogate can be implemented to provide extra functionality. Examples of such extra functionality are an authentication mechanism to allow or deny access to the MBU resources using some predefined policy or a caching mechanism to minimize the management overhead of the BAN and the communications between the MBeanServer and its surrogate.

  This last possibility is explored in this research by implementing a simple cache as prove of concept to demonstrate the possibilities that this architecture can offer. Details are given in chapter 5 and some benchmarks exploring the use of the caching mechanism are shown in section 6.2.3.

  However having a surrogate also has some disadvantages. It introduces a state synchronization problem. The surrogate must be aware of the state changes on the device service [16]. This issue is further accentuated if some caching mechanism is in place, and must be taken into account when implementing the MBeanServerSurrogate.

  The MBeanServerSurrogate is a new component (i.e. not part of the JMX specification) and is further described in chapter 5.

- *BAN MBeanServer – MBeanServerSurrogate communication*: Communication between the MBeanServer and the MBeanServerSurrogate is handled by the JMXMSP Connector. As it can be

seen in Figure 5 this connector consists of two distributed components: the JMXMSPConnectorServer residing on the MBU and the JMXMSPConnector residing on the Surrogate Host with the MBeanServerSurrogate.

The JMXMSP Connector is a new connector (i.e. not part of the JMX specification) built on top of the JMX Generic Connector, in order to be able to use the MSP Interconnect as transport protocol. This new connector is discussed in detail in chapter 4.

- *MBeanServerSurrogate – Management Application communication*: Communication between the MBeanServerSurrogate and a client (i.e. typically a remote management application, such as JConsole) is handled using standard connectors as specified in [18], such as the JMX RMI connector or the JMXMP connector, although other connectors may be used (e.g. SNMP connector, HTML connector, WSM connector). This is due to the fact that from a client perspective the MBeanServerSurrogate is seen as a regular MBean server.

  Further details about the communication between the MBeanServerSurrogate and remote management applications are given in subsequent chapters whenever appropriate.

In this architecture the BANManagementService creates an MBeanServer to provide its management functionality. In order to be able to participate on the Jini network to offer this service a surrogate object representing the BANManagementService (i.e. the BANManagementSurrogate) has to be provided and uploaded to a Surrogate Host. For this to happen, the device hosting the service and a Surrogate Host must first find one another (see section 2.3.1).

Once the device knows the location of a Surrogate Host a connection is established using the MSP Interconnect and the BANManagementSurrogate is directly or indirectly uploaded. The Surrogate Host is then in charge of activating the surrogate. Once activated, the surrogate can create the MBeanServerSurrogate and establish a new connection with the JMXMSPConnectorServer to take care of the different JMX management interactions, and discover a Lookup Service, create a Service Object for the MBeanServerSurrogate and register it as described in section 2.2.4.

A management application that wants to make use of the management service will first have to discover a Lookup Service and query it to obtain the Service Object to connect to the MBeanServerSurrogate. Once in possession of the Service Object the management application can invoke the methods defined by the service interface directly on it.

It is important to notice that the proposed architecture is transparent from the point of view of the mobile service developer in the sense that in order to make a service remotely manageable it simply needs to be instrumented using JMX by implementing its own specific MBean interface and registered with the BAN MBeanServer. The mobile service developer does not need to know how the communication between the MBeanServer and the MBeanServerSurrogate is handled.

Furthermore, it is also transparent from the management application perspective as long as this is JMX compliant. Once the Service Object that can be used to connect to the MBeanServerSurrogate has been

registered by the BANManagementSurrogate with the Lookup Service and retrieved by the client, the communication is handled in the same way it would if the connection had been established directly with the MBeanServer on the mobile device.

3.3 Evaluation of Requirements

The current design uses the Java Management Extensions as management instrumentation technology that provides all the management capabilities required, and identified in section 3.1. JMX provides attribute, operations, notifications and meta-data management with a well documented and full-featured API. Furthermore, JMX provides a remote API to handle remote management operations and defining different connector types to handle the communications between a client management application and an MBeanServer. The JMX remote API specification defines as well the Generic Connector and Generic Connector Protocol that we use in our design in order to minimize the design and implementation effort.

The proposed architecture is based on and extends MSP which, as discussed in chapter 2, is based on the Jini Surrogate Architecture. The BAN management architecture makes use of the MSP interconnect protocol in order to handle the communications between the BAN management service and its surrogate on the Surrogate Host. This surrogate acts on behalf of the BAN management service and makes use of the resources provided by the Surrogate Host in order to be able to participate in a Jini network, and therefore providing the BAN management service with a way to indirectly participate in a Jini network.

From this, we can see that, except from one, all the requirements identified in section 3.1 are met by the BAN management architecture presented in this research. The last of the requirements identified states that the "BAN management instrumentation must be efficient in terms of memory and CPU usage". This requirement is beyond the scope of this research.

In the following, the JMXMSP connector designed to handle the communications between the BAN MBeanServer and the MBeanServerSurrogate using the MSP interconnect protocol is described in chapter 4.

In chapter 5 the MBeanServerSurrogate that acts on behalf of the BAN MBeanServer and its different components are described.

In chapter 6 the proposed architecture is evaluated by performing a series of tests and the performance of the current design is further evaluated.

In chapter 7 we present the conclusions to this research and give some indications for future work on the subject.

4 The JMXMSP Connector

In the previous chapter, the high level overview of the proposed BAN management architecture has been discussed. One of the components in this architecture is the JMXMSP connector. That is, a communication means between the BAN MBeanServer and the MBeanServerSurrogate. In this chapter the design of this communication connector is further detailed.

4.1 JMXMSP Connector Overview

The JMXMSP Connector supports the communication between the MBeanServer that resides on the MBU and the MBeanServerSurrogate on the Surrogate Host at the JMX communications level. The connector consists of two distributed components: the JMXMSPConnectorServer, which resides on the MBU and the JMXMSPConnector running on the Surrogate Host (see Figure 5).

The JMXMSP Connector is based on the Generic Connector described in [18], which is an optional connector (i.e. its implementation is not mandatory in order to comply with the JMX specification) designed to be configurable by plugging in:

- *A transport protocol*: the protocol used to send requests from the client to the server and responses and notifications from the server to the client.
- *Object wrapping*: the way objects are wrapped in order to be sent from the client to the server, to avoid issues when deserializing them, since they can be of classes just known to the target MBean class loader but not to the class loader of the connector server.

The Generic Connector consists of two classes: the GenericConnector and the GenericConnectorServer. Besides, the specification defines the interfaces to be implemented by new configurations of the connector, in particular the MessageConnection interface, the MessageConnectionServer interface and the ObjectWrapping interface. The implementation of these interfaces provides the pluggable transport protocol to be used by each end of the communication, and takes care of establishing the connection and serializing and deserializing the various messages exchanged between the GenericConnectorServer and the GenericConnector.

Furthermore, the Generic Connector specification describes the Generic Connector Protocol that implementations must follow in order to be able to interoperate with other implementations. This protocol defines the set of messages exchanged between the server and the client sides of the connection and the order in which they must be exchanged [18]. This protocol and the different types of JMX messages defined by the specification are further described in section 4.3.

In our implementation we use the MSP Interconnect implementation as transport protocol for our Generic Connector configuration to support the interactions between the client and the server ends of the connection and native Java serialization as object wrapping. This implementation consists of three packages: Messages, IO and Interconnect.

These packages provide the interfaces that define the way the MSP Interconnect protocol handles the communications between a device service and its surrogate. Therefore, in order to use the MSP Interconnect as transport protocol for the JMXMSP Connector implementation, these interfaces must be used. The most relevant interfaces for our implementation, grouped by package, are presented here:

- *MSP IO package*: within this package the MSPSurrogateConnection and MSPSurrogateHostConnection interfaces define the methods used to handle the communication on the device service side of the connection, and the MSPDeviceService interface has to be implemented by any device service.
- *Interconnect package*: especially relevant to our implementation is the InterconnectSession interface, which defines how the communication is handled on the surrogate side of the connection. Besides, the LivenessHandler interface defines the method to be used to make sure a device service is still alive.
- *Messages package*: this package includes MSP Message related interfaces defining the different type of messages and encoding/decoding functions, as well as the MSPServant interface for handling incoming messages through the connection it is attached to.

In order to establish a connection between the BAN management service and its surrogate, it is necessary to obtain first a connection between the service and the Surrogate Host so that the surrogate can be uploaded and activated. This can be accomplished by using a ServiceHostConnection and supplying the URL of the Surrogate Host location. If the location of the Surrogate Host is known to the device service this URL can be supplied directly, otherwise the Surrogate Host can be located by means of the discovery mechanism provided by the MSP Interconnect protocol.

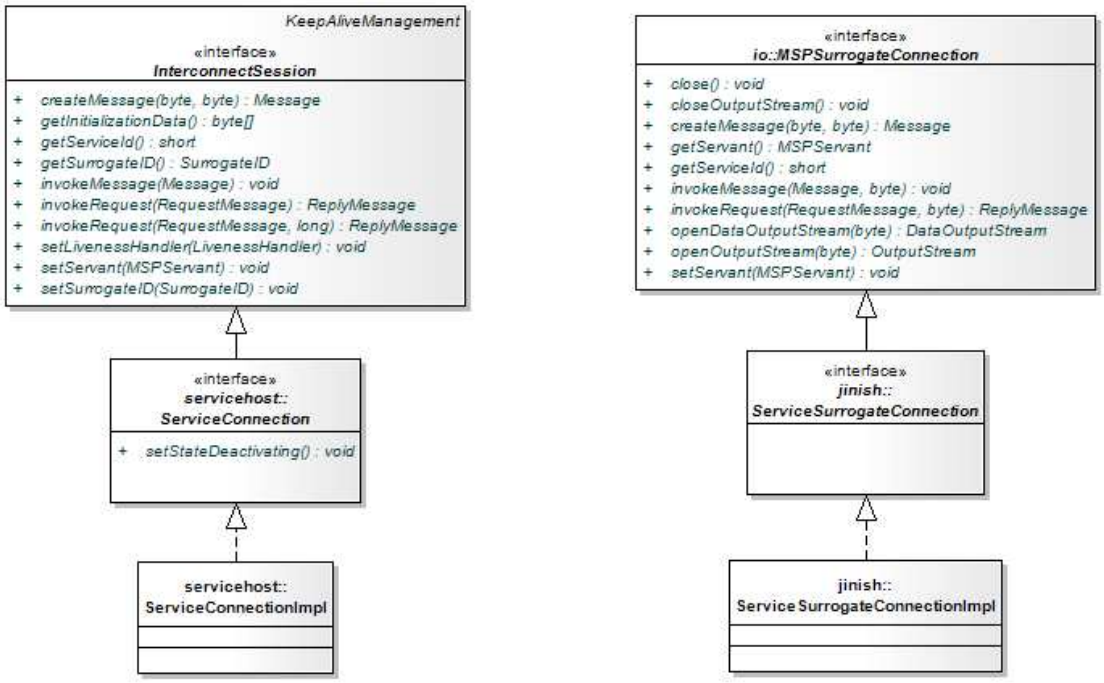

Figure 6. ServiceConnection and ServiceSurrogateConnection

Once the ServiceHostConnection has been established its registerSurrogate method can be used to obtain a ServiceSurrogateConnection. This method accepts as an argument an MBUService object, which is an instance of the device service to be registered. This object must provide the location of the surrogate so that it can be retrieved by the Surrogate Host upon registration. The surrogate is usually packed for convenience in a Jar file and located in a web server.

Note that the ServiceHostConnection represents the connection between the BANManagementService and the Surrogate Host, whereas the ServiceSurrogateConnection is the connection between the BANManagementService and the BANManagementSurrogate. For a UML representation of both connection classes see Figure 6.

When the device surrogate has been uploaded, the Surrogate Host takes care of calling the surrogate activate method that will in turn create a ServiceConnection representing the connection between the surrogate and the corresponding device service. Once the service has a ServiceSurrogateConnection and the surrogate has a ServiceConnection, the service and its surrogate are ready to interact using the messages defined by the MSP messages package. This connection will be used by the JMXMSP Connector in order to handle the JMX management operations exchanged between the BAN MBeanServer and the MBeanServerSurrogate as specified by the Generic Connector Protocol. See section 4.3 for more details.

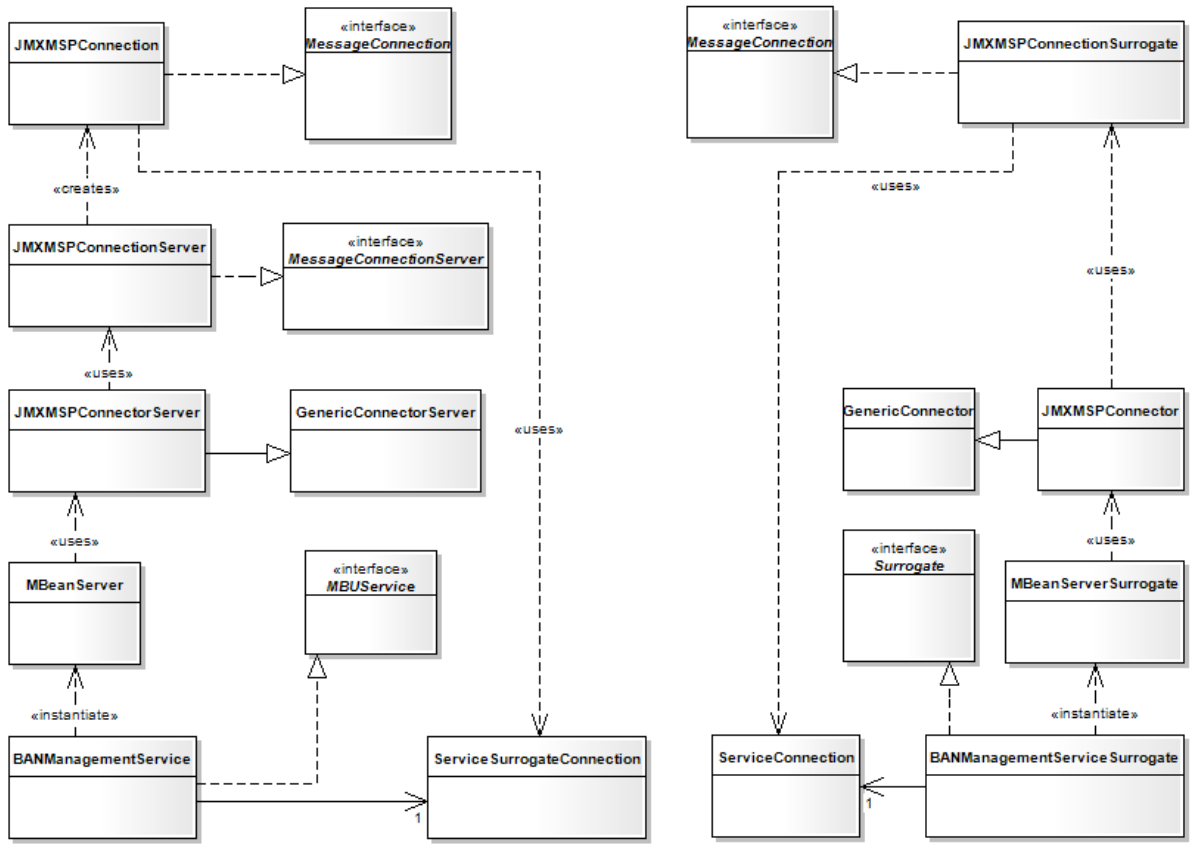

Figure 7. BAN management architecture overview

The JMXMSP Connector implementation mainly consists of the JMXMSPConnection, JMXMSPConnectorServer and JMXMSPConnectionServer classes on the server side (MBU) and the JMXMSPConnector and JMXMSPConnectionSurrogate classes on the client side (Surrogate on the Surrogate Host). These classes make use of the different classes and interfaces provided by MSP and described above in order to use the MSP Interconnect as transport protocol.

The relation between these classes and the interfaces they implement is shown in Figure 7.

4.2 JMXMSP Connector Components

Once the BAN management service has a ServiceSurrogateConnection and the MBeanServerSurrogate service has a ServiceConnection and the surrogate has been activated, the MBeanServerSurrogate has to establish a new connection with the JMXMSPConnectorServer attached to the BAN MBeanServer to take care of the different JMX management operations.

This process can be divided in three different phases:

1. *JMX connection setup*: First a JMXMSPConnectorServer is created and attached to the BAN MBeanServer to listen for client connection requests. Then the MBeanServerSurrogate creates a JMXMSPConnector and initiates a connection handshake in order to establish a connection with the BAN MBeanServer. If the connection handshake succeeds, the connection to handle the JMX management operations between the BAN MBeanServer and the MBeanServerSurrogate has been successfully created. The BAN MBeanServer is then
2. *Management phase*: This phase is where a management application interacts with the BAN MBeanServer via the MBeanServerSurrogate by invoking operations on the different MBeans registered with the BAN MBeanServer, modifying their attributes, receiving notifications, etc.
3. *JMX connection close-down*: Either side of the connection can at any moment close the connection established by sending a CloseMessage to the other end. This may be done, for instance if either side receives a message in the wrong order when performing the connection handshake [18]. After the connection has been closed the JMXMSPConnector should initiate again the JMX connection setup phase in order to be able to exchange management operations again.

