Report

Bachelor assignment:

Assessment of Wireless CORBA in MobiHealth context

<table>
<thead>
<tr>
<th>Author</th>
<th>Jasper Aartse Tuijn (s0004685)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>12 July 2004</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Dr.ir. A.T. van Halteren</td>
</tr>
</tbody>
</table>
Abstract

In the last years, new technologies are more focused on mobile and wireless use. This change of focus implies that existing 'settled' techniques have to adapt to the new situation in order to let them be usable for the newest generations of applications. Wireless CORBA has been introduced to add support for wireless and mobility aspects to the existing CORBA standard. This report describes and documents a practical study on how this technology can be used in the existing MobiHealth system. The result from this study is a comparison on the use of the wireless connection between the currently used protocol that is based on HTTP and a redesigned protocol based on the new Wireless CORBA technology.
Assessment of Wireless CORBA in MobiHealth context
Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>I</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>III</td>
</tr>
<tr>
<td>PREFACE</td>
<td>V</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 CONTEXT</td>
<td>1</td>
</tr>
<tr>
<td>1.2 ASSIGNMENT</td>
<td>1</td>
</tr>
<tr>
<td>1.3 APPROACH AND PLANNING</td>
<td>1</td>
</tr>
<tr>
<td>2 THE MOBIHEALTH SERVICE PLATFORM</td>
<td>3</td>
</tr>
<tr>
<td>2.1 INTRODUCTION</td>
<td>3</td>
</tr>
<tr>
<td>2.2 BODY AREA NETWORK</td>
<td>3</td>
</tr>
<tr>
<td>2.3 JINI AND MOBIHEALTH</td>
<td>4</td>
</tr>
<tr>
<td>2.4 BANIP</td>
<td>6</td>
</tr>
<tr>
<td>3 ANALYSES OF THE BANIP</td>
<td>7</td>
</tr>
<tr>
<td>3.1 EXTERNAL PERSPECTIVE</td>
<td>7</td>
</tr>
<tr>
<td>3.2 INTERNAL PERSPECTIVE</td>
<td>8</td>
</tr>
<tr>
<td>3.3 PROTOCOL DATA UNITS</td>
<td>11</td>
</tr>
<tr>
<td>3.4 PROTOCOL CALCULATIONS</td>
<td>13</td>
</tr>
<tr>
<td>3.5 HTTP CALCULATIONS</td>
<td>17</td>
</tr>
<tr>
<td>4 CORBA</td>
<td>19</td>
</tr>
<tr>
<td>4.1 INTRODUCTION</td>
<td>19</td>
</tr>
<tr>
<td>4.2 IDL</td>
<td>19</td>
</tr>
<tr>
<td>4.3 NAMING SERVICE</td>
<td>20</td>
</tr>
<tr>
<td>4.4 EVENT SERVICE</td>
<td>21</td>
</tr>
<tr>
<td>4.5 ORB</td>
<td>21</td>
</tr>
<tr>
<td>4.6 GIOP AND IIOP</td>
<td>22</td>
</tr>
<tr>
<td>4.7 IOR</td>
<td>23</td>
</tr>
<tr>
<td>5 WIRELESS CORBA</td>
<td>24</td>
</tr>
<tr>
<td>5.1 INTRODUCTION</td>
<td>24</td>
</tr>
<tr>
<td>5.2 ARCHITECTURE</td>
<td>25</td>
</tr>
<tr>
<td>5.3 MOBILE IOR</td>
<td>26</td>
</tr>
<tr>
<td>5.4 GIOP TUNNELING PROTOCOL</td>
<td>27</td>
</tr>
<tr>
<td>5.5 HANDOFFS</td>
<td>28</td>
</tr>
<tr>
<td>5.6 ACCESS RECOVERY</td>
<td>31</td>
</tr>
<tr>
<td>5.7 GTP MESSAGE FORWARDING</td>
<td>32</td>
</tr>
<tr>
<td>6 REDESIGN OF BANIP USING CORBA</td>
<td>33</td>
</tr>
<tr>
<td>6.1 INTRODUCTION</td>
<td>33</td>
</tr>
<tr>
<td>6.2 INTERNAL ARCHITECTURE</td>
<td>34</td>
</tr>
<tr>
<td>6.3 INTERFACES</td>
<td>36</td>
</tr>
<tr>
<td>6.4 USING WIRELESS CORBA</td>
<td>37</td>
</tr>
</tbody>
</table>
Assessment of Wireless CORBA in MobiHealth context

7 ANALYSIS AND COMPARISON.......................................................................................... 42
  7.1 TEST-ENVIRONMENT ........................................................................................................ 42
  7.2 STARTING THE TEST-SOFTWARE ....................................................................................... 43
  7.3 ANALYSIS OF THE GTP MESSAGES.................................................................................. 46
  7.4 USAGE OF WIRELESS CONNECTION ............................................................................... 49

8 CONCLUSION.......................................................................................................................... 51

9 REFERENCES.......................................................................................................................... 53

APPENDIX A – TIME SCHEDULE .......................................................................................... 54

APPENDIX B - INTERFACE DEFINITIONS .............................................................................. 55
  B.1 IIOPInterconnect.idl ........................................................................................................... 55
  B.2 IDL FOR ALTERNATIVE WIRELESS CORBA BANip ....................................................... 56

APPENDIX C – C++ CODE ........................................................................................................ 57
  C.1 BANServer.h ...................................................................................................................... 57
  C.2 BANServer.cc .................................................................................................................. 57

APPENDIX D – JAVA CODE ..................................................................................................... 60
  D.1 IIOPInterconnectServer.java ........................................................................................... 60
  D.2 RegistrarImpl.java ............................................................................................................ 61
  D.3 SurrogateImpl.java .......................................................................................................... 62
Preface

With this assignment I conclude the D2-phase (In the new Bachelor/Master-system called the Bachelor-phase) of my study Telematics at the University of Twente. I carried-out the assignment internally at the University of Twente from April until July 2004.
I heard about this assignment while talking to my supervisor about the Bachelor assignment. I was very interested because of the challenges this assignment offered: A new Technology (Wireless CORBA), a programming language (C++), which is new for me and the use of a Linux-based environment, which I already have experience with.

I would like to thank my supervisor Aart van Halteren; he always had positive comments and good ideas, which really motivates me. I also want to thank every one else who showed interest in what I was doing.

Jasper Aartse Tuijn

Enschede, 12th of July 2004
1 Introduction

This chapter introduces the environment of the assignment. The first paragraph describes the context where the assignment is taking place. The second paragraph then describes the assignment itself, followed by the approach and planning which describe how the goals of the assignment are reached.

1.1 Context

The website of the MobiHealth project says:

“MobiHealth is a mobile healthcare project funded by the European Commission. The MobiHealth consortium unites 14 partners from five European countries and represents all the relevant disciplines.

Partners include hospitals and medical service providers, universities, mobile network operators, mobile application service providers and mobile infrastructure and hardware suppliers.

The MobiHealth system allows patients to be fully mobile whilst undergoing health monitoring.

The patients wear a lightweight monitoring system - the MobiHealth BAN (Body Area Network) -, which is customized to their individual health needs.

Therefore, a patient who requires monitoring for short or long periods does not have to stay in hospital for monitoring. With the MobiHealth BAN the patient can be free to pursue daily life activities.”

[MobiHealth 2004]

A BAN is a promising tool for monitoring various signals of the wearer, such as vital signs and location. The BAN data is conveyed to a healthcare professional using a public wireless network technology such as GPRS or UMTS. Currently, the BAN data is encapsulated in an HTTP based application protocol, the BAN interconnect protocol. HTTP introduces a significant overhead and thus reduces the effective data rate.

1.2 Assignment

The MobiHealth project had the use of HTTP as underlying protocol as a boundary condition. Within this assignment, this boundary condition does not exist. Together with the development of Wireless CORBA this is where the idea came from to develop a wireless CORBA based BAN interconnect protocol.

The Wireless CORBA specification is tailored to communication with wireless mobile devices. It is expected that the use of a wireless CORBA implementation will provide a more efficient means to convey BAN data. The goal of this assignment is to investigate and demonstrate how to use the wireless CORBA specification to control the BAN.

1.3 Approach and planning

This paragraph describes the approach and planning of the assignment. First, the global tasks are defined then the main-tasks are divided into smaller tasks.
**Approach**
To establish the desired result first the broad outlines are defined, this results in the following main-tasks:

- Set-up a MICO CORBA C++ environment
- Set-up a Wireless CORBA environment
- Compare efficiency of the standard CORBA protocol with the wireless CORBA protocol.
- Build a demonstrator that visualizes the differences in efficiency.
- Report and presentation.

The first two are for the set-up of the environment needed for the research environment and the learning process to get some experience with (Wireless) CORBA and C++. The second two main-tasks are the real research and testing. In the last task, the final report will be completed and the assignment will be finalized with a presentation.