The different components of the JMXMSP Connector involved in this connection setup and its usage, as well as their interactions are discussed in this section.

4.2.1 JMXMSPConnectorServer and JMXConnector

The JMXMSPConnectorServer class extends the GenericConnectorServer and is attached to the BAN MBeanServer (see Figure 8) in order to listen for client connection requests and it creates a connection for each request. In our design a reference to the BAN MBeanServer object is passed as an argument to the JMXMSPConnectorServer constructor, together with a reference to the JMXMSPConnectionServer

object, since they will be needed in order for the JMXMSPConnectorServer constructor to call its parent's constructor.

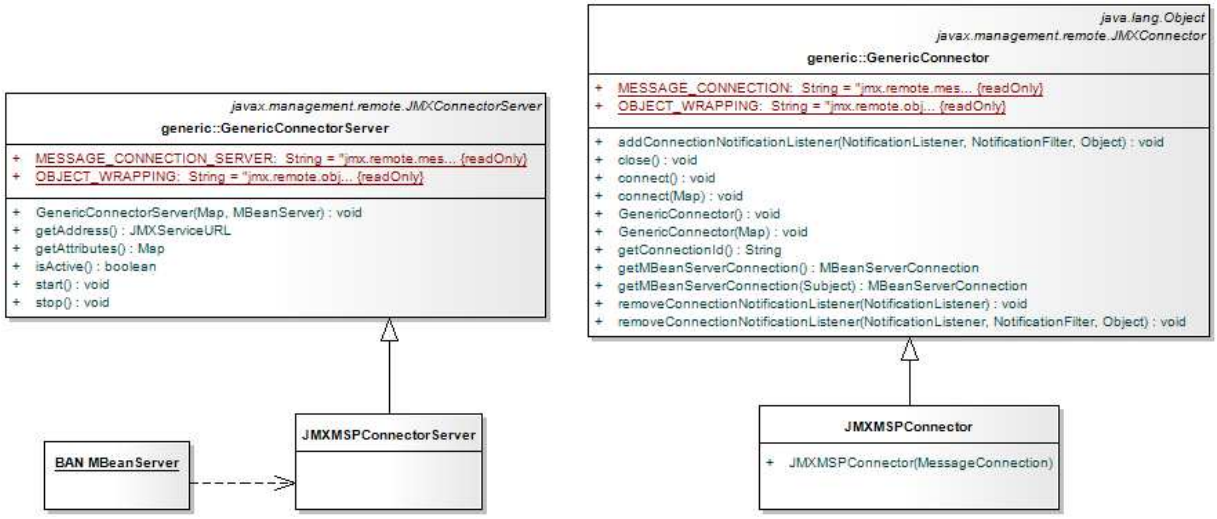

Figure 8. JMXMSPConnectorServer and JMXMSPConnector.

The JMXMSPConnectorServer is created in the BANManagementService and its constructor calls the constructor of its parent class (i.e. the GenericConnectorServer) with two parameters: an environment Map and the MBeanServer the JMXMSPConnectorServer is attached to. This environment Map can contain two attributes: a *MESSAGE_CONNECTION_SERVER* attribute specifying how connections are made to this connector server and an *OBJECT_WRAPPING* attribute specifying the type of object wrapping for parameters whose deserialization requires special treatment. In our design, just the *MESSAGE_CONNECTION*_SERVER attribute is set in order to use the JMXMSPConnectionServer provided when creating the JMXMSPConnectorServer.

As soon as the connector's start method is called it will start listening for client connections, and will keep doing so until its stop method is called. Every time a client connection is established or broken, a notification of class JMXConnectionNotification is emitted.

The JMXMSPConnector defines the client side of the connector. This class extends the GenericConnector class and its implementation is similar to that of the JMXMSPConnectorServer. The constructor takes a JMXMSPConnectionSurrogate as an argument that is used to set the *MESSAGE_CONNECTION* attribute when calling the GenericConnector's constructor method.

4.2.2 JMXMSPConnectionServer

The JMXMSPConnectionServer implements the MessageConnectionServer interface (see Figure 9), which is the interface that specifies how the connector server establishes new connections with the clients. In our design an instance of JMXMSPConnectionServer is communicated to the JMXMSPConnectorServer using the *MESSAGE_CONNECTION_SERVER* attribute of the constructor's

environment Map. This is done in order to specify how the connector server creates new connections to clients that will use our JMXMSPConnection described in the following section.

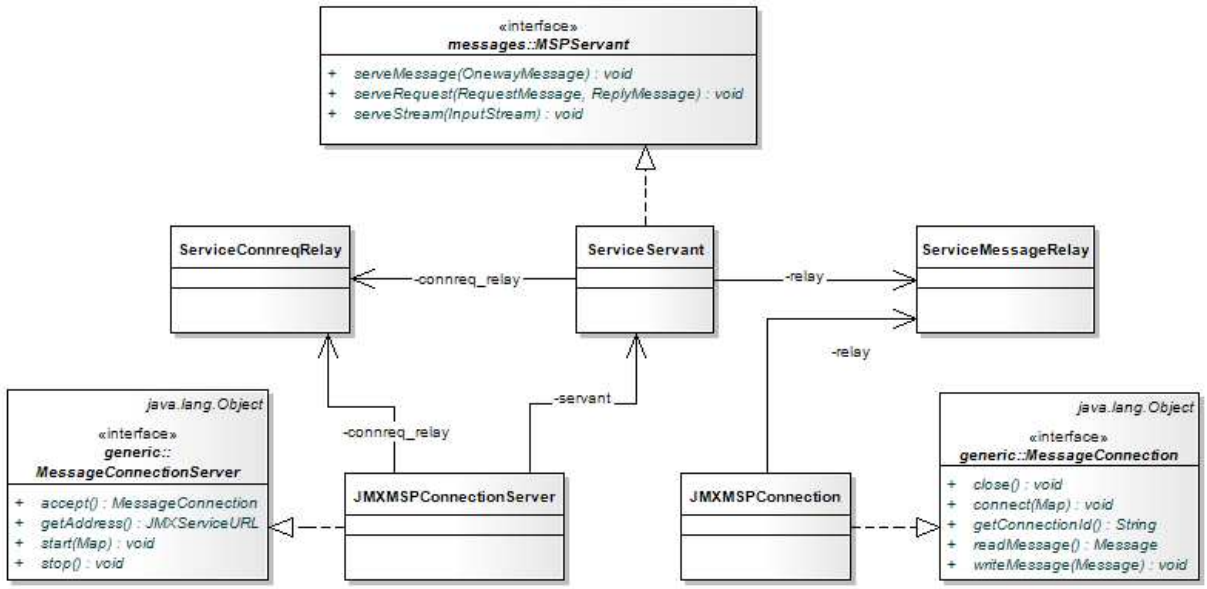

Figure 9. JMXMSPConnectionServer and related components of the JMXMSP Connector.

When a new JMXMSPConnectionServer is created by the BANManagementService, a reference to a ServiceSurrogateConnection obtained by the BANManagementService is supplied to its constructor. Every new JMXMSPConnection created by the JMXMSPConnectionServer will make use of the ServiceSurrogateConnection to communicate with the other end of the connection.

But in order to create new JMXMSPConnection connections, we need some kind of mechanism to be able to handle incoming connection requests. The JMXMP connector, which is the connector based on the Generic Connector that uses TCP as underlying transport protocol, solves this issue using TCP sockets. Like any other MessageConnectionServer implementation, the SocketConnectionServer used by the JMXMP connector implements an accept method. This method handles the creation of new connections returning a new MessageConnection whenever a new connection request is received by the MessageConnectionServer. The JMXMP connector implementation of the accept method uses a java.net.ServerSocket object in order to listen for incoming connections in a specified TCP port. The ServerSocket class provides an accept method that listens for a connection to be made to the socket. This method blocks until a connection is requested. At this point the method accepts the connection and returns a java.net.Socket that is used to instantiate the new SocketConnection [19]. The sequence diagram in Figure 10 illustrates this process.

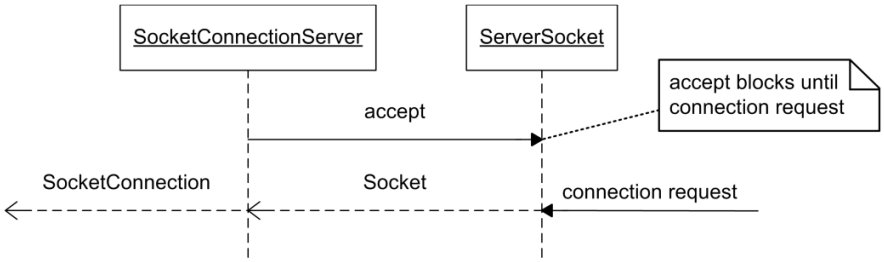

Figure 10. JMXMP connector connection creation.

In our design we emulate this behavior using a ServiceConnreqRelay object. This object serves as a relay for incoming connection requests. It implements a put method and a take method. When a connection request arrives at the BANManagementService via the ServiceSurrogateConnection it is handled by a ServicServant object (described in detail in section 4.2.4) which places the connection request message on the ServiceConnreqRelay using its put method.

On the other hand, our JMXMSPConnectionServer's accept method implementation calls the JMXMSPConnection's constructor method in order to create a new JMXMSPConnection. This constructor uses as one of its parameters the return value of the ServiceConnreqRelay's take method (see Listing 1). Since the take method on the relay will block until a connection request message has been put on the relay using its put method, the JMXMSPConnection's constructor method will block as well until the take method returns, and thus the JMXMSPConnectionServer will remain idle until then as well.

Listing 1. JMXMSPConnection creation using the ServiceConnreqRelay's take method.

```
// The JMXMSPConnection's constructor will return as soon as
// relay.take() returns
JMXMSPConnection jmxmsp_conn = new
JMXMSPConnection(serviceSurrogateConnection, relay.take());
```

When the ServiceConnreqRelay's take method returns (indicating that a connection request has arrived at the relay) a new JMXMSPConnection instance is created by the JMXMSPConnectionServer. A sequence diagram showing the JMXMSPConnection creation described is shown in Figure 11.

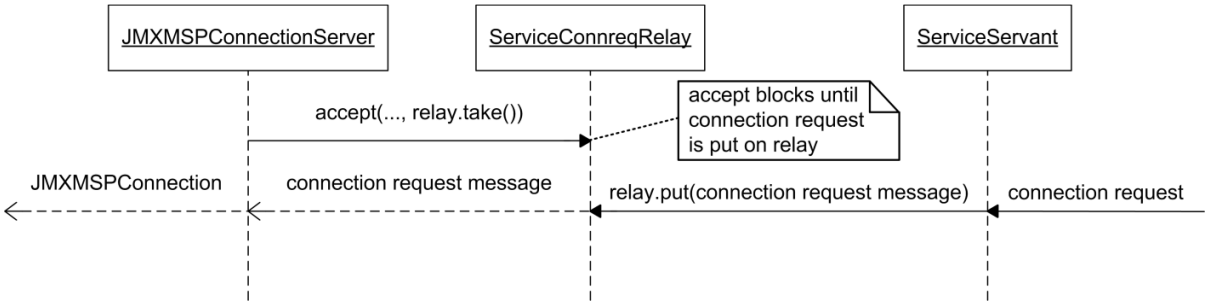

Figure 11. JMXMSPConnection creation.

Once a new JMXMSPConnection has been created a second relay similar to the ServiceConnreqRelay and providing the same take and put methods is instantiated. This new relay is called ServiceMessageRelay and is created in order to handle any incoming JMX messages that arrive at the ServiceServant that is attached to the ServiceSurrogateConnection.

As it can be seen in Figure 9, besides the accept method, the JMXMSPConnectionServer class must implement the getAddress, start and stop methods. In our design the start method simply sets an *isActive* Boolean flag to true indicating that the JMXMSPConnectionServer accepts new connections and the stop method sets *isActive* to false so that new connection attempts are refused. This behavior is designed following the MessageConnectionServer interface description provided by the JMX Remote API 1.0[20].

The getAddress method returns a JMXServiceURL object representing the address of the JMXMSPConnectionServer. As described in chatper 3, in our design we want that management applications that want to connect to the BAN MBeanServer to manage its resources they do so via the MBeanServerSurrogate running on the Surrogate Host. With this in mind, the getAddress implementation on the JMXMSPConnectionServer returns null to avoid client applications connecting directly to the BAN MBeanServer.

4.2.3 JMXMSPConnection and JMXMSPConnectionSurrogate

Both the JMXMSPConnection and JMXMSPConnectionSurrogate implement the MessageConnection interface. This interface specifies the full-duplex transport used by both ends of the connection of a JMXMSP Connector to communicate with each other. Although similar, the JMXMSPConnection and JMXMSPConnectionSurrogate implementations are slightly different and have been kept as different classes for convenience, instead of extending yet another class, but with some modifications they could be implemented as a single class, using different constructors.

The JMXMSPConnection class is the MessageConnection implementation returned by the JMXMSPConnectionServer when a connection request has been received. This class implements, among others, the readMessage and writeMessage methods. The readMessage method is used to handle incoming JMX messages that arrive at the ServiceMessageRelay via the ServiceSurrogateConnection's ServiceServant. This method reads an incoming message using the ServiceMessageRelay's take method. As described in the previous section, the take method will block until the ServiceServant attached to the ServiceSurrogateConnection receives an incoming message and places it on the ServiceMessageRelay using its put method. The ServiceServant will place incoming messages on the ServiceMessageRelay just if they are JMX operation messages. A sequence diagram illustrating how the JMXMSPConnection handles incoming JMX operation messages is shown in Figure 12.

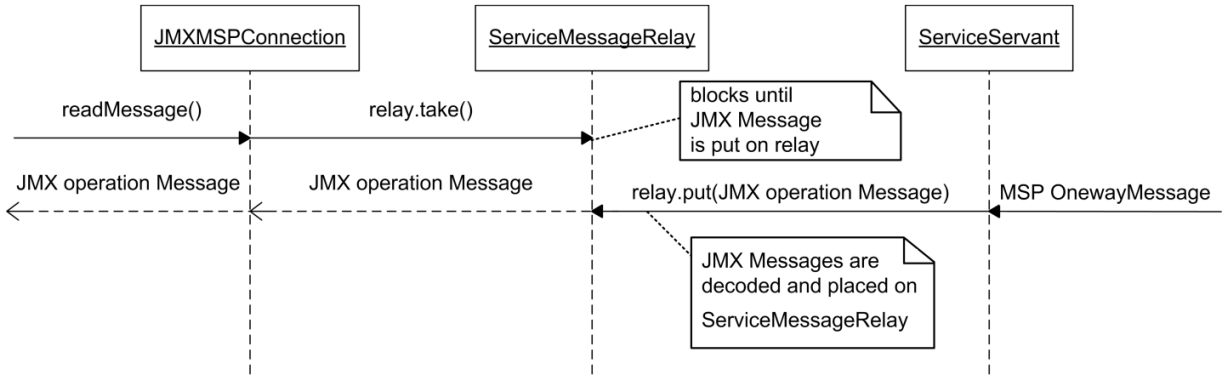

Figure 12. Incoming JMX operation message handling.

On the other hand, the writeMessage method encapsulates the different JMX operation messages to be sent into MSP One-way messages by creating a OnewayMessage object and including in its body the JMX operation message encoded as a byte array. An OperationID is set on the OnewayMessage in order to identify the type of JMX operation message included. The resulting MSP message is sent via the connection using the ServiceSurrogateConnection's invokeMessage method. Figure 13 illustrates how the outgoing JMX operation messages are handled by the JMXMSPConnection.

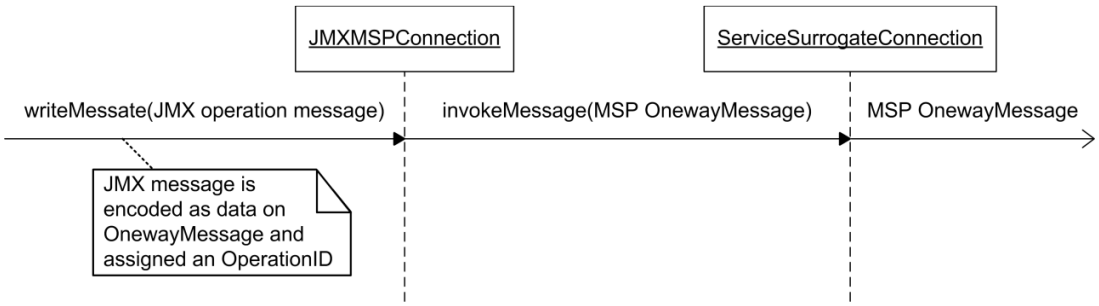

Figure 13. Outgoing JMX operation message handling.