**Tasks**
To make a planning the main-tasks must be divided into practical tasks:

1. Set-up a MICO CORBA C++ environment
   - Learning basics of C++
   - Using MICO

2. Set-up a Wireless CORBA environment
   - Set-up of MIWCO
   - Testing MIWCO

3. Compare efficiency of the HTTP based CORBA BANip with the wireless CORBA based BANip.
   - Make and implement MobiHealth application protocol (BAN interconnect protocol) using CORBA
   - Research the overhead introduced by the current interconnect protocol in the MobiHealth project.
   - Research overhead introduced by Wireless CORBA in MobiHealth project.
   - Compare the results in theory.
   - Compare the results in practice.

4. Build a demonstrator that visualizes the differences in efficiency.
   - Implement a demonstrator that visualizes the practical results obtained by the previous task.

5. Report and presentation.
   - Combine the different documents as one report and deliver this report.
   - Prepare a presentation.
   - Deliver presentation.

The deadline for the delivery of the report is 5 July 2004
In the time schedule, these tasks are mapped onto the available weeks. The time schedule is included in appendix A.
2 The MobiHealth Service Platform

This chapter describes how the MobiHealth system is constructed. The greater part of the content of this chapter originates from the paper ‘BANip: enabling remote healthcare monitoring with Body Area Networks’ [Dokovsky 2003]

2.1 Introduction

The MobiHealth service platform enables remote monitoring of patients using 2.5/3G public wireless infrastructures. Patient data is collected using a Body Area Network (BAN). A healthcare practitioner can view and analyze the patient data from a remote location. In this setting, the BAN acts as a provider of patient data and the healthcare practitioner acts as a user of that data. [Dokovsky 2003]

2.2 Body Area Network

The healthcare BAN consists of sensors, actuators, communication and processing facilities. Depending on what type of patient data must be collected, different medical sensors can be integrated into the BAN. For example, to measure pulse rate and oxygen saturation, an oximeter sensor is attached to the patients’ finger.

Communication between entities within a BAN is called intra-BAN communication. The current prototype uses Bluetooth for intra-BAN communication. To use the BAN for remote monitoring external communication is required which is called extra-BAN communication. The gateway that facilitates extra-BAN communication is called the Mobile Base Unit (MBU). The current prototype employs an HP iPAQ H3870, that runs Familiar Linux and a J2ME [7] compliant Java virtual machine, as an MBU. Other mobile devices, such as a Smart Phone or a J2ME enabled PDA could also act as an MBU. Figure 2-1 shows the architecture of a BAN. Sensors and actuators establish an ad-hoc network and use the MBU to communicate outside the BAN.

Figure 2-1: BAN architecture [Dokovsky 2003]
The extra-BAN communication where is referred to implements the BAN interconnect protocol which is the subject of this assignment. The MBU is one of the end-points of the communication protocol.

### 2.3 JINI and MobiHealth

This paragraph describes what JINI is and what it does to contribute to the MobiHealth Service Platform and how it relates to this assignment.

**JINI**

One of the problems that the MobiHealth service platform solves is the dynamic discovery of the services offered by a BAN. A BAN is configured to the needs of end-users and can typically be switched on/off anytime at the convenience or the necessity of the end-user (e.g., patient). A key design decision is to represent the services offered by a BAN as a JINI service. JINI provides mechanisms whereby the service lifetime, service location and service implementation details become irrelevant to the service user.

A JINI service provider can dynamically attach or detach itself from a given JINI network community, a so-called djinn. A special core service, called LookUp Service (LUS), supports the registration of services. A service provider (e.g., a BAN) registers a Service Proxy into LUS together with a set of predefined service attributes. Before a service provider can register a service, it must contact the LUS, which is the discovery process (i.e., interaction 1 and 3 in Figure 2-2). The figure shows the interactions between the service provider, service user and the LUS, the first 4 interactions belong to the set-up phase and are preliminary to the service execution phase, i.e., the data transfer phase in monitoring applications. An essential element of the discovery and registration process is the exchange of a Service Proxy. This proxy encapsulates the client part of the service logic and communication protocols needed for service execution.

![Figure 2-2: Service discovery and registration](image)

A service user only needs to know the interface to the service. In some cases, the service proxy includes the whole service implementation, moving the service execution entirely into the service users’ virtual machine (VM). In other cases, the service proxy contains networking functionality, which will allow the service user to connect to the remote service provider. The MobiHealth platform uses the latter scheme, which allows for adaptation of the protocol used for service execution, without changes in the service user (e.g., the peer client of the monitoring system at a healthcare center).
When a service user discovers a LUS, it can browse either for a particular service or it can register itself for service availability notifications based on event messages.

**Leasing**

Service resources in a JINI network are acquired for a certain period based on a service lease. A service user must renew a lease within certain period. If that lease cannot be renewed, for whatever reason (network failure, system crash, etc.), the client can conclude that the service is not available anymore. Similarly, the service provider can free specific resources just because a service user has not renewed a granted lease.

**JINI Surrogate Architecture**

Although service execution can be based on any protocol, JINI requires that RMI (Remote Method Invocation) is available for LUS discovery. Consequently, a service provider must execute in a Java VM that supports RMI. Limited resources inhibit the use of RMI on the MBU, therefore making the MBU unfit as a host for a JINI service provider.

The MobiHealth project employs the JINI Surrogate Architecture (Figure 2-3) to enable a BAN to still offer services in a JINI network.

According to the Surrogate Architecture, a device that initially cannot join a JINI network because it does not meet the connectivity and functional requirements, can still join if a surrogate object represents it. The surrogate object acts as a service provider on behalf of the device and shields service users from the specific means to communicate with the device. At application level, this yields that a healthcare practitioner at a healthcare centre transparently retrieve BAN data from the surrogate host as if he retrieves the data from the BAN. In a multi stakeholder m-health delivery model, surrogate hosts typically reside at the domains of the call centers.

Furthermore, surrogates rely on a Surrogate Host for life cycle management and a runtime environment. Device specific communication between the surrogate and the device is called the Interconnect Protocol, which is the subject of this assignment. As described in paragraph 2.2 one endpoint of the Interconnect Protocol consists of the MBU, the other endpoint is the Surrogate Host.

![Figure 2-3: Elements of Surrogate Architecture](image)

The surrogate architecture specification requires the interconnect protocol to fulfill at least three mechanisms: discovery, surrogate upload and keep-alive.
The purpose of the discovery mechanism is to make a surrogate host aware of the device existence and vice versa. Implementation depends on the device communication capabilities.

Once a device and the surrogate host discovered each other, the device can join the JINI network. To do so, the surrogate host must be provided with the surrogate object that will act in the JINI network on the behalf of the device. The device itself can upload the surrogate object or it can send to the surrogate host the location point from where the surrogate can be downloaded.

After a surrogate has been instantiated and activated by the surrogate host, the device must guarantee that it is able to perform the exported service. Consequently, the interconnect protocol must implement a keep-alive mechanism to inform the surrogate host if the device is still active and connected. As soon as the device cannot confirm its online status, the surrogate host can deactivate the JINI service and its correspondent surrogate object.

[Dokovsky 2003]

### 2.4 BANip

The MobiHealth service platform uses a surrogate object to act on behalf of the MBU and thus allows a BAN to offer its services in a JINI network despite its resource limitations. The BAN interconnect protocol (BANip) is the protocol for interaction between the MBU surrogate and the MBU device. The BANip fulfils the three basic requirements of the surrogate architecture (i.e. discovery, surrogate upload and keep-alive).

The MBU-device has, when connected to a GPRS network, a private IP-address, this means the MBU-devices cannot be directly be reached from the Surrogate Host. The current HTTP-based BANip solves this by sending the command that must be sent to the MBU as reply to the Keep-alive messages. This means when the keep-alive messages are sent every 30 seconds, there can be only sent one command addressed to the MBU in every 30 seconds.

The current BAN interconnect protocol uses HTTP as Lower Level Service; there might be better alternatives to HTTP. This assignment looks into Wireless CORBA as the BANip. To make a good comparison, the current BANip must be analyzed. The next chapter describes this analysis of the current HTTP-based BAN interconnect protocol.

[Dokovsky 2003]
3 Analyses of the BANip

This chapter gives a detailed analysis of the BAN interconnect protocol. The first paragraph describes the service provided by the protocol to the users of the protocol. Paragraph 4.1 will then describe how the protocol is internally structured to provide this service. The protocol uses Protocol Data Units (PDUs) to communicate between the Protocol Entities; paragraph 3.3 describes these. Paragraph 3.4 presents the calculations that describe how much data the protocol uses of the connection.

The modeling scheme used in this chapter comes from the lecture notes [Visser 2002] of the lecture ‘The Design of Telematics Systems’ at the University of Twente.

3.1 External perspective

The users of the BAN interconnect protocol are located at two different places. The Surrogate Host is located on a fixed network; the other user is the device that is connected to a mobile network.

Figure 3-1 shows how the SAPs connect these users to the BAN Interconnect Service Provider.

![Figure 3-1: BAN Interconnect Service Provider and the Service Users](image)

There are two Service Access Points (SAPs) located in the picture. At each one certain Service Primitives can occur, which let the users communicate with each other using the protocol.