Furthermore, the JMXMSPConnection class must implement the methods connect, close and getConnectionId as specified by the MessageConnection interface. In our implementation getConnectionId returns a default string identifier, but could be implemented to return specific connection ID's or even a random identifier. No action is executed by our close method implementation. This is due to the fact that we are using the ServiceSurrogateConnection as the underlying connection to send messages via the JMXMSPConnection and closing the ServiceSurrogateConnection would mean losing the connection between the BANManagementService and the BANManagementSurrogate and not just the connection between both connector ends. Furthermore, no action executed by our JMXMSPConnection's connect method implementation since it is the JMXMSPConnector that takes care of initiating the handshake to establish the connection.

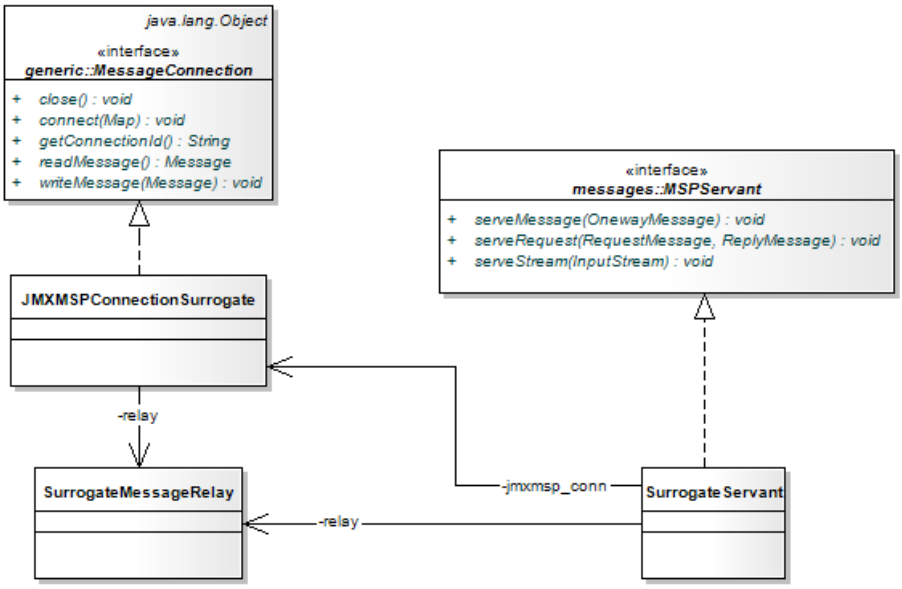

Figure 14. JMXMSPConnectionSurrogate and related components of the JMXMSP Connector.

At the other side of the connection, the MBeanServerSurrogate creates a JMXMSPConnectionSurrogate used to create the JMXMSPConnector to connect to the connector server. Figure 14 shows the JMXMSPConnectionSurrogate and other related components of the JMXMSPConnector. The JMXMSPConnectionSurrogate class implements the readMessage, writeMessage, close and getConnectionId much in a similar way as the JMXMSPConnection by using a ServiceConnection and a SurrogateMessageRelay instead and implementing the connect method. This method is used to send an MSP message of type OnewayMessage with MSG_TYPE_CONNECTION_REQUEST as operation Id to identify the message as a connection request. This message is sent via the ServiceConnection using the invokeMessage method provided by MSP (see Figure 15).

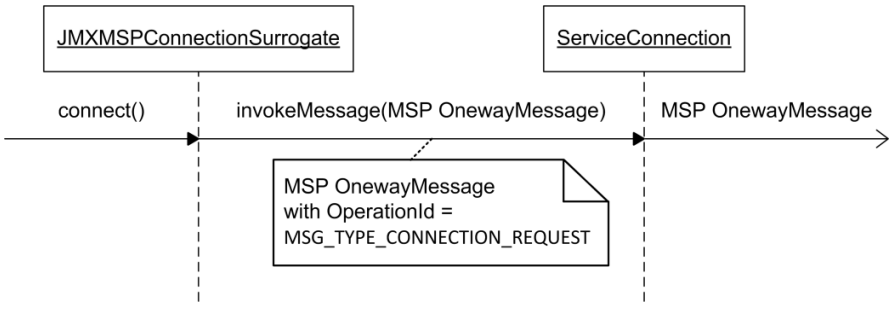

Figure 15. JMXMSPConnectionSurrogate initiates the connection handshake sending a OnewayMessage with OperationId = MSG_TYPE_CONNECTION_REQUEST.

4.2.4 ServiceServant and SurrogateServant

The MSPServant interface from the MSP messages package defines the methods to be implemented for handling incoming messages through the connection it is attached to. In our implementation this

interface is implemented by the ServiceServant and the SurrogateServant classes, the first one is designated to handle messages received by the BANManagementService and the latter to handle the messages received by the BANManagementSurrogate.

The MSPServant interface defines the following methods:

- serveMessage: method called by the object implementing MSPServant when a message is received.
- serveRequest: method used to handle incoming requests and their corresponding response messages.
- serveStream: method used to handle continuous streams of data.

For our design we require just the serveMessage. This method is called on the ServiceServant whenever an invokeMessage is called on the ServiceConnection on the surrogate, and on the SurrogateServant when the invokeMessage is called on the ServiceSurrogateConnection on the MSP management service.

The serveMessage method checks the OperationID of the incoming message in order to decide what to do. If the OperationID is of type *MSG_TYPE_CONNECTION_REQUEST*, the message is a connection request simply containing a string that will be put on the ServiceConnreqRelay. In turn, the ServiceConnreqRelay's take method will return and enable the creation of a JMXMSPConnection on the JMXMSPConnectionServer's accept method.

For any other OperationID received, indicating any of the Generic Connector Protocol messages, the incoming OnewayMessage will be decoded and cast into a javax.management.remote.message.Message and put on the ServiceMessageRelay for the JMXMSPConnection to handle using the readMessage method. Further details about the Generic Connector Protocol and the different messages exchanged are given in section 4.3.

4.3 Network Message Exchange

Using the MSP Interconnect protocol to exchange JMX messages between the BAN MBeanServer and the MBeanServerSurrogate requires that these messages are wrapped inside MSP messages. This involves encoding the different JMX messages as a byte array that is added to the MSP message body together with the type of operation it represents so that the receiving MSPServant can decide what to do.

MSP provides three types of messages within the messages package: RequestMessage, ReplyMessage and OnewayMessage. On the other hand, the Generic Connector Protocol specification provided in [18] defines the following types of message:

- CloseMessage
- HandshakeBeginMessage
- HandshakeEndMessage

- HandshakeErrorMessage
- MBeanServerRequestMessage
- MBeanServerResponseMessage
- NotificationRequestMessage
- NotificationResponseMessage
- VersionMessage

In our implementation we just require the OnewayMessage type provided by MSP and make use of all the message types defined by the Generic Connector Protocol.

MSP messages include a ServiceID that uniquely identifies the service sending the message (or the service the message is destined for, when the message is sent from the surrogate), an OperationID identifying the type of operation the message represents so that the receiving MSPServant can handle it accordingly and a SequenceID identifying the order number of the message in order to match request-response interactions. Furthermore each MSP message has a body containing data specific to the operation to be performed by the message.

In order to wrap a JMX message in a OnewayMessage we define a JMX message – OperationID mapping. This mapping is shown in the following table:

Table 1. JMX messages to MSP OperationID's mapping.

JMX Generic Connector Protocol message	OperationID used by the receiving MSPServant
CloseMessage	MSG_TYPE_CLOSE
HandshakeBeginMessage	MSG_TYPE_MBS_HANDSHAKE_BEGIN
HandshakeEndMessage	MSG_TYPE_MBS_HANDSHAKE_END
HandshakeErrorMessage	MSG_TYPE_MBS_HANDSHAKE_ERROR
MBeanServerRequestMessage	MSG_TYPE_MBS_REQUEST
MBeanServerResponseMessage	MSG_TYPE_MBS_RESPONSE
NotificationRequestMessage	MSG_TYPE_NOTIFICATION_REQUEST
NotificationResponseMessage	MSG_TYPE_NOTIFICATION_RESPONSE

These messages are exchanged between the BAN MBeanServer and the MBeanServerSurrogate following the Generic Connector Protocol specification but especially encoded as OnewayMessages with the appropriate OperationID. The specification states that the connection handshake is initiated by the server side of the connection as soon as the connect method of the JMXConnector class has been called by the client. In our implementation this method is implemented by the JMXMSPConnectionSurrogate since we are using a custom MessageConnection implementation. Once this method has been called and the connection request has been received, the server side sends a HandshakeBeginMessage in order to initiate the handshake and establish the connection. The message exchange that follows is shown in Figure 16, taking into account the message mapping described.

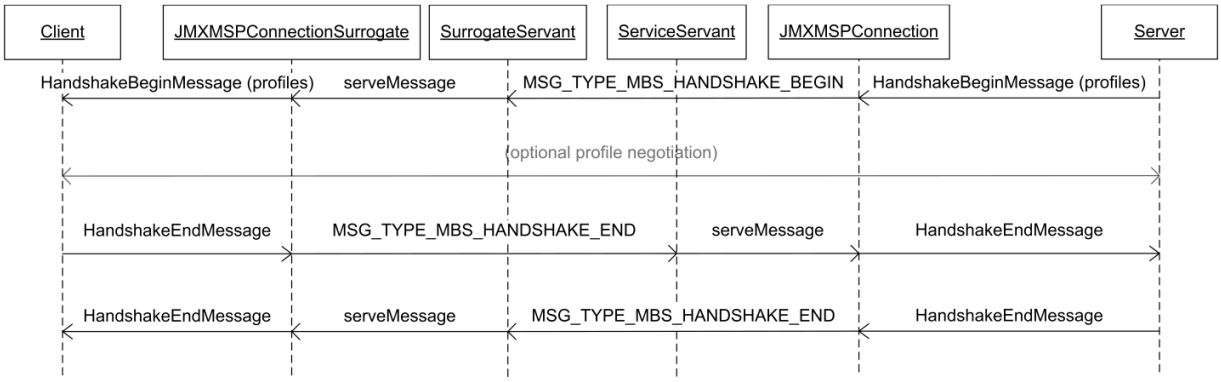

Figure 16. Handshake message exchange.

Note that Figure 16 shows an optional profile exchange. This profile exchange has not been included in the current design but could be used in order to negotiate the different profiles supported by the server and the client. For further details we refer the reader to [18].

If either side receives a misplaced message (i.e. not in the right sequence), the side receiving the message issues a HandshakeErrorMessage and immediately closes the connection. Once the connection handshake has been completed successfully the client can obtain a reference to the BAN MBeanServer using the getMBeanServerConnection method on the JMXMSPConnector instance. Using the MBeanServerConnection obtained the client can perform operations on the MBeans registered with the BAN MBeanServer (i.e. the BAN resources that have been exposed for management).

These operations are the different method invocations on the MBeanServerConnection and are wrapped inside MBeanServerRequestMessage objects and written to the server using the JMXMSPConnection.writeMessage method. In our implementation, in the writeMessage method, these MBeanServerRequestMessage objects are in turn wrapped inside OnewayMessages encoded as byte arrays on the body of the MSP message and with the appropriate OperationID using the JMXMSPConnectionSurrogate. When the ServiceServant decodes the message and the BAN MBeanServer receives the MBeanServerRequestMessage, the MBeanServer will perform the specified operation and return the result in an MBeanServerResponseMessage and transmitted as a OnewayMessage with the appropriate OperationID. JMX notification messages are exchanged in the same way. The MBeanServer operations and notifications exchanged are shown in Figure 17.

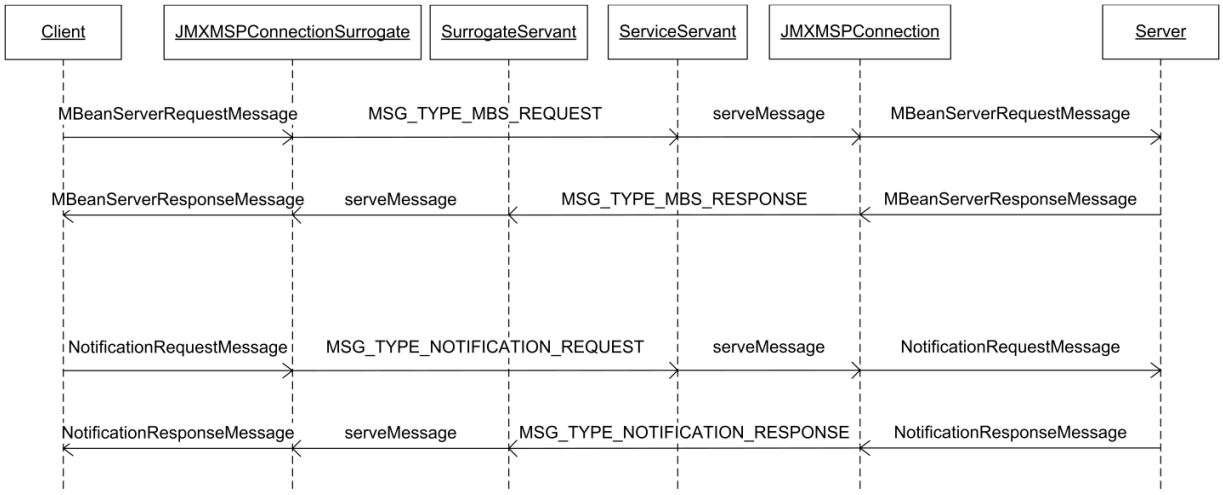

Figure 17. MBeanServer operations and notifications exchange.

5 MBeanServerSurrogate

In Chapter 3, we have seen that in order to comply with the MSP architecture, a service running on the mobile device (MBU) needs a surrogate object running on a Surrogate Host and acting on its behalf in order to provide its service to interested clients elsewhere in the internet. In our architecture we have the BAN management service running on the mobile device and its corresponding surrogate on the Surrogate Host. In this chapter, we further detail the design of the BAN management surrogate and its main component, the MBeanServerSurrogate, and describe the design choices made in order to comply with the requirements identified in section 3.1.

5.1 BAN Management Surrogate Overview

As soon as the Surrogate Host receives a registration request from the BANManagementService it tries to obtain its surrogate from the location specified by a URL included on the request. This URL points to the JAR file with the different components of the surrogate, and in our architecture is located on an accessible directory of an HTTP server.

This surrogate contains the BANManagementSurrogate class that implements the net.jini.surrogate.Surrogate interface, as specified by [12]. The Surrogate interface defines the activate and deactivate methods that the Surrogate Host uses to control the surrogate execution. The activate method is used to activate the surrogate and allocate the resources needed by the BAN management surrogate, such as creating a ServiceConnection, a JMXMSPConnector or the MBeanServerSurrogate. This method is called once by the Surrogate Host and must return on a timely manner, otherwise the surrogate will be discarded and the deactivate method called to deactivate it [12].

The activate method needs two parameters: a hostContext and a context. The hostContext is an object implementing the net.jini.surrogate.HostContext interface and supplied by the Surrogate Host which defines the methods to access the execution environment of the surrogate that is being activated.

The context parameter provides access to the specific context for the surrogate being activated and depends on the interconnect used for the surrogate-device communication. Since our design uses the MSP Interconnect, the context contains a java object that can be cast into a ServiceConnection. As shown in Figure 6 this ServiceConnection implements the InterconnectSession and KeepAliveManagement interfaces.

The KeepAliveManagement interface defines the method setKeepAliveHandler used to specify a keep-alive handler for the surrogate in order to take care of its *liveness* obligations. This handler implements the net.jini.surrogate.KeepAliveHandler interface which defines a keepAlive method. In our BAN management surrogate this interface is implemented by the LifeHandler class, which also implements the LivenessHandler interface provided by MSP and which defines the isAlive method.

The isAlive method from our LifeHandler is called whenever a keep-alive message is received from the surrogate. The keepAlive method in our implementation is called periodically by the KeepAliveTask

component from the MSP Interconnect every 4000 milliseconds to request the surrogate to check if it still has a connection with the device. If the keepAlive method detects that the period since the last time a keep-alive message was received exceeds the 4000 miliseconds, it calls the serviceTimedOut method, which in turn calls the cancelActivation method on the hostContext object provided by the Surrogate Host. The UML diagram shown in Figure 18 gives an overview of the BAN management surrogate without the JMXMSP connector components.

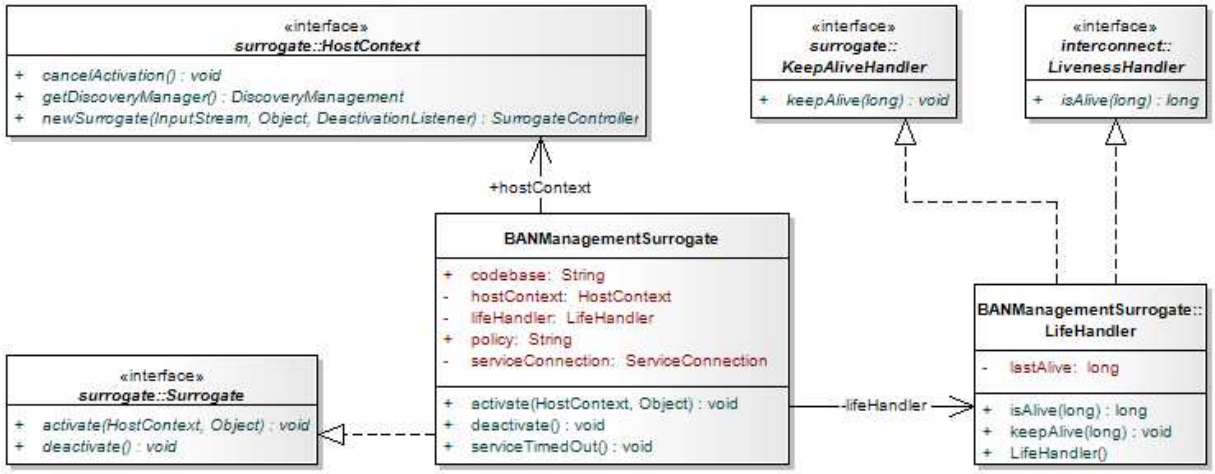

Figure 18. BAN management surrogate overview

Once the Surrogate Host has downloaded the BAN management surrogate and activated it, the surrogate needs to obtain a connection to the BANManagementService by creating a ServiceConnection from the context parameter provided by the Surrogate Host on the activate method. When this connection has been established, and in order to be able to interact with the BAN MBeanServer and exchange the different JMX management operations the MBeanServerSurrogate needs a JMXMSPConnector. The JMXMSPConnector makes use of a JMXMSPConnectionSurrogate that defines the transport protocol and makes use of the ServiceConnection as underlying transport protocol as described in section 4.2.3. Once the ServiceConnection has been established and the JMXMSP Connector is in place, the surrogate is ready to create the MBeanServerSurrogate object that will act on behalf of the BAN MBeanServer. A LifeHandler object is then created and both the KeepAliveHandler and LivenessHandler properly set using the methods provided by the ServiceConnection. Then the ServiceConnection created is set on the MBeanServerSurrogate so that it can make use of it to communicate with the BAN MBeanServer and the MBeanServerSurrogate is started. The sequence diagram illustrating this process is shown in Figure 19:

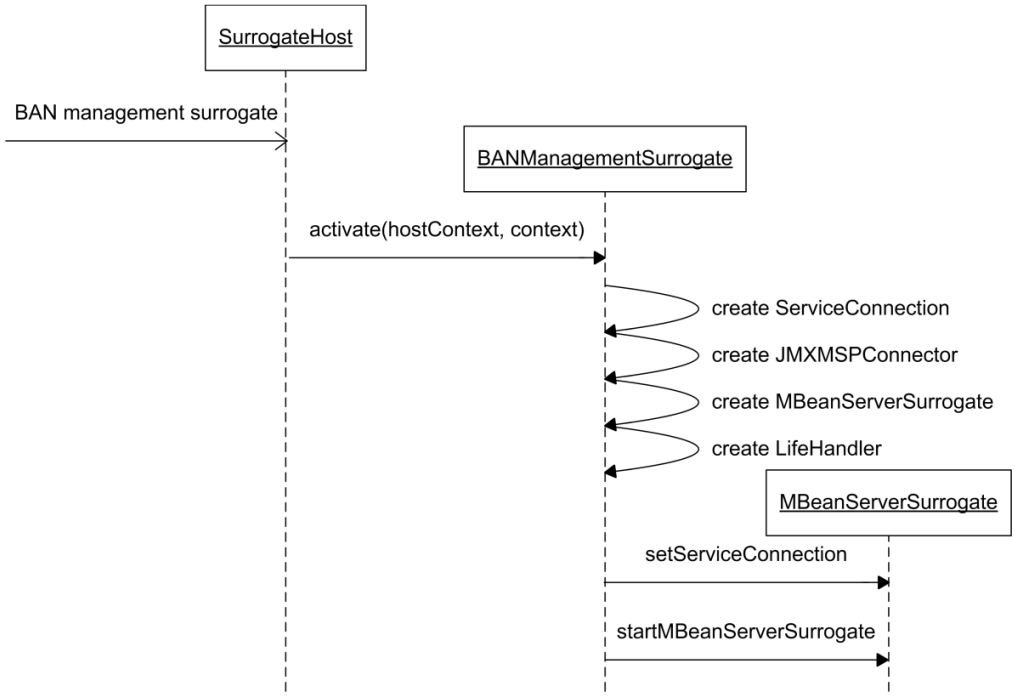

Figure 19. BANManagementSurrogate retrieval and activation and MBeanServerSurrogate creation.

Besides the different interfaces that have been mentioned so far, [12] specifies a few other interfaces included in the net.jini.surrogate package that either have not been implemented or have not been directly used In the current design of the BAN management surrogate. Here we give a brief description of these interfaces for the sake of completion:

- GetCodebase: this interface provides a method to retrieve the codebase annotation. If the surrogate class implements this interface, the Surrogate Host invokes the getCodebase method and uses the set of URL's returned to set the codebase for the surrogate before calling the activate method.
- DeactivationListener: an interface that allows a parent surrogate to be notified whenever one of its child surrogates has been deactivated.
- SurrogateController: this interface allows a parent surrogate to deactivate a child surrogate.

The getCodebase interface has not been implemented because in the current design it is not necessary for the BAN management surrogate to specify any codebase since all the necessary resources for the surrogate to be able to be activated are contained in the surrogate JAR.

As for the DeactivationListener and SurrogateController interfaces, they have not been implemented due to the fact that they are intended to provide functionality for Surrogate with parent-child relationships (i.e. a surrogate with either a parent surrogate or a child surrogate). For more information on these interfaces we refer the reader to [12].

In order for the Surrogate Host to easily obtain the surrogate specified on a registration request, the surrogate contents are packaged on a JAR file that contains:

- A manifest file with, at least, a single, mandatory Surrogate-Class header that specifies the name of the surrogate class and an optional Surrogate-Codebase header, specifying the resources of the surrogate's codebase. These resources may be Java classes, as well as any other data such as HTML files or icons.
- The class implementing the Surrogate interface together with any other resources necessary for the surrogate implementation.

In our implementation this JAR file mainly consists of the different class files necessary for the MBeanServerSurrogate to be able to use the JMXMSP Connector and the class files of the BAN management surrogate itself[3].

The different components of the JMXMSP Connector used by the surrogate have been already described in detail in the previous chapter. The remaining of this chapter is dedicated to describe the remaining components of the BAN management surrogate.

5.2 MBeanServerSurrogate

Once the JMXMSPConnector has been properly set, the BANManagementSurrogate class creates an instance of the MBeanServerSurrogate. The MBeanServerSurrogate is the component that acts on behalf of the BAN MBeanServer running on the mobile device. This means that it must be able to perform the same operations the BAN MBeanServer is capable of performing, therefore the MBeanServerSurrogate should implement the MBeanServer interface. A class diagram showing this interface is shown in Figure 20.

One possibility for the MBeanServerSurrogate to be able to act on behalf of the BAN MBeanServer would be to implement the MBeanServer interface by extending the MBeanServer base class (located in the package com.sun.jmx.mbeanserver). But there is another possibility that allows us to have the MBeanServerSurrogate act on behalf of the BAN MBeanServer and that offers a convenient way to intercept method calls originally destined for the MBeanServerSurrogate and interpose additional behavior before rerouting the call to the BAN MBeanServer. This technique is base on the proxy pattern and is the technique we explore in our MBeanServerSurrogate design. For more information on the proxy pattern we refer the reader to [21].

[3] Other classes that are not directly relevant to the contents of this research are also included in the file providing some utility classes that are part of the Apache Commons project, from the BeanUtils, Collections and Digester components. For more information on these classes, we refer the reader to [25].

Figure 20. MBeanServer interface.

In Java the proxy pattern can be implemented using dynamic proxies, which are part of the java.lang.reflect package. A UML diagram describing the proxy pattern is shown in Figure 21, and a sequence diagram describing the interactions between the different components involved is shown in Figure 22.

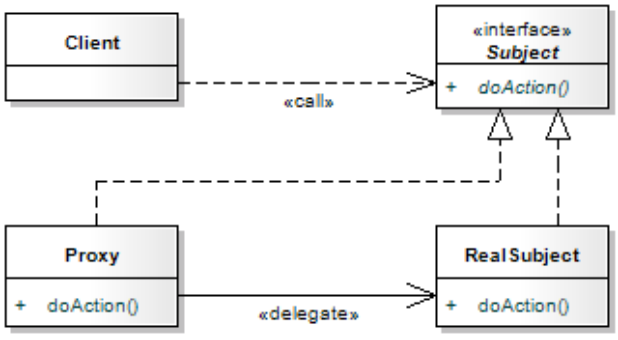

Figure 21. UML diagram of the proxy pattern.

Using dynamic proxies allows us to create a class that implements one or more interfaces specified at runtime in a way that any method invocation through one of its interfaces on an instance of the class is encoded and forwarded to the object being proxied. In order to create a proxy instance, the static

method java.lang.reflect.Proxy::newProxyInstance() can be used. This method accepts three parameters:

- The class loader to define the proxy class.
- The list of interfaces that the proxy instance implements.
- An invocation handler object implementing the InvocationHandler interface that intercepts and handles the method calls.

The Proxy.newProxyInstance method returns a proxy instance with the specified invocation handler of a proxy class that is defined by the specified class loader and that implements the specified interfaces.

The invocation handler implements the invoke method defined by the InvocationHandler interface, which will be called whenever a method is invoked on the proxy instance the invocation handler is associated with. The method invocation is encoded using a java.lang.reflect.Method object identifying the interface method invoked on the proxy instance and an array containing the values of the arguments passed in the method invocation on the proxy instance. The value returned by the invoke method is then the return value of the method that was invoked on the proxy instance.

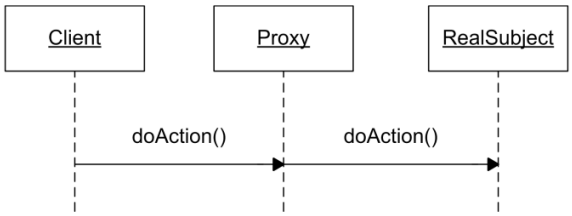

Figure 22. Proxy pattern sequence diagram.

The MBeanServerSurrogate is implemented using dynamic proxies in order to be able to create a proxy instance for the MBeanServer interface with a custom invocation handler. This allows us to intercept the methods that are invoked on the MBeanServer proxy instance and add some additional behavior before rerouting the method call to the BAN MBeanServer. This additional behavior may be for example a security check to see if the client invoking the method is allowed to access the BAN MBeanServer for management or the caching functionality described in section 5.4 and implemented in the current design.

The MBeanServerSurrogate class contains two static member classes: the MBSSurrogateMethodsMap and the MBSSurrogateHandler. The former class contains a Map which maps each java.lang.reflect.Method method defined on the MBeanServer interface to their corresponding MBeanServerConnection equivalent. This mapping is necessary because the MBeanServerConnection used to talk to a remote MBeanServer does not define all the methods defined by the MBeanServer interface which represents a local MBeanServer (and which extends the MBeanServerConnection interface). This can be seen comparing the MBeanServer interface shown in Figure 20 with the MBeanServerConnection interface shown in Figure 23: The MBeanServer interface extends the MBeanServerConnection interface by defining additional methods for local MBean manipulation. Since we are going to create a proxy instance for the MBeanServer interface, the MBSSurrogateMethodsMap

class will be used as a filter to just be able to invoke on the MBeanServer proxy instance the methods that are defined by the MBeanServerConnection interface and not the methods that are only defined by the MBeanServer interface.

Figure 23. MBeanServerConnection interface.

The MBSSurrogateHandler class implements the InvocationHandler interface and thus implements the invoke method. The invoke method takes as a first argument the MBeanServer proxy instance, which is of type Object. The second argument is the java.lang.reflect.Method instance corresponding to the interface method invoked on the MBeanServer proxy instance (i.e. one of the methods defined by the MBeanServerConnection interface). And the third argument is array of objects containing the values of the arguments passed in the method invocation on the MBeanServer proxy instance. The invoke checks if the method invoked on the MBeanServer proxy is one of the methods that can be invoked on the MBeanServerConnection by checking MBSSurrogateMethodsMap. If the method is found on the Map the corresponding java.lang.reflect.Method object is obtained. This object is then used to invoke the method on the supplied MBeanServerConnection with the arguments passed to the InvocationHandler's invoke. If successful, this method call on the MBeanServerConnection returns an Object object with the BAN MBeanServer response. This Object is as well the return value of method invoked on the proxy.

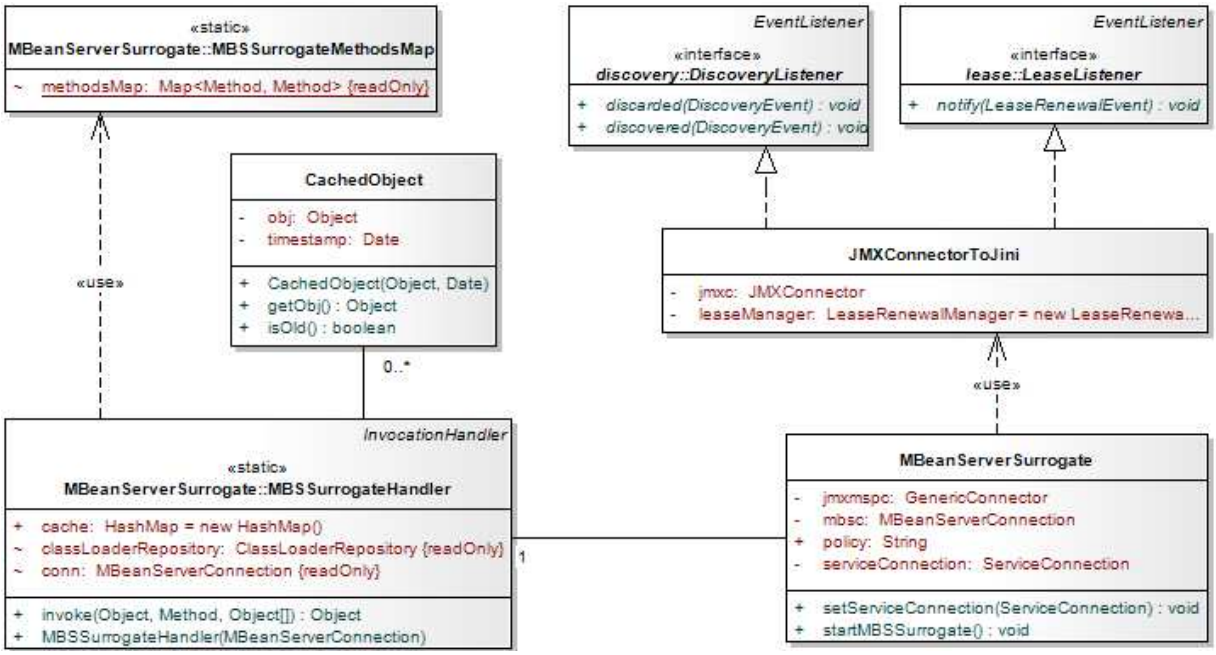

Figure 24. MBeanServerSurrogate overview.

The MBeanServerSurrogate class also implements the startMBeanServerSurrogate. This methods creates the JMXMSPConnectionSurrogate and supplies the ServiceConnection obtained from the context parameter provided by the Surrogate Host upon activating the BANManagementSurrogate. Then the SurrogateMessageRelay and the SurrogateServant are set in order to handle the communication between the BAN MBeanServer and the MBeanServerSurrogate. At this point the JMXMSPConnector can be created and its connect method called, which initiates the connection handshake as described in section 4.3.

If the connection is successfully established an MBeanServerConnection is obtained calling the getMBeanServerConnection method on the JMXMSPConnector. This connection is provided to the constructor of the MBSSurrogateHandler in order to create an invocation handler that will take care of the method invocations on the MBeanServer proxy. The MBeanServer proxy instance can now be created using the Proxy.newProxyInstance with the system class loader, the MBeanServer interface and the MBSSurrogateHandler obtained as parameters.

At this point, if everything went as expected, the BAN MBeanServer is running on the mobile device, the MBeanServerSurrogate is running on the Surrogate Host and a connection has been established between both using the JMXMSP Connector. The next step is to create the connector server that management applications will use in order to connect to the MBeanServer proxy, to remotely manage the BAN MBeanServer. Our implementation can be easily modified in order to either use the RMI connector, the JMXMP connector or any other suitable connector type by modifying the JMXServiceURL supplied to the JMXConnectorServerFactory.newJMXConnectorServer method. When the connector server for the MBeanServer proxy has been created, its start method is called in order to start listening for client connections.