Table 3-1 shows the Service Primitives which can occur at SAP 1, they give access to the registration process and sensor-data subscriptions.
Table 3-1: Service Primitives at SAP 1

<table>
<thead>
<tr>
<th>SU &lt;&gt;SP</th>
<th>Interaction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>registration_ind</td>
<td>MBUIdentifier</td>
</tr>
<tr>
<td>=&gt;</td>
<td>request_subscription_req</td>
<td>SensorDataComposition, transcodingChain</td>
</tr>
<tr>
<td>=&gt;</td>
<td>activate_subscription_req</td>
<td>SensorDataComposition, SubscriptionIdentifier, transcodingChain</td>
</tr>
<tr>
<td>=&gt;</td>
<td>deactivate_subscription_req</td>
<td>SensorDataComposition, SubscriptionIdentifier, transcodingChain</td>
</tr>
<tr>
<td>&lt;=</td>
<td>data_ind</td>
<td>SensorData</td>
</tr>
</tbody>
</table>

The first two Service Primitives (SP) let the MBU register itself at the Surrogate Host; the succeeding three let the Surrogate Host manage the Sensor Data that will be sent. The last Service Primitive is needed to receive the Sensor Data.

The Service Primitives shown in Table 3-1 do depend on each other: The request_subscription_req can only occur after a registration_ind has occurred. Only when the request_subscription_req has occurred the activate_subscription_req can occur. After the activate_subscription_req, there can be more occurrences of data_ind or one occurrence of deactivate_subscription_req, after which only activate_subscription_req can occur.

Table 3-2 shows the Service Primitives which can occur at SAP 2. This SAP let the user register itself and this SAP provides the Service Primitives for the management of the subscription on Sensor Data.

Table 3-2: Service Primitives at SAP 2

<table>
<thead>
<tr>
<th>SU &lt;&gt;SP</th>
<th>Interaction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt;</td>
<td>registration_req</td>
<td>MBUIdentifier</td>
</tr>
<tr>
<td>&lt;=</td>
<td>registration_response_ind</td>
<td>SurrogateIdentifier</td>
</tr>
<tr>
<td>&lt;=</td>
<td>request_subscription_ind</td>
<td>SensorDataComposition, transcodingChain</td>
</tr>
<tr>
<td>&lt;=</td>
<td>activate_subscription_ind</td>
<td>SensorDataComposition, SubscriptionIdentifier, transcodingChain</td>
</tr>
<tr>
<td>&lt;=</td>
<td>deactivate_subscription_ind</td>
<td>SensorDataComposition, SubscriptionIdentifier, transcodingChain</td>
</tr>
<tr>
<td>=&gt;</td>
<td>sent_data_req</td>
<td>Sensor Data</td>
</tr>
</tbody>
</table>

Also at this SAP, the Service Primitives depend on each other: The registration_response_ind can only occur after a registration_req. The request_subscription_ind can only occur after a registration_response_ind has occurred. Only when the request_subscription_ind has occurred the activate_subscription_ind can occur. After the activate_subscription_ind, there can be more occurrences of data_ind or one occurrence of deactivate_subscription_ind, after which only activate_subscription_ind can occur.

3.2 Internal perspective

The previous paragraph describes the communication of the BAN Interconnect Service Provider with the environment. This paragraph describes few decomposition steps to describe the internal architecture of the BAN interconnect service provider.
3.2.1 First decomposition step

The first decomposition will explain the two Protocol Entities, the Lower Level Service (HTTP) and the interaction between them. Figure 3-2 shows the first decomposition step.

Figure 3-2: BAN interconnect protocol Entities and HTTP as Lower Level Service

The picture shows that there are two extra Service Access Points between both Protocol Entities (PE) and the Lower Level Service; these describe how the Protocol Entities make use of HTTP as the Lower Level Service (LLS). Table 3-3 shows the Service Primitives which can occur at SAP 3. Table 3-4 shows the Service Primitives which can occur at SAP 4.

<table>
<thead>
<tr>
<th>PE &lt;-&gt; LLS</th>
<th>Interaction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>http_request_ind</td>
<td>header, body</td>
</tr>
<tr>
<td>=&gt;</td>
<td>http_response_req</td>
<td>header, body</td>
</tr>
</tbody>
</table>

Table 3-3: Service Primitives at SAP 3

<table>
<thead>
<tr>
<th>PE &lt;-&gt; LLS</th>
<th>Interaction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt;</td>
<td>http_request_req</td>
<td>header, body</td>
</tr>
<tr>
<td>&lt;=</td>
<td>http_response_ind</td>
<td>header, body</td>
</tr>
</tbody>
</table>

Table 3-4: Service Primitives at SAP 4

As shown in both tables the Protocol Entity at the mobile side sends the http_request to the Protocol Entity at the fixed side. The response is going back the other way.
Both the Protocol Entities use the BAN interconnect protocol to communicate with each other. To describe the BAN interconnect protocol, the next step of decomposition describes the components of the two Protocol Entities must be described.

**3.2.2 Second decomposition step**

By decomposing the two Protocol Entities, the interaction between those Protocol Entities can be explained in terms of the protocol elements communication with each other. This provides a much clearer image of the BAN interconnect protocol.

Figure 3-3 illustrates how the different protocol elements in the Protocol Entities communicate with each other using the Lower Level Service.

An important difference with the previous abstraction level as described in the previous paragraph is the splitting of SAP 1 in two SAPs; this is done because the components of the fixed side Protocol Entities (Surrogate and HttpAdapter) use different Service Primitives. The Service Primitives that are used in the registration process can occur at SAP 1B, the rest can occur at SAP 1A.

Together these SAPs have the same functionality as in the previous paragraph.

Each of the new components of the Protocol Entities shown in Figure 3-3 is described below:
**HttpAdapter**: The HttpAdapter coordinates the registration process of the MBUs. If a MBU registers, it activates a Surrogate.

**Surrogate**: A Surrogate represents the MBU on the fixed side in the JINI network, it checks the connection between the two Protocol Entities, let the user subscribe on a Group of sensors and receives the sensor data that is subscribed on.

**Registrar**: The Registrar registers the MBU at the Surrogate Host, delegate’s subscription management to the User, maintains the connection by sending keep-alive messages and sends the sensor-data that the Surrogate is subscribed on.

**3.3 Protocol Data Units**

The protocol entities described in the previous paragraph communicate with each other via the Lower Level Service; this is done by exchanging PDUs. This paragraph describes all the PDUs that the Protocol Entities can transfer to each other. The structures of the PDUs shown in this paragraph are defined in IDL (see paragraph 4.2).

All PDU-messages are encapsulated in an HttpMessagePDU; this PDU consists of the protocol version, message type and the encapsulated PDU.

The BAN interconnect protocol defines the following PDUs:

**HttpMessagePDU**: This PDU always contains one of the other PDUs. The other PDUs are always encapsulated into the HttpMessagePDU. Besides the other PDU, it contains a protocol_version and the message_type. The struct ‘HttpMessagePDU’ in the C-code below defines this PDU:
const int UNKNOWN_MESSAGE = -1;
const int NOOP = 0;
const int SENSOR_DATA = 1;
const int KEEP_ALIVE = 2;
const int POLL_DATA = 3;
const int Deregister = 4;
const int REQUEST_SUBSCRIPTION = 5;
const int ACTIVATE_SUBSCRIPTION = 6;
const int DEACTIVATE_SUBSCRIPTION = 7;
const int CHUNKED_SENSOR_DATA = 8;
const int REGISTRATION_PDU = 9;
const int RESPONSE_PDU = 10;

union Message switch (int) {
    case UNKNOWN_MESSAGE:
    case NOOP:
    case SENSOR_DATA: //DEPRECATED!!!
    case KEEP_ALIVE:
        HttpKeepAlive keepAliveData;
    case POLL_DATA:
    case Deregister:
        case REQUEST_SUBSCRIPTION:
            HttpRequestSubscription requestSubscriptionData;
    case ACTIVATE_SUBSCRIPTION:
        HttpActivateSubscription activateSubscriptionData;
    case DEACTIVATE_SUBSCRIPTION:
        HttpDeactivateSubscription deactivateSubscriptionData;
    case CHUNKED_SENSOR_DATA:
    case REGISTRATION_PDU:
        HttpRegistrationPdu registrationData;
    case RESPONSE_PDU:
        HttpResponsePdu responsePduData;
};

struct HttpMessagePDU {
    short int protocol_version;
    unsigned int message_type;
    Message messageData;
};

**HttpRegistrationPDU:**
The Registrar sends this PDU to the HttpAdapter to do a request to register its MBU at the surrogate host. The C-code below defines this PDU:

```c
struct HttpRegistrationPdu {
    unsigned int surrogateUrl_length;
    string surrogateUrl;
    unsigned int initData_length;
    char[] initData;
    unsigned int surrogateJar_length;
    char[] surrogateJar;
};
```

**HttpResponsePDU:**
The HttpAdapter sends this PDU to the Registrar in response to an HttpRegistrationPDU. The C-code below defines this PDU:

```c
struct HttpResponsePdu {
    unsigned int connectURLLength;
    char[] connectURL;
};
```

**HttpRequestSubscription:**
The Surrogate sends this PDU to the HttpAdapter to subscribe the user to certain sensor data. The C-code below defines this PDU:
struct HttpRequestSubscription {
    unsigned int ssid_length;
    char[] ssid;
    unsigned int end_point_length;
    char[] end_point;
    unsigned int len_transcoding_chain;
    char[] transcoding_chain;
};