5.3 Joining the Jini Network

In order for the BAN management service to fully take advantage of the Jini technology it must be able to join the Jini network and advertise itself. The BAN management service takes advantage of the Jini Surrogate Architecture to accomplish this by delegating the responsibility to register with a Lookup Servite to the BAN management surrogate.

When the MBeanServerSurrogate creates the connector server that management applications will use in order to remotely manage the BAN MBeanServer a JMXConnectorToJini object is created using the connector server as an argument in its constructor.

The JMXConnectorToJini class implements DiscoveryListener and LeaseListener interfaces. The DiscoveryListener interface defines the discovered method, called when one or more Lookup Services has been found, and the discarded method, called when they are discarded. The LeaseListener interface defines the notify method, called by the LeaseRenewalManager (instantiated by the JMXConnectorToJini) whenever it's not able to renew a lease that it is managing, and the lease's desired expiration time has not yet been reached.

The JMXConnectorToJini creates a JMXConnector with the same JMXServiceURL used by the connector server and creates a LookupDiscovery that handles the process of using the multicast discovery to find lookup services. The LookupDiscovery is a helper class that simplifies the process of finding Lookup Services using no more information than lookup service group membership, and in our implementation is used to discover all lookup services that are within range, and which belong to any group by passing the LookupDiscovery.ALL_GROUPS constant to its constructor. Once created the LookupDiscovery object, the JMXConnectorToJini sets itself as the DiscoveryListener interested in receiving DiscoveryEvent notifications whenever a Lookup Service is found. In such case, the JMXConnectorToJini discovered method is called.

This method creates an array with the ServiceRegistrar objects obtained from each Lookup Service discovered and creates a ServiceItem object (i.e. the Service Object to be registered with the Lookup Service as described in section 2.2.4) using the JMXConnector previously created with the connector server JMXServiceURL. This ServiceItem is then registered with each Lookup Service found providing a Lease.FOREVER as requested lease duration and a lease for the newly registered object is added to the LeaseRenewalManager, again with a Lease.FOREVER as desired expiration in order to try to keep the lease for as long as possible.

At this point, the Service Object for the JMXConnector has been registered with one or more Lookup Services if any was found within range. Interested management applications can then query the Lookup Service to obtain the Service Object in order to interact with the MBeanServer proxy in the Surrogate Host.

5.4 Caching

As mentioned in chapter 3 our implementation explores the possibilities of the MBeanServerSurrogate by incorporating a simple caching mechanism to leverage the management overhead on the BAN MBeanServer. This caching implementation has been included as prove of concept and would require further development to become completely usable, but shows interesting potential as a possible path to pursue further development on the BAN management architecture proposed.

The caching mechanism is implemented using a HashMap as a simple, straightforward way to cache the return values of the invoke method implemented by the MBSSurrogateHandler. A method invocation on the MBeanServer proxy instance that has not been previously invoked yields to an invocation of the same method on the remote BAN MBeanServer. The object returned by this method invocation is used to create a CachedObject. This CachedObject contains the object returned by the method invocation together with the timestamp when the CachedObject is created. Furthermore it provides an isOld method that returns true if the CachedObject is older than a certain period specified in milliseconds.

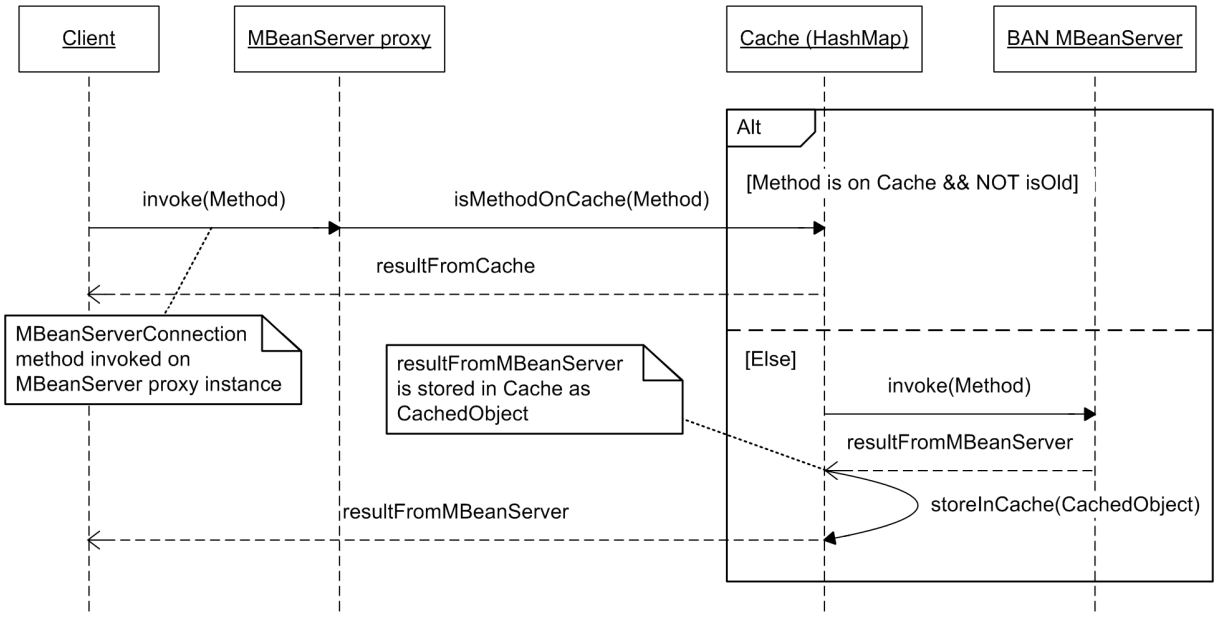

Figure 25. MBeanServerSurrogate caching mechanism.

When a method is invoked on the MBeanServer proxy, the invoke method checks the HashMap to see if that method has been previously invoked and the return value stored. If the method has been previously invoked and it is not an old value, the object is obtained from the cache and immediately returned. Otherwise, either the value stored in the cache is old (with respect to the expiration period specified by the CachedObject) or the method has never been invoked before. In both cases the method is called on the MBeanServerConnection and a fresh return value obtained from the BAN MBeanServer. A sequence diagram showing this process is shown in Figure 25.

Figure 23 shows all the methods that can be invoked on an MBeanServerConnection. An invocation to any of these methods is subject to be stored on the MBeanServerSurrogate cache. The current design of the caching mechanism does not make any distinction on the methods to be stored on the cache. Every method invoked on the MBeanServer proxy is stored on the cache with the same expiration period. This could be improved by differentiating the methods that are useful to be cached from those that are not and assigning different expiration periods. For instance, it may not be useful to store a createMBean method invocation in the cache, since this method returns an ObjectInstance onject, containing the ObjectName and the Java class name of the newly instantiated MBean. Storing a createMBean method invocation may return the wrong ObjectInstance to subsequent invocations of such method.

On the other hand, storing a getAttribute method invocation may be very convenient, especially if the expiration period assign to the CachedObject is carefully chosen. The getAttribute method returns the value of an attribute of an MBean. If this attribute does not change frequently over time it may be useful to store the getAttribute invocation requesting this attribute with a longer expiration period, keeping unnecessary method invocations from being forwarded to request the value directly from the BAN MBeanServer.

Section 6.2.3 presents some tests carried out in order to demonstrate the current design of this caching mechanism, showing the performance gain that can be obtained from using such mechanism and its shortcomings, and describes two possible improvements over the current design.

5.5 Client – BAN Management Surrogate Interaction

When a management application wants to remotely manage the BAN it must do so by means of the MBeanServerSurrogate, thus it needs to connect to the connector server provided by the MBeanServerSurrogate. If the client is aware of the location where this connector is listening for connections it can directly create a JMXConnector with a JMXServiceURL using the well known URL of the connection server, and then connect. But if the client doesn't know the JMXConnector location it needs first to make use of a Lookup Service in order to obtain a Service Object that will allow it to connect to the connector server.

In our implementation the JMXConnectorToJini directly exports a JMXConnector that is already configured with the JMXServiceURL of the connector server. All what is left for the client to do is to use a LookupDiscovery object in order to find any available Lookup Services within range, in the same way the JMXConnectorToJini does, and query every Lookup Service found using a ServiceTemplate specifying that is looking for a JMXConnector.

The client obtains a JMXConnector from every Lookup Service found that had such Service Object. If the JMXConnectorToJini specifies more attributes when creating the Service Object, for instance the specific device name where the BAN MBeanServer is running, if these details are known to the client as well, the client can use them to build a more specific query and obtain a filtered set of results. With the JMXConnector obtained from the Lookup Service, the client can then connect to the connector server in a completely transparent way calling the connect method on the JMXConnector.

An alternative for the JMXConnectorToJini to export the service would be to create the Service Object using the JMXServiceURL instead of the JMXConnector and register it with the Lookup Service. In such situation the client must first create a JMXConnector and use the obtained JMXServiceURL obtained to connect to the connector server.

Either way, the client can now obtain an MBeanServerConnection using the getMBeanServerConnection method from the JMXConnector and communicate with the MBeanServer proxy running on the Surrogate Host.

6 Evaluation

In the previous chapters we have first presented the background information necessary to understand the subject of this research, then given an overview of the proposed BAN management architecture and finally presented the JMXMSP Connector and BAN Management Service Surrogate that make up the core of this architecture. In this chapter we present an evaluation of the proposed architecture and the prototype developed and give some indications on its performance.

6.1 Experimental Setup

The setup used in our architecture relies on the MobiHealth Service Platform which provides an implementation of the Mobile Service Platform and includes a Surrogate Host implementation based on the Madison Surrogate Host contributed by Sun Microsystems that runs on a desktop computer.

Furthermore, this desktop computer runs a simple HTTP server to provide an accessible location for the Surrogate Host to retrieve the BAN management surrogate JAR as well as any other files if and when they are needed. This HTTP server also provides a location to store any MBeans that can be dynamically retrieved, instantiated and registered by the BAN MBeanServer.

A Lookup Service runs on this desktop computer as well. The Lookup Service used is called Reggie and is an implementation provided by Sun and supplied as part of the standard Jini distribution. Note that Surrogate Host, HTTP server and Lookup Service could have been running in a distributed environment using different machines. Figure 26 shows the setup described.

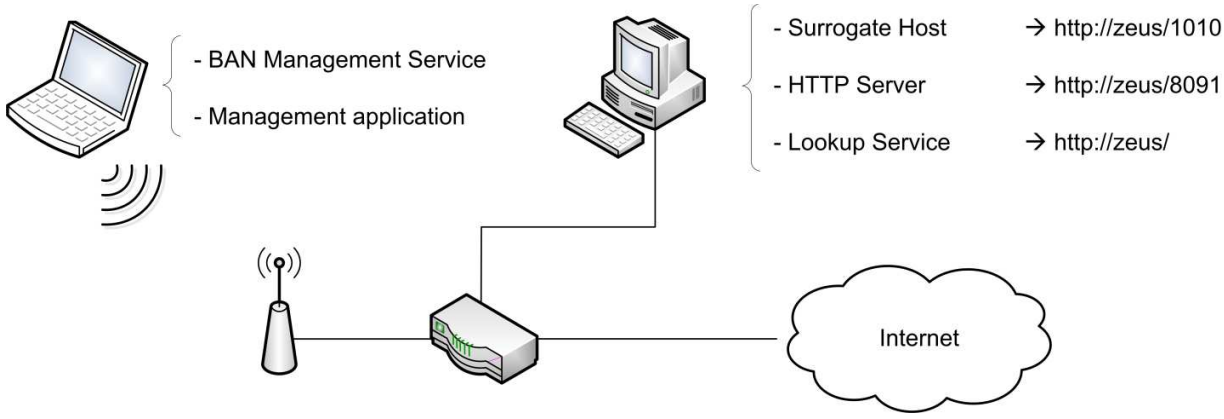

Figure 26. Experimental setup.

A second computer (a laptop connected via WiFi to the Internet) is used to run the BAN management service.

6.2 Prototype Evaluation

In this section we provide an evaluation of the BAN management architecture prototype describing different scenarios that demonstrate the functionality provided by the prototype. For each demonstration its purpose is first described and then its results are presented and discussed, providing some insight on its performance.

In order to ease the testing process, every demo has been carried out using a simple test client that performs the different steps in a sequential order and presents the results directly on the command line, therefore all the results are text based.

6.2.1 Retrieving information from the MBeanServer

The purpose of this demonstration is to show the type of information that can be retrieved from the MBeanServer and how this can be accomplished.

In our design, once the test client (i.e. our management application) finds a Lookup Service and obtains a JMXConnector, the client can obtain an MBeanServerConnection. This interface defines a fair amount of methods that the management application can use in order to interact with the MBeanServer proxy in the Surrogate Host. The proxy is in charge of forwarding the method calls on the MBeanServerConnection to the BAN MBeanServer, but this is completely transparent to the management application. Listing 2 shows the code fragment a management application uses to obtain an MBeanServerConnection and send a request to the BAN MBeanServer to obtain the number of MBeans registered.

Listing 2. Code fragment to obtain an MBeanServerConnection and get the number of registered MBeans on the BAN MBeanServer.

```
// Once a Lookup Service has been discovered, it is queried using a template
// in order to obtain a JMXConnector.
jmxc = (JMXConnector) registrar.lookup(template);
// JMXConnector retrieved from LUS. Now connect.
jmxc.connect();
// Obtain an MBeanServerConnection
MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
int mbeanCount = mbsc.getMBeanCount();
```

The code fragment in Listing 2 shows that for a management application to access the BAN MBeanServer through its proxy on the Surrogate Host the process is absolutely transparent. The same code could be used to access directly the BAN MBeanServer by using a different JMXConnector with the appropriate JMXServiceURL rather than using that obtained from the Lookup Service, which is primed with the URL of the proxy's connector server.

The UML diagram shown in Figure 23 presents the different methods defined by MBeanServerConnection interface. In order to retrieve information from the BAN MBeanServer in this

demonstration we use the getMBeanCount method to retrieve the number of MBeans that are already registered, getDomains to obtain the existing domains, queryNames to get the names of MBeans controlled by the MBean server and getMBeanInfo to retrieve information about the different MBeans.

Listing 3 shows the result of running this demo. First the Lookup Service running on the desktop computer is located using a LookupDiscovery object (see sections 5.3 and 5.5). Once it is found, we query the Lookup Service for a JMXConnector object that is used to obtain an MBeanServerConnection. This MBeanServerConnection is then used to invoke different method to retrieve MBeanServer information.

Listing 3. Retrieving information from the MBeanServer

```
Lookup service found...
JMXConnector retrieved from LUS
Now connecting to the JMXConnectorServer...
Connected!
Get an MBeanServerConnection
MBeanServerConnection obtained

Now perform Information retrieval demo tests...

Display BAN MBeanServer Domains:
      Domain[0] = JMImplementation
      Domain[1] = DefaultDomain

2 MBeans registered with the BAN MBeanServer:
      MBean: MLet (from domain: DefaultDomain)
      MBean: MBeanServerDelegate (from domain: JMImplementation)

Now for each registered MBean, display information:

MLet MBean information retrieved:
      Attributes exposed for management by the MLet MBean:
            Attr name: LibraryDirectory
            Attr type: java.lang.String
            Attr description: Attribute exposed for management
            Attr name: URLs
            Attr type: [Ljava.net.URL;
            Attr description: Attribute exposed for management
      Operations exposed for management by the MLet MBean:
            Operation name: getMBeansFromURL
            Operation description: Operation exposed for management
            ...
      Constructors of the MLet MBean:
            Constructor name: javax.management.loading.MLet
            Constructor description: Public constructor of the MBean
            ...
      Notifications emitted by the MLet MBean:
            (no notifications emitted)

MBeanServerDelegate MBean information retrieved:
      Attributes exposed for management by the MBeanServerDelegate MBean:
            Attr name: LibraryDirectory
            Attr type: java.lang.String
```

```
            Attr description: Attribute exposed for management
            Attr name: URLs
            Attr type: [Ljava.net.URL;
            Attr description: Attribute exposed for management
      Operations exposed for management by the MBeanServerDelegate MBean:
            Operation name: getMBeansFromURL
            Operation description: Operation exposed for management
            ...
      Constructors of the MBeanServerDelegate MBean:
            Constructor name: javax.management.loading.MLet
            Constructor description: Public constructor of the MBean
            ...
      Notifications emmited by the MBeanServerDelegate MBean:
            (no notifications emitted)

All tests done.
```

Two things can be noticed from these results. The first is that the only two MBeans registered with the BAN MBeanServer at the moment of running this test are the MLet MBean and MBeanServerDelegate MBean. The former is used in our implementation in order to be able to remotely create MBeans and is further described in section 6.2.2, while the latter represents the MBean server from a management perspective and is in charge of emitting the MBeanServerNotifications when an MBean is registered/unregistered with the MBean server.

The second interesting thing that we can notice is that when retrieving information from the different attributes, operations, constructors and notifications from Standard MBeans their descriptions are generic. The reason for this is that the contents of the MBeanInfo returned by the getMBeanInfo method for a Standard MBean are determined using reflection, and parameter names are not available through the reflection API. For further information on the subject and a way to overcome this problem by adding annotations to the Standard MBeans we refer the reader to [22].

In order to give an indication on the performance of this test the different requests have been timed in order to have an estimation of how long does it take from the moment the request is executed until the results have been returned by the BAN MBeanServer via the MBeanServer proxy. This is accomplished using a simple Stopwatch class found in [23]. This class returns the difference in milliseconds between two calls to the System.currentTimeMillis() method. As it is stated in the Java API, the granularity of the value returned by a System.currentTimeMillis() call depends on the underlying operating system and may be larger than a millisecond, so in order to obtain more precise performance measurements other methods may be used. The following table summarizes the results of this first test:

Table 2. Information retrieval performance test results using the JMXMSP connector.

Query	Average RMI execution time (in milliseconds)
getMBeanCount	500
getDomains	515
queryNames (retrieving the name of all registered MBeans, and with no query expression supplied)	594

getMBeanInfo (info retrieved for the MLet MBean)	781
getMBeanInfo (info retrieved for the MBeanServerDelegate MBean)	578

The results presented in Table 2 show the average remote method invocation execution time over 5 executions of the test. The average is used to give a more insightful result considering the variations on the resulting times for the same type of queries over different executions.

As we can see in the results presented in Table 2, the average execution time for each remote query is of the order of a few hundred milliseconds. In [24] some performance measurements are carried out on MSP, one of the results states that the Round-Trip Time (RTT) of the keep-alive messages measured at the surrogate is on average of a few hundred milliseconds, with very similar results to those obtained on our test. In our experimental setup these keep-alive RTT's are logged as well by the ClientSAPServant and show the same average times. This means that having the remote method invocation execution times on the order of hundreds of milliseconds it is actually what can be expected when using MSP and we can conclude that there is no significant overhead added by our management architecture in terms of the time it takes to perform a remote management operation on an MBean registered with the BAN MBeanServer.

In order to compare the performance results presented in Table 2, the same tests described have been carried out using a modified version of the same test client where the connection is established directly with the BAN MBeanServer, and thus without using the MBeanServerSurrogate. In this modified version of the test, the BAN MBeanServer creates a JMXMPConnectorServer instead of our JMXMSPConnectorServer. The modified test client creates a JMXConnector that connects directly to the JMXMPConnectorServer and then performs the same tests to retrieve information from the MBeans registered with the BAN MBeanServer. Again, the results presented show the average time for each query after executing the test 5 times.

Table 3. Information retrieval performance test results using the JMXMP connector and connecting directly to the BAN MBeanServer.

Query	Average RMI execution time (in milliseconds)
getMBeanCount	31
getDomains	15
queryNames (retrieving the name of all registered MBeans, and with no query expression supplied)	16
getMBeanInfo (info retrieved for the MLet MBean)	47
getMBeanInfo (info retrieved for the MBeanServerDelegate MBean)	16

As we can see in the results presented in Table 3, the average remote method invocation execution time for every tested query is significantly smaller than when performing the query via the MBeanServer surrogate.

6.2.2 Remote MBean Creation and Invocation of Exposed Methods

This test is designed to show how MBeans can be created, registered and managed from a remote management application. In order to create and register an MBean remotely, the management application can invoke the createMBean method remotely on an MBeanServerConnection. This method instantiates and registers an MBean in the MBean server (in our design this is the remote BAN MBeanServer). The ClassLoader to be used can be specified as one of the parameters of the method invocation. If no ClassLoader is provided, the MBeanServer will use its Default Loader Repository to load the class of the MBean.

In our design, however, we take a different approach in order to create and register MBeans with the BAN MBeanServer using the m-let service. The m-let service is an instance of the MLet class in the javax.management.loading package, which extends the URLClassLoader of the java.net package and which is also an MBean [25]. The m-let service provides the functionality to instantiate and register MBeans obtained from a specified URL. This URL points to the location of an m-let text file which provides information on the MBeans to be obtained. The information on each MBean is enclosed on an MLET tag as it can be seen in the m-let text file in Listing 4.

Listing 4. M-let text file specifying information on the two test MBeans used for this demonstration.

```
<MLET
CODE = jmxmsp.testmbeans.StateControl
ARCHIVE = "mbeans.jar"
CODEBASE = http://zeus:8091/mbeans
NAME = :type=StateControl
>
</MLET>

<MLET
CODE = jmxmsp.testmbeans.Calculations
ARCHIVE = "mbeans.jar"
CODEBASE = http://zeus:8091/mbeans
NAME = :type=Calculations
>
</MLET>
```

Each MLET tag contains a set of attributes which are briefly described in the following table:

Table 4. Description of the different m-let text file attributes.

ATTRIBUTE NAME = *attribute value*	**Attribute Description**
CODE = *class*	Class name (including package) of the MBean to be obtained. Either CODE or OBJECT must be specified.
OBJECT = *serfile*	.ser file containing a serialized representation of the MBean to be obtained.
ARCHIVE = *"archiveList"*	Specifies one or more .jar files containing MBeans or other resources used by the MBean to be obtained. Must contain the files specified by the CODE/OBJECT attributes.

CODEBASE = *codebaseURL*	Identifies the location of the .jar files specified by the ARCHIVE attribute. If not present, the base URL of the m-let text file is used. (optional)
NAME = *mbeanname*	Name to be assigned to the MBean instance when the m-let service registers it. (optional)
VERSION = *version*	Version number of the MBean and associated .jar files to be obtained. (optional)
arglist	Specifies one or more parameters to be passed to the MBean's constructor.

In our design we create and register an MLet MBean with the BAN MBeanServer. Since the MLet class extends URLClassLoader, when it is registered with the MBeanServer it is added to its Default Loader Repository. This allows us to instantiate and register MBeans that are retrieved from the remote locations specified by an m-let file, providing great flexibility while deploying the different MBeans that we want to be able to manage remotely. In our design, these MBeans are packaged in a JAR file and placed in an accessible location on the same HTTP server used to deploy the surrogate JAR. In the same directory where we place the *mbeans.jar* we place the *mlet.txt* file with the details of the different MBeans contained in the JAR file. Note that different JAR files containing different sets of MBeans could be created and placed on the same, or even on different HTTP servers. To retrieve them, one or more m-let text files could be used.

The presented setup, allows us to easily update or add new MBeans to the set of MBeans that we need to be able to create and register with the BAN MBeanServer: they can simply be packaged on the *mbeans.jar* and uploaded to the HTTP server. Listing 5 shows how the MLet is created and registered in our implementation.

Listing 5. Code fragment to create and register an MLet with the BAN MBeanServer

```
// The BANMBeanService creates the BAN MBeanServer
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
// Create the MLet ObjectName
ObjectName mletName = new ObjectName(mbs.getDefaultDomain()+":type=MLet");
// Get an MLet instance and add the URL created
MLet mlet = new MLet();
// Register mlet with the BAN MBeanServer
mbs.registerMBean(mlet, mletName);
```

For this demonstration we have placed an *mbeans.jar* file containing the classes of two simple MBeans: the Calculations MBean which exposes some methods to perform simple calculations and the StateControl MBean which exposes methods for getting and setting the value of a String attribute and a reset method that resets the attribute to its original value and emits a notification indicating that the value has been reset.

The test client first finds a Lookup Service and obtains a JMXConnector that is used to get an MBeanServerConnection in order to interact with the MBeanServer proxy in the Surrogate Host, which

in turn communicates with the BAN MBeanServer. Once the connection has been obtained, the client retrieves the number of MBeans that are already registered with the BAN MBeanServer and invokes the queryNames method to get their names. Then the client creates and registers a Calculations MBean and a StateControl MBean with the BAN MBeanServer. In our design this is done using the functionality provided by the m-let service. First the ObjectName used to identify the m-let service already registered with the BAN MBeanServer is created. A URL object with the location of the *mlet.txt* file specifying the StateControl and Calculations MBeans is created. Both of these objects are used in order to invoke the getMBeansFromURL method on the MLet MBean. This method returns either the object instance of the MBean that has been downloaded and registered or a Throwable object if there was an error or an exception. These are returned in a Set object with as many elements as there are MLET tags in the target m-let text file.

This means that once the getMBeansFromURL call returns, if there were no errors, the Calculations and StateControl MBeans have been obtained from the HTTP server and registered with the BAN MBeanServer. Listing 6 shows the code used in our design to remotely create and register the MBeans with the BAN MBeanServer.

Listing 6. Code used to remotely create and register MBeans using the m-let service.

```
// Create the ObjectName that identifies the m-let service already registered
// with the BAN MBeanServer
ObjectName mlet = new ObjectName("DefaultDomain:type=MLet");
// Create the URL with the location of the mlet.txt file that specifies the
// Calculations and StateControl MBeans.
URL mbeansURL = new URL("http://zeus:8091/mbeans/mlet.txt");
Object params[] = {mbeansURL};
String signature[] = {"java.net.URL"};
// Invoke the getMBeansFromURL method provided by the MLet MBean with the
// supplied URL
Set registeredMBeans = (Set) mbsc.invoke(mlet, "getMBeansFromURL", params,
signature);
```

Once both MBeans have been registered, the test client proceeds to perform some management operations on both MBeans. It first tries to get the value of the StateControl attribute exposed for management by the StateControl MBean. This attribute is a String that by default contains the value "*Default state*". In order to retrieve the value of this attribute the getAttribute method is invoked on the MBeanServerConnection. Then the client uses setAttribute in order to change the value of the String and retrieves its value again in order to show that the value indeed changed. The code fragments used to perform these two management operations are shown in Listing 7:

Listing 7. Getting and setting attributes exposed for management by an MBean.

```
// Create ObjectName to identify the MBean where to get and set the attribute
ObjectName mbeanName = new ObjectName("DefaultDomain:type=StateControl");
// Get the StateControl attribute
String StateControl = (String) mbsc.getAttribute(mbeanName, "StateControl");
// Create an Attribute object with the new value to be set
Attribute attribute = new Attribute("StateControl", "New state!!!");
```

```
// Set the new attribute value
mbsc.setAttribute(mbeanName, attribute);
```

Then the test client instantiates a ClientListener and adds it to the StateControl MBean invoking the addNotificationListener on the MBeanServerConnection. The ClientListener is a simple class that implements the javax.management.NotificationListener interface in order to handle the notifications received from MBean. When a notification is emitted by the StateControl MBean, the ClientListener will handle the notification by printing information related to the notification, such as its source, the message included in the notification if any and the notification type.

Now the client is ready to receive notifications from the StateControl MBean. This MBean exposes a reset method for management. When invoked, this method resets the value of the StateControl attribute to its initial value (Default state) and creates an AttributeChangeNotification object that sends to any registered listeners.

The test client invokes the reset method by calling the invoke method on the MBeanServerConnection and specifying the appropriate ObjectName for the StateControl MBean and the method to be invoked (i.e. reset) as a String parameter. Listing 8 shows the code fragment used to accomplish this.

Listing 8. Remotely invoking a method on the StateControl MBean.

```
mbsc.invoke(new ObjectName("DefaultDomain:type=StateControl"), "reset", null,
null);
```

As it can be seen on the outcome of the test client shown in Listing 10, the test client then waits to receive a notification. Once the reset method has been invocated the StateControl MBean sends the notification which is then handled by the ClientListener by showing its source, message and type.

An additional method invocation is then carried out on the Calculations MBean in order to show how parameters can be supplied for these invocations. The test client uses an add method exposed for management by the Calculations MBean in order to add two supplied integers.

Listing 9. Providing parameters to a method invocation on the Calculations MBean.

```
// Create ObjectName for the Calculations MBean
ObjectName mbeanName = new ObjectName("DefaultDomain:type=Calculations");
// Create parameters and signature
Object params[] = { new Integer(42), new Integer(42) };
String signature[] = {"int", "int"};
// invoke add method and show results
System.out.println("\t 42 + 42 = "+ mbsc.invoke(mbeanName, "add", params,
signature));
```

Listing 9 shows how in order to invoke a method on a remote MBean that accepts parameters for the method call, these must be provided inside an array of Object objects, and the method signature specified with an array of Strings with each parameter type.

The console output of the tests described can be seen in Listing 10.

Listing 10. Remote creation and registration of MBeans, method invocations and notification receptions.

```
Lookup service found...
JMXConnector retrieved from LUS
Now connecting to the JMXConnectorServer...
Connected!
Get an MBeanServerConnection
MBeanServerConnection obtained

Now perform remote MBean creation and operations demo tests...

2 MBeans were already registered with the BAN MBeanServer:
      MBean: MLet (from domain: DefaultDomain)
      MBean: MBeanServerDelegate (from domain: JMImplementation)

Invoke getMBeansFromURL with the MLet txt file location
      to create and register the demo MBeans with the BAN MBeanServer...

The following MBeans have been registered with the BAN MBeanServer:
      MBean: StateControl (from domain: DefaultDomain)
      MBean: Calculations (from domain: DefaultDomain)

Get attribute StateControl: Default state
Changing attribute...
Get attribute StateControl again: New state!!!

Add notification listener to get notifications from StateControl...

Now we invoke the reset method on StateControl, which resets
      the value of the StateControl attribute and emits a
      notification to registered listeners...

Waiting for notification......
Received notification:
      Source: DefaultDomain:type=StateControl
      Message: StateControl has been reset to Default state
      Type: jmx.attribute.change

Get attribute StateControl to verified that it changed: Default state

Now perform a simple addition using the CalculationsMBean add method:
       42 + 42 = 84

Unregister the StateControl and Calculations MBeans...

2 MBeans registered with the BAN MBeanServer:
      MBean: MLet (from domain: DefaultDomain)
      MBean: MBeanServerDelegate (from domain: JMImplementation)

All tests done.
```

Like on the previous test, some performance benchmarks have been carried out in order to provide an estimation of how long does it take for some of these remote method invocations from the moment

they are executed until the results have been returned by the BAN MBeanServer via the MBeanServer proxy. The same Stopwatch class used in the previous test has been used. Again, the results presented show the average time for each query after executing the test 5 times. The following table summarizes the results:

Table 5. Remote MBean creation and method invocation performance test results.

Query	Average RMI execution time (in milliseconds)
getMBeansFromURL	10266
getAttribute (on StateControl MBean)	804
setAttribute (on StateControl MBean)	703
addNotificationListener (for StateControl)	5063
reset (on StateControl MBean)	500
add (on StateControl MBean)	609

From the results presented in Table 5 we can conclude that getMBeansFromURL and addNotificationListener are the most time consuming operations. In the case of getMBeansFromURL this result is clearly expected since its invocation means that the BAN MBeanServer must first obtain the m-let text file from the HTTP server (shown in Listing 4), then read the different MLET tags, obtain the *mbeans.jar* from the HTTP server, unpack the contents and register an instance of each MBean before sending the invocation results back to the management application. This, together with the fact that both the original method invocation and its response go through the MBeanServer proxy on the Surrogate Host, results in a longer RMI execution times.

Once again, in order to compare the performance results obtained in the tests described, the same tests have been carried out using a modified version of the test client where the connection is established directly with the BAN MBeanServer (i.e. without using the MBeanServerSurrogate). In this modified version of the test, the BAN MBeanServer creates a JMXMPConnectorServer instead of our JMXMSPConnectorServer, and the modified test client uses a JMXConnector to connect directly to the JMXMPConnectorServer and perform the same tests creating a new Calculations and a new StateControl MBean and performing the different management operations previously described. The results of this alternative test are presented in Table 6.

Table 6. Remote MBean creation and method invocation performance test using the JMXMP connector and connecting directly to the BAN MBeanServer.

Query	Average RMI execution time (in milliseconds)
getMBeansFromURL	4532
getAttribute (on StateControl MBean)	16
setAttribute (on StateControl MBean)	15
addNotificationListener (for StateControl)	32
reset (on StateControl MBean)	125
add (on StateControl MBean)	15

Again, connecting directly to the BAN MBeanServer results in far smaller RMI execution times for every query compared to the amount of time needed for every query when using the MBeanServerSurrogate.

6.2.3 Caching demonstration

The following test is designed to show how the caching mechanism included in our design works and the performance improvements that can be obtained with it enabled, as well as some important drawbacks that arise from using such caching mechanism. The previous tests have been carried out with this caching mechanism disabled to provide the most meaningful performance results when executing remote management operations from a management application over an MBean registered with the BAN MBeanServer.

Like the previous test, the test client obtains a JMXConnector from the Lookup Service and uses it to get an MBeanServerConnection to interact with the BAN MBeanServer via the MBeanServer proxy in the Surrogate Host. Then it uses the getMBeanFromURL in order to register the StateControl and Calculations MBeans with the BAN MBeanServer.