HttpActivateSubscription:
The Surrogate sends this PDU to the HttpAdapter to activate an existing subscription. The C-code below defines this PDU:

struct HttpActivateSubscription {
    unsigned int ssid_length;
    char[] ssid;
    unsigned int sue_length;
    char[] sue;
    unsigned int len_transcoding_chain;
    char[] transcoding_chain;
};

HttpDeactivateSubscription:
The Surrogate sends this PDU to the HttpAdapter to deactivate a currently activated subscription. The C-code below defines this PDU:

struct HttpDeactivateSubscription {
    unsigned int ssid_length;
    char[] ssid;
    unsigned int sue_length;
    char[] sue;
};

HttpKeepAlive:
The HttpAdapter sends this PDU to the Surrogate that represents the MBU to let the Surrogate know if the connection between them still exists. Keep alive messages are needed by the JINI platform. The C-code below defines this PDU:

struct HttpKeepAlive {
    unsigned int payload_length;
    char[] payload;
};

In the current MobiHealth system are also some other message-types defined, they contain no data:

- NOOP
- POLL_DATA
- Deregister
- CHUNKED_SENSOR_DATA
- UNKNOW_MESSAGE

3.4 Protocol calculations
This paragraph describes the calculations of the amount of data used by each PDU defined in the previous paragraph. The fields in a PDU are sent sequentially without a delimiter. Three different types of data are used: short int (Uses two bytes of data), byte, int (Uses four bytes of data).
The main message HttpMessagePDU that encapsulates one of the other PDUs has two fields besides the message field (which contains the encapsulated PDU). The protocol_version field is of type short int, which uses 2 bytes. The message_type field is of type int, which uses four bytes. Therefore, the size of a PDU that is send is increased by six bytes extra overhead.

Below the calculations are shown for each PDU.

### 3.4.1 HttpRegistrationPDU

There are three variable size fields in the HttpRegistrationPDU. To calculate the size of this PDU these values must be filled in, for this the default values, taken from the current implementation of the MobiHealth system, will be used.

**SurrogateUrl:**
The HttpRegistrationPDU consist of the surrogate jar-file or an URL to the surrogate jar-file. MobiHealth only uses the URL to the jar-file. The default value in the current system is “http://svc.mobihealth.org/jini/mbu_surrogate.jar” (ban/banware/src/org/mobihealth/management/subscription/impl/RegistrarFSM.java), so this will be used for the calculation. The URL consists of 48 characters, which takes 48 bytes of data.

**InitData:**
The initData field consists of a device name, which has the default value “test-device” (ban/banware/src/org/mobihealth/management/subscription/impl/RegistrarFSM.java); this consists of eleven characters, which results in eleven bytes of data.

**SurrogateJar:**
As described above, this value will not be used, so it is empty.

Table 3-1 shows the calculation of the total size.

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>surrogateUrl_length</td>
<td>4</td>
</tr>
<tr>
<td>surrogateUrl</td>
<td>48</td>
</tr>
<tr>
<td>initData_length</td>
<td>4</td>
</tr>
<tr>
<td>initData</td>
<td>11</td>
</tr>
<tr>
<td>surrogateJar_length</td>
<td>4</td>
</tr>
<tr>
<td>surrogateJar</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>71</strong></td>
</tr>
</tbody>
</table>

Table 3-5: Data calculation of the HttpRegistrationPDU

The resulting HttpMessagePDU consists of $71 + 6 = 77$ bytes.

### 3.4.2 HttpResponsePDU

The connectURL field of the HttpResponsePDU has a variable size. The URL is something like “http://aaa.bbb.ccc.ddd:eeee/”. This string is 28 characters long, which takes 28 bytes of data. Table 3-5 shows the calculation of the total size.
Assessment of Wireless CORBA in MobiHealth context

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectURLLength</td>
<td>4</td>
</tr>
<tr>
<td>connectURL</td>
<td>28</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>32</strong></td>
</tr>
</tbody>
</table>

Table 3-6: Data calculation of the HttpResponsePDU

The resulting HttpMessagePDU consists of 32 + 6 = 38 bytes.

### 3.4.3 HttpRequestSubscription

The HttpRequestSubscription message contains three variable size fields:

- **ssid**: The ssid-field contains the identifier of the SensorSet. In practice, it contains the identifier of the MBU; this identifier is a string of ten characters. Therefore, this field uses ten bytes.

- **End_point**: The default value is “http.push”, which consist of nine characters, which take nine bytes of data. (Source: “./banware/src/org/mobihealth/management/impl/MBUManager.java”)

- **transcoding_chain**: The default value is “Filter” (Source: “/banware/src/org/mobihealth/management/impl/SensorSetManager.java”). This string takes six bytes of data.

Table 3-7 shows the calculation of the total size.

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid_length</td>
<td>4</td>
</tr>
<tr>
<td>ssid</td>
<td>10</td>
</tr>
<tr>
<td>end_point_length</td>
<td>4</td>
</tr>
<tr>
<td>end_point</td>
<td>9</td>
</tr>
<tr>
<td>len_transcoding_chain</td>
<td>4</td>
</tr>
<tr>
<td>transcoding_chain</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>37</strong></td>
</tr>
</tbody>
</table>

Table 3-7: Data calculation of the HttpRequestSubscription message

The resulting HttpMessagePDU consists of 37 + 6 = 43 bytes.

### 3.4.4 HttpActivateSubscription

The HttpActivateSubscription message contains three variable-length fields.

- **ssid**: The same size as specified in the previous paragraph: ten bytes.
Assessment of Wireless CORBA in MobiHealth context

sue:
The URL is something like “aaaa://bbb.ccc.ddd.eee:ffff/”, which is 28 bytes of data.

Transcoding_chain:
The same size as specified in the previous paragraph: six bytes.

Table 3-8 shows the calculation of the total size.

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid_length</td>
<td>4</td>
</tr>
<tr>
<td>ssid</td>
<td>10</td>
</tr>
<tr>
<td>sue_length</td>
<td>4</td>
</tr>
<tr>
<td>sue</td>
<td>28</td>
</tr>
<tr>
<td>len_transcoding_chain</td>
<td>4</td>
</tr>
<tr>
<td>transcoding_chain</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>56</strong></td>
</tr>
</tbody>
</table>

Table 3-8: Data calculation of the HttpActivateSubscription message

The resulting HttpMessagePDU consists of 56 + 6 = 62 bytes.

3.4.5 HttpDeactivateSubscription

In the HttpDeactivationSubscription message are two variable size fields:

sue:
This field has the same size as the ssid-field in the previous paragraph: 10 bytes.

sue_length:
This field has the same size as the field in the previous paragraph: 28 bytes.

Table 3-9 shows the calculation of the total size.

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssid_length</td>
<td>4</td>
</tr>
<tr>
<td>ssid</td>
<td>10</td>
</tr>
<tr>
<td>sue_length</td>
<td>4</td>
</tr>
<tr>
<td>sue</td>
<td>28</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>46</strong></td>
</tr>
</tbody>
</table>

Table 3-9: Data calculation of the HttpDeactivateSubscription message

The resulting HttpMessagePDU consists of 46 + 6 = 52 bytes.

3.4.6 HttpKeepAlive

The variable size field “payload” is left empty in the current implementation of the MobiHealth system. Table 3-10 shows the calculation of the total size.
Assessment of Wireless CORBA in MobiHealth context

<table>
<thead>
<tr>
<th>Field</th>
<th>Nr of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload_length</td>
<td>4</td>
</tr>
<tr>
<td>payload</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3-10: Data calculation of the HttpKeepAlive message

The resulting HttpMessagePDU consists of $4 + 6 = 10$ bytes.

### 3.5 HTTP calculations

The HTTP protocol, which is used as Lower Level Service, uses headers that increase the amount of data used for a PDU. This paragraph contains the calculations of this overhead. The HTTP protocol uses a request/response model (When chunking is used, the server never gives a response; the client can keep sending data)

Each request consists of some headers and a body. Different headers are used for the different BANip-messages which are sent over HTTP, underneath the size of the HTTP-messages is calculated per BANip-message.

**RegistrationRequest:**

<table>
<thead>
<tr>
<th>Header</th>
<th>#bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
</tr>
<tr>
<td>User-Agent: Profile/MIDP-1.0</td>
<td></td>
</tr>
<tr>
<td>Configuration/CLDC-1.0</td>
<td></td>
</tr>
<tr>
<td>Content-Language: en-US</td>
<td>25</td>
</tr>
<tr>
<td>Accept:</td>
<td>16</td>
</tr>
<tr>
<td>text/*</td>
<td></td>
</tr>
<tr>
<td>Content-type: application</td>
<td>40</td>
</tr>
<tr>
<td>octet-stream</td>
<td></td>
</tr>
<tr>
<td>Connection: close</td>
<td>19</td>
</tr>
<tr>
<td>Pragma: no-cache</td>
<td>18</td>
</tr>
<tr>
<td>Content-length: 77</td>
<td>22</td>
</tr>
</tbody>
</table>

**Total nr of bytes in the header**

(Source: ban/banware/src/org/mobihealth/management/subscription/impl/RegistrarFSM.java)

The content-length is 77 bytes (see paragraph 3.4.1). The total amount of bytes used by the header and the body is $210 + 77 = 297$ bytes. This message consists of $(210/297 \times 100) = 71\%$ HTTP overhead.

**HttpKeepAlive:**

<table>
<thead>
<tr>
<th>Header</th>
<th>#bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
</tr>
<tr>
<td>User-Agent: Profile/MIDP-1.0</td>
<td></td>
</tr>
<tr>
<td>Configuration/CLDC-1.0</td>
<td></td>
</tr>
<tr>
<td>Content-Language: en-US</td>
<td>25</td>
</tr>
<tr>
<td>Accept:</td>
<td>16</td>
</tr>
<tr>
<td>text/*</td>
<td></td>
</tr>
<tr>
<td>Content-type: application</td>
<td>40</td>
</tr>
<tr>
<td>octet-stream</td>
<td></td>
</tr>
<tr>
<td>Pragma: no-cache</td>
<td>18</td>
</tr>
<tr>
<td>Content-length: 10</td>
<td>21</td>
</tr>
</tbody>
</table>

**Total nr of bytes in the header**

(Source: ban/banware/src/org/mobihealth/management/subscription/impl/RegistrarFSM.java)
The only differences in comparison with the header used in the RegistrationRequest message are the missing field “Connection” and the different content-length. The content length is 10 bytes according to paragraph 3.4.6, so the total size of the HTTP-message is 190 + 10 = 200 bytes, this message consists of (190/200 * 100) = 95% HTTP-overhead.

**HttpRegistrationResponse, HttpRequestSubscription, HttpActivateSubscription, HttpDeactivateSubscription:**

Those messages all use the same header; only the content-length field is different. The table below shows the calculations for the general part of the header:

<table>
<thead>
<tr>
<th>Header</th>
<th>#bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1 200 OK\r\n</td>
<td>17</td>
</tr>
<tr>
<td>Content-type: application/octet-stream\r\n</td>
<td>40</td>
</tr>
<tr>
<td><strong>Total nr of bytes in the general part of the header</strong></td>
<td><strong>57</strong></td>
</tr>
</tbody>
</table>

The empty OK-response also uses this header. When the content-length header of 20 bytes is added, the size of the empty OK-response is 57 + 20 = 77 bytes.

The HttpRegistrationResponse HTTP-message has a body part of 38 bytes (see paragraph 3.4.2), thus the content-length header is 21 bytes long. The total size of the HTTP-message is 57 + 21 + 38 = 116 bytes. This message consists of ((57 + 21)/116 * 100) = 67% HTTP-overhead.

The HttpRequestSubscription HTTP-message has a body part of 43 bytes (see paragraph 3.4.3), thus the content-length header is 21 bytes long. The total size of the HTTP-message is 57 + 21 + 43 = 121 bytes. This message consists of ((57 + 21)/121 * 100) = 64% HTTP-overhead.

The HttpActivateSubscription HTTP-message has a body part of 62 bytes (see paragraph 3.4.4), thus the content-length header is 21 bytes long. The total size of the HTTP-message is 57 + 21 + 62 = 140 bytes. This message consists of ((57 + 21)/140 * 100) = 56% HTTP-overhead.

The HttpDeactivateSubscription HTTP-message has a body part of 52 bytes (see paragraph 3.4.5), thus the content-length header is 21 bytes long. The total size of the HTTP-message is 57 + 21 + 52 = 130 bytes. This message consists of ((57 + 21)/130 * 100) = 60% HTTP-overhead.
4 CORBA

This chapter describes the Common Object Request Broker Architecture (CORBA); the first paragraph introduces CORBA, what it does, what is the purpose, etc. The other paragraphs describe the main-concept used in CORBA and which are important in the scope of this assignment.

4.1 Introduction

CORBA (Common Object Request Broker Architecture) is a middleware specification developed by the Object Management Group (OMG). CORBA supports interoperability between software components situated on different computing systems and programmed in different programming languages.

The idea behind CORBA is that a client can invoke a method on an object situated ‘somewhere’, possibly written in a different programming language. Figure 4-1 shows an invocation on an object.

![Figure 4-1: Invocation of a method on an object](image)

De client has to know the interface of the object, because the object is located on another location there has to be an explicit specification of this interface that is known by both sides. CORBA uses IDL (Interface Definition Language) for this purpose. The next paragraph describes IDL.

[Henning 99]

4.2 IDL

The Interface Definition Language is the language that is used to specify the types and interfaces used by the application, in order to separate object implementations and their interface. When the client and the server have the same IDL specification of the object, they are able to communicate with each other.

Because IDL is language-independent, there are compilers available for different languages. These compilers create a language specific types and APIs, which the server/client-implementations of the applications then can use.

The OMG has defined some standard CORBA-interfaces: The OMG Naming Service, the OMG Trading Service and the OMG Event Service. The next paragraphs will describe the Naming Service and the Event Service; the other services will not be described, because they are not needed for this assignment.

[Henning 99]
4.3 Naming Service

The OMG Naming Service allows a human readable name to be associated or bound to an object. The reference to that object can subsequently be found by resolving that name within the Naming Service.

Using the Naming Service a name is bound to an object relative to a naming context. Different names can be bound to an object in the same or different contexts at the same time, this is called a name binding. A naming context is an object that contains a set of name bindings in which each name is unique. In file management terms, a naming context is basically a directory structure for objects. A name is always resolved relative to a context; there are no absolute names. To resolve a name is to determine the object associated with the name in a given context. To bind to a name is to create a name binding in a given context.

Because a context is like any other object, it can also be bound to a name in a naming context, thus creating a naming graph. A naming graph allows more complex names to reference an object. For an example of a naming graph, see Figure 4-2. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a compound name) defines a path in the naming graph to navigate the resolution process.

A server registers an object reference with the Naming Service by binding the object reference to a naming context. This name can then be used by other components in the system to find the registered object.

![Figure 4-2: A naming graph](Halteren 98)
4.4 Event Service

The OMG Event Service allows an application to use a decoupled communications model. This means that the server and client are decoupled by the use of an event channel. The server, which is called the supplier, produce an event, then all the clients, who are called consumers, receive it. The Event Channel is responsible for the registration of the suppliers and consumers.

There are two different models specified to transport the events:

**The push model:** The supplier pushes an event on the Event Channel, and then the event channel pushes the event to the consumers.

**The pull model:** The consumer pulls an event from the Event Channel, and then the Event Channel pulls the event from the supplier. The direction of the flow of events is the other way around, compared to the push model.

[Henning 99]

4.5 ORB

When the Client invokes a method on the remote object, for the client it looks like it is a direct invocation. However, in reality CORBA transfers the invocation through the Object Request Broker (ORB) as seen in Figure 4-3. An ORB enables communication between clients and objects, transparently activating those objects that are not running when requests are delivered to them.

![Figure 4-3: Invocation of an operation on a CORBA object](image)

When the IDL specification of the interface is compiled for a certain language, the IDL-compiler generates stubs and skeletons generated for the object. The stub is the connection between the client and the ORB at the client side, the server-side uses the skeleton, to do the real invocation on the object. Figure 4-4 shows the architecture of CORBA.
A request from the client is invoked via either a static stub or the Dynamic Invocation Interface. The request is send to the Client ORB Core, which transmits it to the Server ORB Core. Here the right Object adapter is invoked which dispatches the request to the object. Again, there is the possibility for the server to choose between a static compiled Skeleton and the Dynamic Skeleton Interface to be used by the object.

### 4.6 GIOP and IIOP

As seen in the previous paragraph there is a client-side ORB and a server-side ORB. GIOP (General Interoperable ORB Protocol) is the protocol that the ORBs use to communicate with each other; this is an abstract definition of what sort of messages can be transferred and how these are formatted. GIOP defines CDR (Common Data Representation) which specifies the binary representation of the different IDL-types in the transmission.

GIOP does not specify a specific transport layer; it only makes assumptions about the transport layer that is used:

- The transport is connection oriented
- The connection is full-duplex
- The connection is symmetric
- The transport is reliable
- The transport provides byte-stream abstraction
- The transport indicates disorderly loss of connection

For TCP/IP as transport layer IIOP (Internet Inter-ORB Protocol) is specified as a concrete implementation of GIOP.
4.7 IOR

If a client wants to invoke a method on an object, it needs to have a reference to the object. CORBA uses the IOR (Interoperable Object Reference) for this use. An IOR contains the following information:

**Repository ID**: This standardized string identifies the object-type.

**Endpoint info**: This is information on where the object is located.

**Object Key**: The Object Key identifies the object at the server. The server only can interpret this key; there is no transparent format of how the Object Key is constructed.

The Endpoint info and the Object key combination can appear multiple times in an IOR; these combinations are called ‘multiplecomponent’ profiles. A profile is specific for each inter-ORB protocol. For IIOP, the Endpoint info in the profile consists of a hostname a TCP/IP port number, the object key is opaque and only the server can interpret it. Figure 4-5 shows the structure of an IOR.