In order to demonstrate how the simple caching mechanism works, the test client invokes the getAttribute method to obtain the StateControl attribute from the StateControl MBean then it changes the value using the setAttribute and retrieves it again with a new getAttribute invocation. The expiration period set for each CachedObject is 10000 milliseconds. This means that whenever a method invocation occurs at the MBeanServerSurrogate the cache is checked to see if the method has been recently invoked. If such method was invoked less than 10000 milliseconds before, the result of such method invocation is returned from the value stored on the cache. Otherwise the invocation is carried out normally to retrieve the result from the BAN MBeanServer. As it can be seen in Listing 11, this sequence of operations yields interesting results. The first time the getAttribute method is invoked the attribute is retrieved in 392 milliseconds with the value “Default state” corresponding to the default value. The value is then changed to “New state!!!”.

When the value is retrieved again with a new getAttribute invocation, the return value is obtained in 15 milliseconds from the cache because it occurred before the cache entry expired (i.e. less than 10000 milliseconds after the value was stored) returning the same “Default state” String value that was stored in the cache. The test client Thread is then put to sleep for 10000 milliseconds in order to make sure that the CachedObject for the getAttribute method invocation becomes old, and getAttribute is invoked again. This time the value obtained is coming from the BAN MBeanServer since the cached value was considered old and we can see the “New state!!!” value that corresponds to the actual value of the StateControl attribute.

A similar test is then carried out using the Calculations MBean, where its add method is used. First using the invoke method of the MBeanServerConnection the add method is remotely invoked in order to obtain the result of adding two times 42. The result obtained is, as we would expect, 84 and it took 516 milliseconds to obtain this value. But invoking the same method again to obtain the result of adding two times 24 the returned result is again 84 instead of the expected 48, taking 15 milliseconds to obtain it.

This is again due to the fact that the return value for the invoke method was retrieved from the value stored in the cache. If the test client Thread is again put to sleep for 10000 milliseconds and we invoke again the add method on the Calculations MBean to obtain the result of 24 + 24, we obtain the expected 48.

Listing 11. Results of the caching tests.

```
Lookup service found...
JMXConnector retrieved from LUS
Now connecting to the JMXConnectorServer...
Connected!
Get an MBeanServerConnection
MBeanServerConnection obtained
Now perform remote MBean caching demo tests...

Invoke getMBeansFromURL with the MLet txt file location
      to create and register the demo MBeans with the BAN MBeanServer...

The following MBeans have been registered with the BAN MBeanServer:
      MBean: StateControl (from domain: DefaultDomain)
      MBean: Calculations (from domain: DefaultDomain)

Get attribute StateControl: Default state
      getAttribute invocation took: 391 milliseconds
Changing attribute...
Get attribute StateControl again: Default state
      getAttribute value returned from cache took: 15 milliseconds

If we wait for the cached getAttribute to expire (10 sec), getAttribute
returns the new value...

Get attribute StateControl to verify that it changed: New state!!!

Now perform a simple addition using the CalculationsMBean add method:
      42 + 42 = 84
      invoking add took: 516 milliseconds
Now invoke add with different arguments:
      24 + 24 = 84!
      add result returned from cache took: 15 milliseconds
Again, if we wait for the cached add method to expire, add returns the new
result...
       24 + 24 = 48

Unregister the StateControl and Calculations MBeans...

2 MBeans registered with the BAN MBeanServer:
      MBean: MLet (from domain: DefaultDomain)
      MBean: MBeanServerDelegate (from domain: JMImplementation)

All tests done.
```

The performance improvements that result from incorporating a caching mechanism to the MBeanServerSurrogate are obvious. For the getAttribute method invocation obtaining the result from

the BAN MBeanServer took 392 milliseconds as opposed to the 15 milliseconds it took to obtain the result from the cache on the MBeanServerSurrogate. With the add method invocation the result obtained from the BAN MBeanServer took 516 milliseconds, whereas the result obtained from the cache took 15 milliseconds.

This performance improvements show that incorporating a caching mechanism to the MBeanServerSurrogate is an option worth exploring. Even more if we consider that such mechanism allows us to leverage the management overhead over the BAN MBeanServer by reducing the amount of operations that are forwarded by the MBeanServerSurrogate.

However a clear drawback arises from implementing this caching mechanism in the MBeanServerSurrogate: dealing with old return values. The current implementation is no more than a proof of concept of such mechanism, but it becomes clear that it must be further developed in order to become usable. Here we propose two improvements that could be implemented in the current caching mechanism:

- The expiration period of each CachedObject could be set on a per method/per cached object basis. This would mean the result values of different method invocations would be considered old differently or that even the different CachedObject objects would be considered old differently. This would allow to set different priorities to the CachedObject's or even admit a *caching=false* parameter in order to indicate that a certain object should not be cached.
- Cache the different method invocations storing the parameters they were invoked with. The current implementation just checks the cache for methods that have been already invoked. As seen in Listing 11, this leads to faulty return values when invoking the same method with different parameters. To solve this, the CachedObject objects could include as attributes the method signature and the parameter values the method was invoked with, and have the caching mechanism check for both besides checking for the method name, and store the value if and only if the method invocation was performed with the same parameters.

Note that, queries performed on the MBeanServerSurrogate that are returned from the cache (for both the getAttribute and setAttribute methods the RMI execution time is 15 milliseconds) result in RMI execution times similar to the time for the same queries when performed directly on the BAN MBeanServer (see Table 6). This shows that using a caching mechanism within the MBeanServerSurrogate can help to greatly reduce the significant performance overhead introduced when using MSP in our architecture as discussed in sections 6.2.1 and 6.2.2.

6.2.4 Using JConsole to Manage the BAN

The previous tests have been carried out using simple management clients that perform a set of sequential management operations on a couple of simple MBeans registered with the BAN MBeanServer. The following demonstration aims at showing how the Java Monitoring and Management Console (JConsole) tool can be used as well to manage those MBeans. JConsole provides a graphical user interface that makes management operations easier to perform, however, we need to connect JConsole

directly to the address of the connector server of the MBeanServer proxy running on the Surrogate Host, as opposed to the previous test clients where we make use of a Lookup Service to obtain a JMXConnector that can be used to connect the MBeanServer proxy. This means that we need to provide the connector server address upon connecting, as it can be seen in Figure 27.

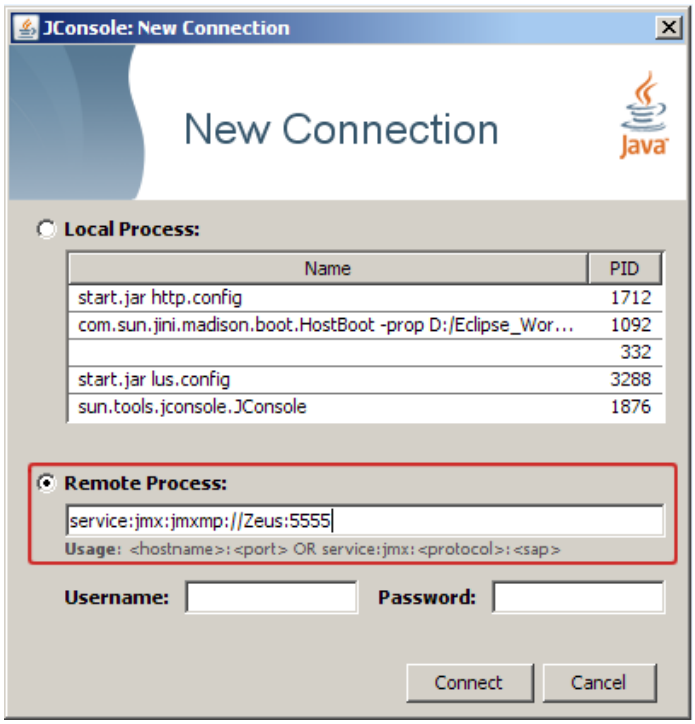

Figure 27. Connectin JConsole to the MBeanServer proxy.

Once the connection has been established, we can see that the only two MBeans that are registered with the BAN MBeanServer are the MLet and the MBeanServerDelegate MBeans (see Figure 28). We can then invoke getMBeansFromURL supplying as parameter the URL of the m-let text file describing the StateControl and Calculations MBean in order to register an instance of each with the BAN MBeanServer.

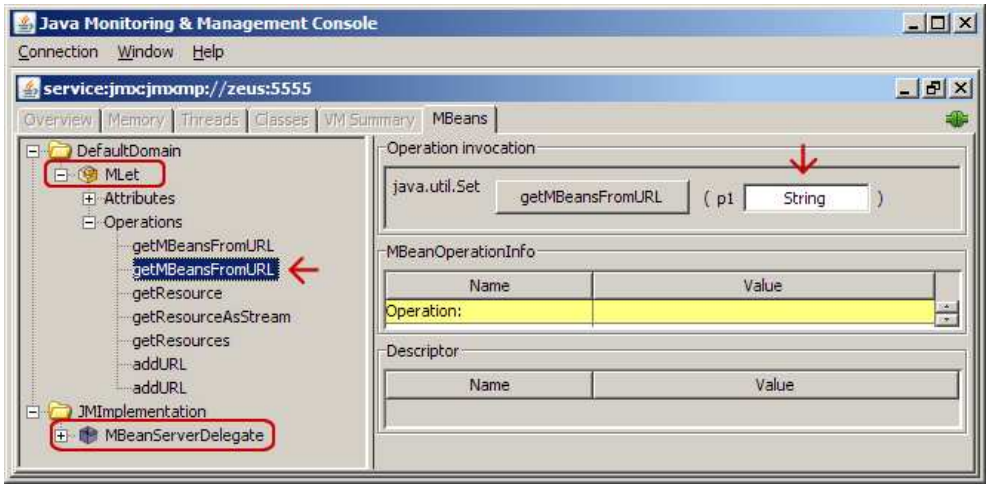

Figure 28. Initially, just the MLet and MBeanServerDelegate MBeans are registered with BAN MBeanServer.

If the getMBeansFromURL invocation succeeds, the StateControl and Calculations MBeans are registered with the BAN MBeanServer and we are able to perform management operations on them. Figure 29 shows both MBeans successfully registered and the value of the StateControl String attribute from the StateControl MBean modified from its default value ("Default state") to "State changed from JConsole".

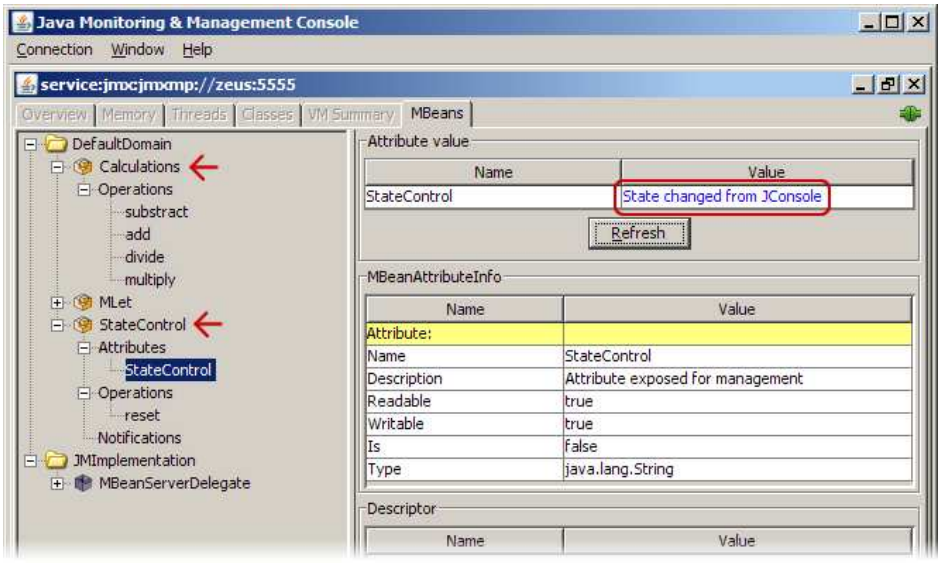

Figure 29. The StateControl and Calculations MBeans are registered with the BAN MBeanServer and we modify the StateControl attribute.

As we have seen in section 6.2.2, the StateControl MBean emits an AttributeChangeNotification when its reset method is invoked. In order to receive notifications emitted by the StateControl MBean we first have to subscribe to the notifications we want to receive. This is done from the Notifications section of the StateControl MBean. There we can select the AttributeChangeNotification and subscribe to it. Now if the reset method is invoked we receive the notification as shown in Figure 30:

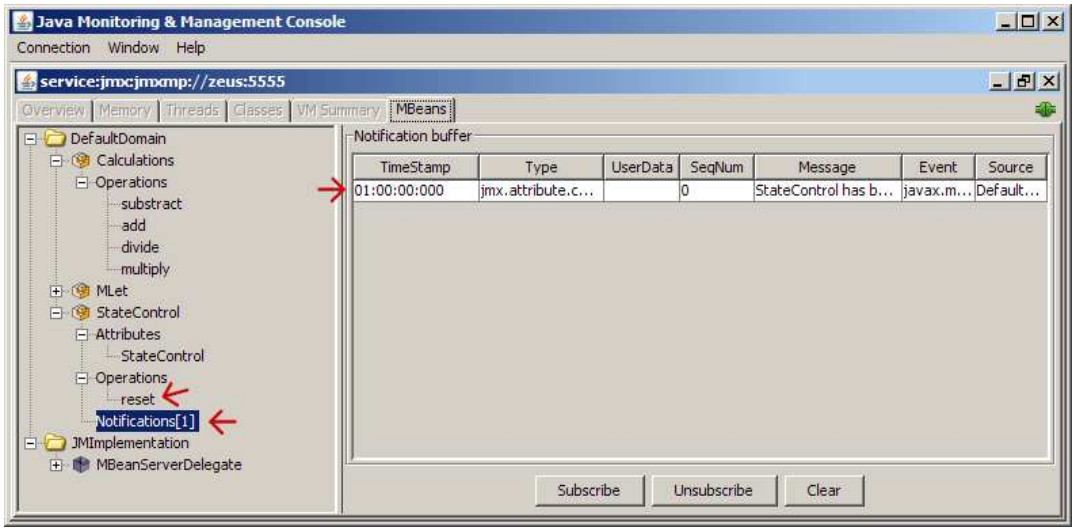

Figure 30. When the StateControl MBean reset method is invoked, we receive a notification.

As a last test we can invoke the add method exposed by the Calculations MBean to obtain the result of 42 + 42 (see Figure 31).

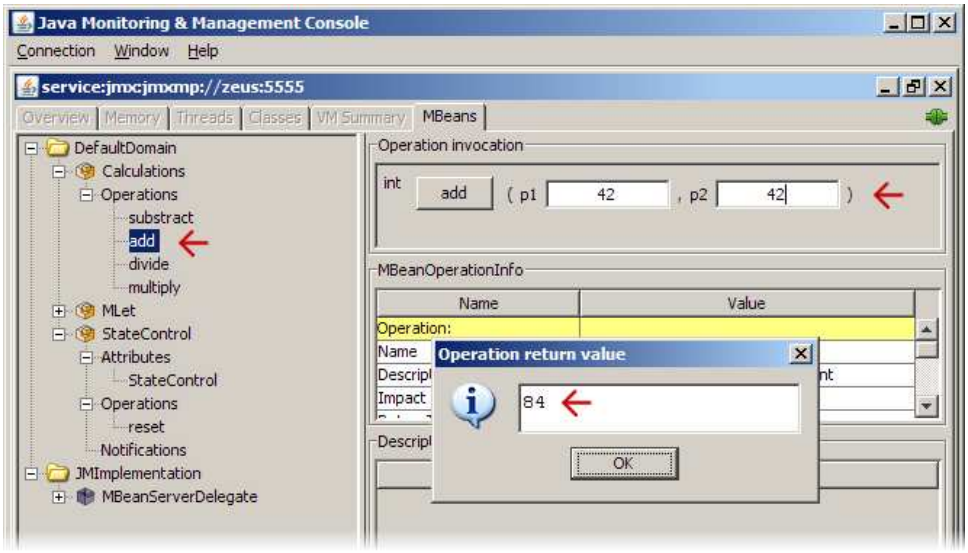

Figure 31. Invoking the add method exposed by the Calculations MBean to add 42 + 42.

7 Conclusions and Future Work

In this master thesis we have presented an architecture for the remote management of a Body Area Network that makes use of JMX as management technology and that is fully aligned with the Mobile Service Platform.

Using JMX allows us to easily instrument BAN resources for management and monitoring with little implementation effort. The BAN MBeanServer running on the MBU serves as a central repository for these resources to be registered and exposed for management on an easy and convenient way.