![Figure 4-5: The structure of an IOR](image)

[Henning 99]
5 Wireless CORBA

This chapter describes the Wireless CORBA standard. The first paragraph gives a short introduction, explaining what it does. The second paragraph describes the architecture of Wireless CORBA and explains how the architecture implements the requirements. The succeeding paragraphs describe the protocols used in Wireless CORBA.

5.1 Introduction

The basic idea of Wireless CORBA is that CORBA objects located on a mobile terminal can still be accessed while the terminal can switch to different networks and communication techniques, while keeping the client side unaware of the use of the Wireless CORBA technique.

Some people might think, “Why not using the Naming Service, it is made for this purpose isn’t it?” When using the CORBA Naming Service, the server object can change its reference by binding the new reference to the existing name. When a client does a lookup on the name, it gets the right reference; the problem with the Naming Service is that the client only looks up the reference once. Therefore, when the reference does change, the client does not know and still tries to use the old reference.

The next paragraph describes the architecture of Wireless CORBA, which fulfills the requirements.
5.2 Architecture

Figure 5-1 shows the architecture of Wireless CORBA.

![Figure 5-1: Architecture of Wireless CORBA](image)

The different components of Wireless CORBA shown in Figure 2-1 are hosted on three different domains:

- The Home domain is the domain on the fixed network where the information on how to access the terminal is known.
- The Visited domain is the domain which provide the interface to the terminal, via this domain communication is possible with the terminal.
- The Terminal domain is the domain located on the mobile terminal, it can be accessed via the visited domain and, the location of the current visited domain is known in the Home domain.

Below the components of Wireless CORBA are described:

**Home Location Agent:**
The Home Location Agent keeps track of which terminal is connected to which Access Bridge. The Access Bridge can carryout location updates of the terminal and query the current location of a terminal. A global unique identifier identifies the terminal. How this identifier is build up does not matter, as long it is unique worldwide. One method to produce such an identifier is to concatenate the following data: IP-version, IP-address, local identifier.
Access Bridge:
The Access Bridge provides the network-side access point to the GIOP tunnel, see paragraph 5.4. The GIOP messages are encapsulated according to the GIOP Tunneling Protocol (GTP) and send to the Terminal Bridge. GTP-messages, which the terminal Bridge sends, are decapsulated. The Access Bridge (may) also support handoffs, so the terminal can seamlessly switch from one Access Bridge to another.

Terminal Bridge:
The Terminal Bridge provides the terminal-side access point to the GIOP tunnel, see paragraph 5.4. The GIOP messages that are received are encapsulated and sent to the Access Bridge. The GTP messages received from the Access Bridge are decapsulated.

In order to let a client reach the Wireless CORBA object the normal CORBA IOR is not enough. If a CORBA IOR references to the Home Location Agent, the Home Location Agent knows the Access Bridge, which the object can be reached on, so it let the client know to reach the Wireless CORBA object via the address information of the Access Bridge. However, when the Access Bridge is contacted, how does it know which terminal to contact? Therefore a Mobile IOR is specified, the following paragraph explains this.

[OMG1 2001][BLACK 2001]

5.3 Mobile IOR

The mobile IOR is the reference to a mobile object. Because a non-Wireless CORBA enabled ORB should be able to use the mobile IOR it consists of the normal IIOP profile, which points to the address of the Home Location Agent.

To support Wireless CORBA it contains an extra profile, this is called the ‘Mobile Terminal’ profile. The Mobile Terminal profile contains information to support the Home Location Agent and the Access Bridge in reaching the target object. This information consists of the ID of the terminal, which hosts the object, and the object key of the object on the terminal.

The fist time a GIOP request is sent to the Home Location Agent for an invocation on a Wireless CORBA object. The Home Location Agent needs to find out the terminal-id and the object key of the Wireless CORBA object. When GIOP version 1.2 is used, it uses the NEED_ADDRESSING_MODE-reply to request the Mobile IOR from the client, and then the Mobile Terminal profile is used. However, GIOP version 1.0 and 1.1 does not support this reply. Because of this the Mobile Object Key format is specified, the object key resulting from this format contains the terminal-id and the object key of the Wireless CORBA object. In this situation, the Home Location Agent replies with an LOCATION_FORWARD-message, this contains the address information of the Access Bridge.

[Black 2001]
5.4 GIOP Tunneling Protocol

This paragraph describes the protocol used between the Access Bridge and the Terminal Bridge; it is called the GIOP Tunneling Protocol. The GIOP tunneling protocol (GTP) is an abstract protocol that handles handoffs between different access bridges. At this moment, there are implementations available for UDP, TCP and WAP WDP as underlying transport protocol.

The figure below shows where GTP is located in the communication between a mobile terminal and the client.

![Figure 5-2: GTP architecture](image)

As can be seen in Figure 5-2 the peer does not have to know that the object is located on a mobile terminal. [Black 2001]

Because in this assignment we will only use TCP as underlying transport layer, there will be no description of the UDP and WAP implementation of the GIOP tunneling protocol.

GTP messages consist of a header and the body, the header consists of 8 bytes, the body is of variable length. Figure 5-3 shows the specification of the GTP header.

![Figure 5-3: GTP header](image)

- **seq_no** (unsigned short): The sequence number of the GTP message
- **last_seq_no_received** (unsigned short): The sequence number of the last received GTP message by the sender.
- **gtp_msg_type** (octet): The type of the GTP message, it defines how the body should be interpreted.
- **flags** (octet): Indicates the endianness of the header.
- **content_length** (unsigned short): The length of the GTP message.

The gtp_msg_type-field can indicate the following types of the message:
- **IdleSync**: (No message body) The Terminal Bridge and Access Bridge send the IdleSync message to acknowledge processed GTP messages. Both Bridges also use it in case the idle-timer of the Bridge times out.
• **EstablishTunnelRequest**: Sent by the Terminal Bridge to request a (re-) establishment of a tunnel at an Access Bridge.
• **EstablishTunnelReply**: Sent by an Access Bridge in reply to an EstablishTunnelRequest.
• **ReleaseTunnelRequest**: Sent by the Access Bridge or the Terminal Bridge to request a teardown of the tunnel.
• **ReleaseTunnelReply**: Sent by the Access Bridge or Terminal Bridge to acknowledge the teardown of the tunnel.
• **HandoffTunnelRequest**: The Access Bridge sends this message to the Terminal Bridge in case of a network initiated handoff.
• **HandoffTunnelReplyCompleted**: Sent by the Terminal Bridge in response to a HandoffTunnelRequest.
• **OpenConnectionRequest**: Sent by the Terminal Bridge or the Access Bridge to allocate a connection on the other side of the tunnel.
• **OpenConnectionReply**: Sent by the Terminal Bridge or the Access Bridge in response to an OpenConnectionRequest.
• **CloseConnectionRequest**: Sent by the Terminal Bridge or Access Bridge to close a connection.
• **CloseConnectionReply**: Sent by the Terminal Bridge or Access Bridge in response to a CloseConnectionRequest.
• **ConnectionCloseIndication**: Sent by the Terminal Bridge or Access Bridge to indicate that the connection has been asynchronously closed.
• **GIOPData**: Sent by the Terminal Bridge or Access Bridge, contains an encapsulated GIOP message.
• **GIOPDataReply**: Sent by the Terminal Bridge or Access Bridge to acknowledge a GIOPData message.
• **GTPForward**: Sent by the Terminal Bridge or Access Bridge to forward messages to or from an old Access Bridge.
• **GTPForwardReply**: Sent by a Terminal Bridge or Access Bridge in response to a GTPForward message.
• **Error**: Sent by Terminal Bridge or Access Bridge to handle GTP errors. The tunnel will immediately be shutdown.

[OMG1 2001]

### 5.5 Handoffs

When the mobile terminal loses the connection from one service provider and it connects to another service provider there has to be a protocol to set-up the connection with the same Access Bridge or another Access Bridge. Figure 5-4 illustrated this. This paragraph describes the Handoff protocols that accommodate this requirement of Wireless CORBA.
Assessment of Wireless CORBA in MobiHealth context

Figure 5-4: Handoffs

The initiation of a handoff can come from either the network or the terminal. Figure 5-5 shows the time sequence diagram of a network-initiated handoff. An external program that invokes the start_handoff method on the old Access Bridge starts the network-initiated handoff. When the terminal cannot establish a connection to the old and new Access Bridge at the same time, the terminal will first end the connection with the current Access Bridge and then connect to the new Access Bridge using the ‘access recovery procedure’, explained in the next paragraph.
Figure 5-5: Message sequence diagram of the network initiated handoff

The invocations mentioned in Figure 5-5 and Figure 5-6 ending on 'Notification' are invocations (push) on the Event Service (see paragraph 4.4). When the new Access Bridge invokes the handoff_completed method on the old Access Bridge, the old Access Bridge knows that the new Access Bridge took responsibility for the terminal. Then the old Access Bridge invokes a subscribe_handoff_notice operation on all Access Bridges that are registered to receive handoff notice for that terminal.

Figure 5-6 shows the Terminal Initiated Handoff. It starts when the terminal sends an EstablishTunnelRequest to the new Access Bridge. First, the new tunnel is set-up and then the old tunnel is released. When the new tunnel is set-up, the new Access Bridge invokes an ArrivingTerminalNotification. When the old tunnel is released, the Old Access Bridge will invoke a subscribe_handoff_notice on all Access Bridges that are registered on handoff notices for that terminal.

[OMG1 2001]
Assessment of Wireless CORBA in MobiHealth context

5.6 Access recovery

When the Terminal Bridge discovers that the connection to the Access Bridge is lost, there are two ways to recover:

- Reconnect to the same Access Bridge: Figure 5-7 shows this process.

Figure 5-6: Message sequence diagram of terminal initiated handoff

Figure 5-7: Message sequence diagram of access recovery at the same Access Bridge
• Connect to another Access Bridge: Figure 5-8 shows this process.

Figure 5-8: Message sequence diagram of access recovery at new Access Bridge

The invocations showed in the Figure 5-7 and Figure 5-8 ending on ‘Notification’ are invocations on the Event Service.

5.7 GTP Message forwarding

If a terminal processes a request when the terminal switches to a new Access Bridge, the reply message of that request must still be sent via the old Access Bridge (Required by the GIOP protocol). Because of this requirement, the Access Bridge contains two operations that are used to relay GTP messages between the old Access Bridge and the new Access Bridge. In the example mentioned above the terminal uses the GTPForward message, then the new Access Bridge invokes the gtp_from_terminal operations on the old Access Bridge.

In addition, when the old Access Bridge receives a request for a terminal that is connected to the old Access Bridge it will use the gtp_to_terminal operation on the new Access Bridge, then the new Access Bridge will use the GTPForward message to send the GTP message to the terminal. The old Access Bridge can learn the address of the new Access Bridge by invoking the query_location operation on the Home Location Agent.
6 Redesign of BANip using CORBA

This chapter will describe the redesign of the BAN interconnect protocol for use with CORBA as Lower Level Service. The first paragraph gives an introduction of the new designed protocol. The succeeding paragraph describes the internal architecture of the protocol. After the interfaces, which are needed by CORBA, are defined in paragraph 6.3, paragraph 6.4 the use of Wireless CORBA in the new protocol will be explained. The modeling scheme used in this chapter comes from the lecture notes [Visser 2002] of the lecture ‘The Design of Telematics Systems’ at the University of Twente.

6.1 Introduction

The HTTP-based BAN interconnect protocol (see paragraph 2.4 and chapter 3) uses PDUs to communicate between the two Protocol Entities. The goal of this assignment is to use Wireless CORBA; because CORBA is an object-based middleware platform, a new protocol must be designed with the same service as the HTTP-based BAN interconnect protocol but with CORBA as Lower Level Service, Figure 6-1 illustrates this.

![Figure 6-1: CORBA-based BAN interconnect protocol](image)

The scope in this assignment is on using Wireless CORBA for the management-information; therefore, the sending of sensor-data is left out of this design. It would probably much better to use raw TCP-sockets for this communication. Because of this, the Service Primitives of SAP 1 and SAP 2 are not exactly the same as defined in chapter 3. The Service Primitives ‘data_ind’ and ‘sent_data_req’ are left out.
The Service Primitives of SAP 3 and SAP 4 are defined by the object invocations made by both Protocol Entities; paragraph 6.3 describes these.

6.2 Internal architecture

This paragraph describes the internal architecture of the CORBA-based BAN interconnect protocol.

Figure 6-2 shows the different Protocol Elements which are communicating with each other.

MBU:
The MBU component represents the ‘real life’ MBU. It can register itself at the fixed Protocol Entity and keep a connection with the surrogate instance. The MBU also takes care of the subscription of the surrogate on sensor data.

Registrar:
The Registrar allows an MBU to register itself at the fixed Protocol Entity. For each MBU that registers itself at the Registrar, it will start an instance of the Surrogate.

Surrogate:
The Surrogate component represents the mobile MBU on the fixed side. Via this Surrogate, the rest of the system on the fixed side can do calls on the MBU.
As can be seen in Figure 6-2 SAP 1 is split in two SAPs, The Service Primitives contained by these SAPs are the same as in the HTTP-based BAN interconnect protocol as described in paragraph 3.2.2.

Figure 6-3 illustrates how the instances of the elements communicate as described above. As can be seen, the MBU registers itself at the fixed Protocol Entity by sending registration information to the Registrar, and then the Surrogate and MBU communicate by sending management information. This management information can consist of the following messages:

**Keep-alive-message**: The MBU sends this message to the Surrogate in order to let the Surrogate know that the connection still exists.

**Subscription-request**: The Surrogate sends this message to the MBU in order to subscribe on sensor data.

**Subscription-activate**: Sent from the Surrogate to the MBU in order to activate an existing subscription, as parameter the Surrogate passes an address where the MBU must send the data.

**Subscription-deactivate**: The Surrogate sends this message to the MBU in order to deactivate a subscription.

![Figure 6-3: Communication between the instances of the components of the CORBA-based BAN interconnect protocol](image-url)
As said before CORBA is an object based middleware platform, to support the invocation of operations on the objects, the interface of these objects must be defined. The next paragraph describes the interfaces for the CORBA-based BAN interconnect protocol.

### 6.3 Interfaces

This paragraph specifies the interfaces of the elements of the CORBA-based BAN interconnect protocol as described in the previous paragraph. The specification is written in IDL (see paragraph 4.2).

Below the IDL specification and the description are given:

**Registrar:**
The registrar needs a method to let MBUs register itself, this method is called ‘registerMBU’. As parameter, an MBU-object must be given in order to create a Surrogate that has the reference to this MBU-object. A reference to that Surrogate is then returned.

```
interface Registrar {
    Surrogate registerMBU(in BAN::MBU mbu);
};
```

**Surrogate:**
The Surrogate needs a method to let keep-alive messages be sent, this method is called ‘keepAlive’. It does not need any parameters.

```
interface Surrogate {
    oneway void keepAlive();
};
```

**MBU:**
The MBU needs the following methods:

- requestSubscription: Is used to request a subscription on data of a certain set of sensors. The ssid-parameter is to identify the set of data it wants. The transcodingChain parameter is to set the transcoding chain that is used.
- activateSubscription: Is used to activate an existing subscription. The serviceUserEndpoint is the address the sensor data must be sent to.
- deactivateSubscription: Is used to deactivate an existing subscription.

```
interface MBU {
    oneway void requestSubscription(in string ssid,
                                    in string transcodingChain);
    oneway void activateSubscription(in string ssid,
                                       in string serviceUserEndpoint, in string transcodingChain);
    oneway void deactivateSubscription(in string ssid);
};
```

Figure 6-4 shows in a Time Sequence Diagram how the interfaces are used.
6.4 Using Wireless CORBA

This paragraph describes what happens when the protocol described in the paragraphs above uses the Wireless CORBA components. The description is split in three steps: Wireless CORBA setup, BAN setup and registration, Subscription and keep-alive.

**Wireless CORBA setup:**
To let the protocol described in the previous paragraphs use the Wireless CORBA technology, first the Wireless CORBA services have to be initialized. When the Terminal Bridge initializes, it has to set up a GTP connection with the Access Bridge. Therefore, Terminal Bridge sends an EstablishTunnelRequest to the GTP port of the Access Bridge. The Access Bridge then invokes the update_location operation on the Home Location Agent with the terminalId and a reference to the Access Bridge as parameters.
Then when the Home Location Agent knows if the terminal is accessible via the Access Bridge the Access Bridge answers with an EstablishTunnelReply. Figure 6-5 illustrates this step.

**Figure 6-5: Sequence diagram of Wireless CORBA setup**

**BAN Setup and Registration:**
The Wireless CORBA enabled ORB creates the reference to the MBU object; this means that it contains a ‘Mobile Terminal’ profile as described in paragraph 5.3. When the object is created, the profile is registered at the terminal, this is done by invoking the register_profile operation on the terminal, and this method returns the reference to the Home Location Agent. When this is done, the terminal knows how to reach the object to invoke operations on it.
In order to register the MBU at the Registrar the MBU wants to invoke the method 'registerMBU' on the Registrar object with the MBU as parameter, this GIOP-request is not send directly to the Registrar. The request is routed to the Terminal Bridge, which opens a connection to the Registrar object via the Access Bridge. Then the GIOP-request is encapsulated into a GIOPData GTP-message and send over the GIOP tunnel to the Access Bridge. The Access Bridge sends the request to the Registrar and sends Surrogate reference over the GIOP tunnel back to the Terminal Bridge, which then replies the reference to the Surrogate.

**Subscription:**
The normal IIOP-profile of the MBU-IOR points to the Home Location Agent. When the client sends the GIOP-request to the Home Location Agent, it returns a
LOCATION_FORWARD-reply (Only in IIOP version 1.0/1.1, version 1.2 uses a NEED_ADDRESSING_MODE-reply, as described in paragraph 5.3) with the IIOP-profile of the Access Bridge.

The Surrogate then forwards the request to the Access Bridge. The Access Bridge opens a connection to the Terminal Bridge over the GIOP tunnel. Then the Access Bridge encapsulates the GIOP message in a GTP-GIOPData message and sends this to the Terminal Bridge.