On the other hand, using MSP allows us to free the BANManagementService of the MBU device limitations by providing a surrogate object (i.e. the BANManagementSurrogate) that is uploaded to a Surrogate Host and that acts on behalf of the BANManagementService. In order to establish a connection between these two components we have designed the JMXMSP Connector. This is a new type of connector based on the Generic Connector provided by JMX that makes use of the MSP interconnect as transport protocol to exchange the different JMX management operations.

With this architecture, management client applications interact with the BANManagementSurrogate in order to manage the BAN resources instead of directly accessing these resources through the BAN MBeanServer. This allows us to introduce some behavior between both components and explore the possibilities of a caching mechanism in order to leverage the management overhead on the BANManagementService.

An important aspect of the presented architecture is its transparency for the mobile service developer since in order to make a resource remotely manageable it simply needs to be instrumented using JMX by implementing its own specific MBean interface and registered with the BAN MBeanServer. The service developer does not have to know how the communication between the BANManagementService and the MBeanServerSurrogate is handled.

Furthermore, it is also transparent for client management applications as long as they are JMX compliant, because once a JMXConnector has been registered by the BANManagementSurrogate with the Lookup Service and retrieved by the management application, the communication is handled in the same way it would if the connection had been established directly with the MBeanServer on the mobile device.

The proposed BAN management architecture and the prototype developed require, however, further research in some areas in order to become a viable solution.

From the tests presented in chapter 6 carried out to assess the performance of the current prototype implementation we conclude that no significant overhead is added by our management architecture in terms of the time it takes to perform a remote management operation on an MBean registered with the BAN MBeanServer. However, due to MSP itself, the time needed for management operations when the client interacts with the MBeanServerSurrogate is significantly larger than the time required for the same operations when performed directly on the BAN MBeanServer. By introducing a caching

mechanism on the MBeanServerSurrogate we have proposed a solution that can partially solve this issue. The simple caching mechanism implemented on the prototype results in management operations that take a similar amount of time when performed via the MBeanServerSurrogate or directly on the BAN MBeanServer. Furthermore, the caching mechanism can greatly reduce the management overhead on the BAN, by reducing the amount of management operations that are performed by the BAN MBeanServer. However, this mechanism also introduces an issue with management operations stored on the cache being out-of-date or stale. This issue requires further research, the current caching implementation being just a proof of concept.

But caching is not the only behavior we can introduce between the management application and the BAN MBeanServer by adding functionality to the MBeanServerSurrogate. We could think, for instance, of an access control mechanism to improve on the security facilities already provided by JMX (i.e. password authentication and SSL) where management applications are assigned different access levels to the BAN resources. Users could, for instance, be assigned roles of “reader” to just be able to read management data or “administrator” to be able to read management data as well as modify management attributes (e.g. a doctor remotely modifying attribute values for the monitoring of the target patient) and perform other management operations. This improved security mechanism would allow for a more fine-grained access control to manage the BAN resources.

Additionally, apart from providing instrumentation for the BAN resources (i.e. sensors, etc) further work on the instrumentation of the management architecture itself needs to be considered. That is, studying the instrumentation possibilities of the BANManagementService running on the mobile device and of the BANManagementSurrogate as well as the different components of the JMXMSP Connector.

We consider the exploration of the MBeanServerSurrogate possibilities and the further instrumentation of the different management architecture components a relevant and challenging research topic for improving the proposed BAN management architecture.

8 Bibliography

[1] Bert-Jan van Beijnum, Val Jones, Ing Widya, Pravin Pawar, and Hailiang Mei, "A Management Architecture for Nomadic MSP Based Body Area Networks," University of Twente, Awareness Deliverable 4.37 December December 2007.

[2] R J van den Berg, "The management of a BAN using JMX 1.2.1, a case study," Enschede, Bachelor Thesis for Telematics 2006.

[3] Val Jones et al., "Mobihealth: Mobile Health Services Based on Body Area Networks," 2006.

[4] Tom Broens, Aart van Halteren, Val Jones, Boris Shishkov, and Ing Widya, "Modelling of Body Area Networks," University of Twente, Awareness Deliverable 4.7 December 2005.

[5] Bluetooth SIG, Inc. (2009, February) Bluetooth.com. [Online]. http://www.bluetooth.com/

[6] ZigBee Alliance. [Online]. http://www.zigbee.org/

[7] V.M. Jones et al., "Biosignal and Context Monitoring: Distributed Multimedia Applications of Body Area Networks in Healthcare," 2008.

[8] Val Jones, Richard Bults, Dimitri Konstantas, and Pieter AM Vierhout, "Healthcare PANs: Personal Area Networks for trauma care and home care," 2001.

[9] Val Jones, H. Mei, T. Broens, and J. Peuscher, "Context Aware Body Area Networks for Telemedicine," 2007.

[10] Jini Architecture Specification. [Online]. http://java.sun.com/products/jini/2.0.1/doc/specs/html/jini-spec.html

[11] Jan Newmarch, *Foundations of Jini 2 Programming*.: Apress, 2006.

[12] Jini Technology Surrogate Architecture Specification. [Online]. https://surrogate.dev.java.net/doc/sa.pdf

[13] Jini Technology Surrogate Architecture Overview. [Online]. https://surrogate.dev.java.net/doc/overview.pdf

[14] Jini Technology IP Interconnect Specification. [Online]. https://ipsurrogate.dev.java.net/doc/sa-ip.pdf

[15] IP Surrogate Project. [Online]. https://ipsurrogate.dev.java.net/specs.html

[16] Aart van Halteren and Pravin Pawar, "Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning," Twente University, Enschede, June 2006.

[17] Pravin Pawar, Bert-Jan van Beijnum, Arjan Peddemors, and Aart van Halteren, "Context-Aware Middleware Support for the Nomadic Mobile Services on Multi-homed Handheld Mobile Devices".

[18] (2006, November) Java Management Extensions (JMX) Specification, version 1.4.

[19] Project OpenDMK. [Online]. https://opendmk.dev.java.net/

[20] Sun Microsystems, Inc. Java Management Extensions Remote API 1.0.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*.: Addison-Wesley Publishing Co., 1995.

[22] Eamonn McManus. Adding information to a Standard MBean interface using annotations. [Online]. http://weblogs.java.net/blog/emcmanus/archive/2005/07/adding_informat.html

[23] Steve Wilson and Jeff Kesselman, *Java Platform Performance: Strategies and Tactics*.: Prentice Hall PTR, 2001.

[24] Pravin Pawar et al., "Performance Evaluation of the Context-Aware Handover Mechanism for the Multi-Homed Nomadic Mobile Services," *Elsevier's Computer Communication Journal*, vol. Volume 31, no. Issue 16, pp. pp 3831-3842, October 2008.

[25] Sun Microsystems, Inc. (2004) Java Dynamic Management Kit 5.1 Tutorial.

[26] Jini Technology Starter Kit v2.1. [Online]. https://starterkit.dev.java.net/downloads/index.html

[27] Jini.org - Jini Technology Wiki. [Online]. http://www.jini.org/

[28] Apache River. [Online]. http://incubator.apache.org/river/RIVER/index.html

[29] Apache Commons Project. [Online]. http://commons.apache.org/

Appendix A – Setup Configuration

The following describes how to configure the BAN management architecture on a computer running Windows as operating system.

Installing Jini and MSP:

In order to install Jini, the Jini Technology Starter Kit and its installation instructions can be obtained from [26]. In order to install MSP, the developer should contact MobiHealth to request for permission and the installation procedure.

Running an HTTP server and a Lookup Service:

The Jini Technology Starter Kit provides both a simple class server and a Lookup Service implementation. The provided class server can be started using the following instruction on a command window:

```
java -Djava.security.policy=http.policy -jar lib/start.jar http.config
```

Note that a policy file called http.policy is used to set the -Djava.security.policy property for the class server's JVM, and an http.config file is used to specify some properties like the port it will be listening to (by default 8091). For convenience, while developing the http.policy file may specify java.security.AllPermission permissions. However, this should be modified on any other deployment scenario.

As for the Lookup Service, Jini provides an implementation of such service, included on the Starter Kit in a file called Reggie.jar. This service can be started with the following instruction on a command window:

```
java -Djava.security.policy=lus.policy -jar lib/start.jar lus.config
```

Again, the http.pollicy file is used to set the -Djava.security.policy property for the JVM executing the Lookup Service and a lus.config file is use to specify any other configuration parameters.

The Surrogate Host:

MSP includes a Surrogate Host implementation based on the Madison Surrogate Host contributed by Sun Microsystems. For details on how to configure the MSP Surrogate Host, the developer should contact MobiHealth.

The most relevant parameter, however, is the port where the Surrogate Host is listening for client connections. In our setup, the Surrogate Host is running on a host named "zeus" and listening for connections on the port 1010. So for a device service to be able to obtain a ServiceHostConnection and register its surrogate with the Surrogate Host, the device service must use the following URL:

```
String registrationUrl = "jinish://localhost:1010";
```

Surrogate and MBeans Deployment:

When the MBeanServerSurrogate has been appropriately packaged in a JAR file, the file must be placed on the HTTP server in an accessible directory. Any MBeans that are going to be remotely instantiated via the MLet service must be placed on the HTTP server as well, properly packaged in a JAR file, together with the appropriate M-let text file. All these files must be located in a directory on the HTTP server that is known to the components that must download them. As an example, the current implementation is running the HTTP server on a host named "zeus" listening on the port 8091. We have placed the surrogate JAR file (called mh-demosurrogate-0.1-surrogate-assembly.jar) in a directory called surrogates in the server's root directory. This means that the BANManagementService must return the following URL through the getSurrogateJAR method:

```
private static final String SURROGATE_JAR_URL =
http://zeus:8091/surrogates/mh-demosurrogate-0.1-surrogate-
assembly.jar
```

The mbeans.jar containing the MBeans to be remotely created using the MLet MBean is placed on a directory called *mbeans* on the HTTP server's root, together with the mlet.txt file describing the different MBeans. This deployment means that the following URL must be used by the MLet MBean when loading the MBeans on the mbeans.jar:

```
URL mbeansURL = new URL("http://zeus:8091/mbeans/mlet.txt");
```

Running the Setup:

With the current setup, in order to start the different components involved in the BAN management architecture the different servers should be started (i.e. the Surrogate Host and the HTTP server) followed by the Lookup Service. The Lookup Service must be started after the HTTP server because the service itself is configured to retrieve some libraries available in the jini-dl and services-dl directories as JAR files.

Then the BANManagementService can be run using the DemoController class which provides an entry point for the BANManagementService, creating the ServiceHostConnection, registering the BANManagementSurrogate and starting the BANManagementService.

At this point, a client management application can query a Lookup Service to obtain the Service Object that allows the communication between the client and the MBeanServerSurrogate.

Appendix B – Code Snippets

What follows is a series of relevant code snippets used by the BAN management architecture.

Connection with the Surrogate Host and surrogate registration:

Class: DemoController

Method: start()

```
String registrationUrl = "jinish://localhost:1010";
// ServiceHostConnection extends MSPSurrogateHostConnection
ServiceHostConnection serviceHostConnection = (ServiceHostConnection)
Connector.open(registrationUrl);

BANManagementService service = new BANManagementService();
ServiceSurrogateConnection serviceSurrogateConnection =
serviceHostConnection.registerSurrogate(demoService);
service.setServiceSurrogateConnection(serviceSurrogateConnection);

// Here the BANManagementService is started.
service.startMBeanServerService();
```

BANManagementService startup:

Class: BANManagementService

Method: startMBeanServerService()

```
[...]
// BAN MBeanServer creation
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
// URL to be used by MLet MBean to obtain the Mlet text file
URL mbeansURL = new URL("http://zeus:8091/mbeans/mlet.txt");
// create ObjectName to register the MLet MBean
ObjectName mletName = new
ObjectName(mbs.getDefaultDomain()+":type=MLet");

MLet mlet = new MLet();
mlet.addURL(mbeansURL);
// register the MLet MBean
mbs.registerMBean(mlet, mletName);

// create the relay used to listen for connection requests
ServiceConnreqRelay connreq_relay = new ServiceConnreqRelay();
// create a new JMXMSPConnectionServer
JMXMSPConnectionServer connserver = new
JMXMSPConnectionServer(serviceSurrogateConnection);
connserver.setConnreqRelay(connreq_relay);
// create a JMXMSPConnectorServer
```

```
jmxmspcs = new JMXMSPConnectorServer(connserver, mbs);
//...and start it
jmxmspcs.start();

// MSPServant
ServiceServant servant = new ServiceServant(connreq_relay);
serviceSurrogateConnection.setServant(servant);
[...]
```

Decoding an MSP message received by the BANManagementService's ServiceServant:

Class: ServiceServant

Method: serveMessage(OnewayMessage msg)

The following code fragment is used to decode the javax.management.remote.message.Message encoded as a byte array and included on the body of the MSP message received. This code is for receiving an MBeanServerRequestMessage.

```
[...]
byte[] data = msg.getDecoder().decodeByteArray();
ByteArrayInputStream bais = new ByteArrayInputStream(data);
ObjectInputStream ois = new ObjectInputStream(bais);
// obtain the javax.management.remote.message.Message encoded on the
body of the MSP message
Message message = (Message) ois.readObject();
// just for debugging purposes, print the type of message received
if (message instanceof MBeanServerRequestMessage) {
     System.out.println("MBeanServerRequestMessage Received");
}
// the decoded message is put on the relay to be consumed by the
JMXMSPConnection's readMessage // method
relay.put(message);
[...]
```

Receiving a JMX operation message in the BANManagementService's JMXMSPConnection:

Class: JMXMSPConnection

Method: readMessage()

```
public Message readMessage() {
     // relay.take will block until a message is put on the relay
     Message message = (Message) relay.take();
return message;
}
```

Sending a JMX operation message using the BANManagementService's JMXMSPConnection:

Class: JMXMSPConnection

Method: writeMessage(Message msg)

The following code fragment shows how a message (in this case a HandshakeBeginMessage) is encoded in the body of an MSP OnewayMessage as a byte array and then is sent through the ServiceSurrogateConnection using the invokeMessage method.

```
[...]
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(msg);
byte[] data = baos.toByteArray();

if (msg instanceof HandshakeBeginMessage) {
      System.out.println("Sending HandshakeBeginMessage");
      OnewayMessage mbsmsg = (OnewayMessage) serviceSurrogateConnection
            .createMessage(nl.utwente.msp.messages.Message.T_ONEWAY,
      OperationIDs.MSG_TYPE_MBS_HANDSHAKE_BEGIN);
      Encoder enc = mbsmsg.getEncoder();
      enc.encodeByteArray(data);

      serviceSurrogateConnection.invokeMessage(mbsmsg, (byte) 10));
}
[...]
```

MBeanServerSurrogate creation:

Class: BANManagementSurrogate.

Method: activate(HostContext hostContext, Object serviceContext)

```
[...]
// getting a ServiceConnection from serviceContext
serviceConnection = (ServiceConnection) serviceContext;
// get initialization data
byte[] initData = serviceConnection.getInitializationData();

// lifeHandler takes care of Liveness
lifeHandler = new LifeHandler();
serviceConnection.setKeepAliveHandler(lifeHandler);
serviceConnection.setLivenessHandler(lifeHandler);

// Create and start the MBeanServerSurrogate
MBeanServerSurrogate mbss = new MBeanServerSurrogate();
mbss.setServiceConnection(serviceConnection);
mbss.startMBSSurrogate();
```

Creating the MBeanServer proxy instance, connecting to the BAN MBeanServer and exporting the JMXMPConnector:

Class: MBeanServersurrogate

Method: startMBSSurrogate()

```
[...]
JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);

// JMXMSPConnection setup.
// JMXMSPConnection will connect MBSSurrogate with the MBS.
SurrogateMessageRelay relay = new SurrogateMessageRelay();
JMXMSPConnectionSurrogate conn = new
JMXMSPConnectionSurrogate(serviceConnection);
conn.setMessageRelay(relay);
SurrogateServant servant = new SurrogateServant(relay);
serviceConnection.setServant(servant);

// create JMXMSPConnector
jmxmspc = new JMXMSPConnector(conn);
jmxmspc.connect();

// The Handler will take care of the method invocations on the proxy
Instance
final MBSSurrogateHandler proxyHandler = new MBSSurrogateHandler(
                          jmxmspc.getMBeanServerConnection());
// Create a ProxyInstance for the MBeanServer interface, with our
invocation Handler and the
// SystemClassLoader
final MBeanServer remote = (MBeanServer) Proxy.newProxyInstance(
     ClassLoader.getSystemClassLoader(), new Class[] {
MBeanServer.class }, proxyHandler);

// Start a JMX RMI/JRMP connector server for the MBSSurrogate.
// This is the connector server that will be used by clients.
final JMXConnectorServer cs = JMXConnectorServerFactory
.newJMXConnectorServer(url, null, remote);
cs.start();

[...]
// create JMXConnectorToJini to register a JMXMPConnector with the
Lookup Service
JMXConnectorToJini jmxcToJini = new JMXConnectorToJini(cs);
[...]
```