The Terminal Bridge knows where the MBU object is located and invokes the method on this object. Because it is a one-way method, the MBU does not send a return-value back to the Surrogate.

Figure 6-7 describes the invocation of the requestSubscription. The other methods that can be invoked on the MBU object are invoked likewise.

Figure 6-7: Sequence diagram of subscription

**Keep Alive:**

The MBU sends keep-alive messages at a periodic interval. The GIOP-request is routed via the GIOP tunnel. Therefore first a connections is set up (OpenConnectingRequest and OpenConnectionReply), then the GIOP-message is encapsulated in a GIOPData GTP-message. The ‘oneway’ keyword in de IDL implicates that the Surrogate will not return a value. Figure 6-8 illustrates this step.
Assessment of Wireless CORBA in MobiHealth context

Figure 6-8: Sequence diagram of Keep-alive invocation

GIOP

GTP

IdleSync

IdleSync

IdleSync
7 Analysis and comparison

This chapter analyses the communication between the fixed side, the mobile side, in order to come to a comparison between the Wireless CORBA based BANip, and the HTTP based BANip.

Many fields in the transmitted messages are of variable size. This size of the messages can be only easily determined by monitoring the data that is sent in a practical environment (Only practical determined values will be calculated). Therefore, a test-environment has been set up; this environment is described in the first paragraph. In the succeeding two paragraphs, the messages that are sent over the wireless link are analyzed.
The fourth paragraph describes how much data is actually used for the different stages when comparing it to the HTTP-based BAN interconnect protocol.

7.1 Test-environment

This paragraph describes the test environment used to test the design.
The most important aspect of the system is that it must support a Wireless CORBA implementation. Because there is only one Wireless CORBA implementation yet (using C++), the Mobile-side software had to be implemented in C++ (The fixed side does not have to support Wireless CORBA, as described in chapter 5).
The current software used for the MobiHealth system is completely written in Java, this means that on the Mobile-side the design cannot be integrated with the existing system, because the fixed-side of the software is also written in Java this can be integrated.
The paragraphs below describe the test-environment for both programming languages, Java and C++.
The IDL-file that is used to define the interfaces in the test-environment is included in Appendix B.1.

7.1.1 Java

The Java SDK 1.4.2 is used.
To support a persistent object reference for the Registrar object the SUN CORBA implementation cannot be used. JacORB is used in order to meet this requirement.
Preferably GIOP version 1.2 is used (MIWCO is build for this version, but does work with GIOP version 1.0), this version does support the ‘NEEDS_ADDRESSING_MODE’ [CORBA specification], which is used by the Home Location Agent replies to the client and needs to tell the client to connect to another address. Unfortunately, this ‘NEEDS_ADDRESSING_MODE’ feature is not implemented in the JacORB implementation.
Because there is no other CORBA implementation available for Java that supports the needed features, a workaround is used. This workaround consists of using GIOP version 1.0 instead of 1.2. In order to do this, JacORB version 1.3.30 (does not support GIOP 1.2 yet) is used.
Assessment of Wireless CORBA in MobiHealth context

7.1.2 C++

MIWCO (MIWCO Is Wireless CORBA) is the only implementation of the Wireless CORBA protocol; it is an extension of the MICO ORB. MIWCO version 2.3p19 is used in the test-environment (latest development snapshot at this moment).

The MICO configuration-file (".micorc") used by the Wireless CORBA components that run on the fixed network contains the following values:

-ORBGIOPVersion 1.0 //Using version 1.0, see previous paragraph
-ORBIIOPVersion 1.0 //Using version 1.0, see previous paragraph
-ORBInitRef NameService=corbaloc::1.0@130.89.145.59:12001/NameService
-ORBDebug All //Print all debug information available

The MICO configuration files used by the Terminal Bridge, which is running on another machine contains the following different line:

-ORBInitRef NameService=corbaloc::1.0@130.89.160.30:12001/NameService

The application uses a different configuration file, the following values are in this configuration file (".miwcoapprc"):

-ORBGIOPVersion 1.2
-ORBIIOPVersion 1.2
-ORBInitRef NameService=corbaloc::1.2@130.89.160.30:12001/NameService
-ORBDebug All //Print all debug information available
-ORBTerminalId %04%82%59%91%3b%08terminal //The id of the terminal
-ORBMTBAAddr inet:127.0.0.1:12003 //The address where the TB is listening
-ORBInitRef MobileTerminalBridge=<IOR> //The IOR of the Terminal Bridge

The GIOP/IIOP-version used by the Mobile application is 1.2. Although the other components use version 1.0, the mobile application needs to send GIOP 1.2 with reference-based addressing to allow the Terminal Bridge to be able to open a connection to other CORBA-objects.

[Kangasharju2 2002]

7.2 Starting the test-software

The software has been tested as a distributed system. It has been distributes over two machines, one for the software meant for the mobile device (IP-address: 130.89.160.30) and one for the software running on the fixed network (IP-address: 130.89.145.59).

The different Wireless CORBA components are started as follows [Kangasharju1 2002]:

**Naming Service:**
This service was started on both systems to allow the Wireless components on both systems register themselves at the local Naming Service.

The startup-commands:

```bash
nsd -ORBIIOPAddr inet:130.89.145.59:12001
```
and

```bash
nsd -ORBIIOPAddr inet:130.89.160.30:12001
```
‘-ORBIIOPAddr’-parameter specifies the address where the Naming Service Daemon must listen to.

**Event Service:**
This service has to be started at both systems.
The startup-commands:

```
  eventd -ORBIIOPAddr inet:130.89.145.59:27566
  eventd -ORBIIOPAddr inet:130.89.160.30:27565
```

‘**ORBIIOPAddr**’: Specifies the address where the Event Daemon must listen.

**Home Location Agent:**
The Home Location Agent has to be started on the machine that represents the fixed network (see paragraph 5.2).
Startup-command:

```
  hla -ORBIIOPAddr inet:130.89.145.59:27360 -POAImplName MobilitySupport -WATMTerminalPrefix %82%59%91%3b
```

‘**ORBIIOPAddr**’: Specifies the address where the Home Location Agent must listen.
‘**WATMTerminalPrefix**’: A prefix for the terminals registered at this Home Location Agent used to get unique terminal identifiers (see the Wireless CORBA specification, section 4.4)
‘**POAImplName**’: Used for convenience.

**Access Bridge:**
The Access Bridge is started on the machine that represents the fixed network (see paragraph 5.2).
Startup-command:

```
  ab -WATMGIOAddr inet:130.89.145.59:41513 -WATMGPAddr inet:130.89.145.59:21100 -POAImplName MobilitySupport -ORBNoIIOPServer >ab.ref
```

‘**WATMGIOAddr**’: This is de GIOP address where the Access Bridge is visible on the network side.
‘**WATMGPAddr**’: This is the address where the Access Bridge listens for incoming GTP connections from Terminal Bridges.
‘**ORBNoIIOPServer**’: The Access Bridge uses a custom IIOPServer; this makes sure MICO does not start one accidentally.

The IOR of the Access Bridge is stored in the file ‘ab.ref’, it is not needed further.

**Terminal Bridge:**
The Terminal Bridge is started on the machine that represents the mobile device (see paragraph 5.2).

```
  tb -WATMGIOAddr inet:130.89.160.30:12003 -WATMGPAddr inet:130.89.145.59:21100 -POAImplName MobilitySupport -ORBTerminalId
```
Assessment of Wireless CORBA in MobiHealth context

%04%82%59%91%3b%08terminal -ORBNoIIOPServer -ORBInitRef
HomeLocationAgent=\<home.ref\> >tb.ref

‘WATMGIOPAddr’: The GIOP-address of the Terminal Bridge where it can be accessed from the terminal side.
‘WATMGTPAddr’: The GTP-address of the initial Access Bridge.
‘ORBTerminalId’: The identifier of the Terminal Bridge’s terminal.
‘ORBNoIIOPServer’: The same as with the Access Bridge.
‘HomeLocationAgent=’: The reference of its Home Location Agent, it is set via ‘-ORBInitRef’

The IOR of the Terminal Bridge is saved in the file ‘tb.ref’, the Application on the mobile device needs this reference.

When all Wireless CORBA components run, the applications itself can be started. There are two applications: The IIOPInterconnectServer and the BANServer. The IIOPInterconnectServer is the server that hosts the Registrar-objects, so it is written in Java. The BANServer hosts the MBU-object, so it is written in C++. The source-code of both applications and the IDL-file that is used is included in appendices B, C and D.

The first application to start is the IIOPInterconnectServer, there are no parameters required for this application. It is started at the machine that represents the fixed network.
**Startup-command:**
```bash
start
```

The IIOPInterconnectServer prints the IOR of the Registrar object to the console; this string is needed to start the BANServer. The BANServer runs on the machine that represents the mobile device.
**Startup-command:**
```bash
BANServer <RegistrarIOR> -ORBIIOPAddr inet:130.89.160.30:3103 -ORBConfFile ~/.miwcoapprc
```

‘ORBIIOPAddr’: The Address where the server listens for IIOPMessages
‘ORBTerminalId’: The terminal identifier of the Mobile Terminal
‘ORBMTBAddr’: The GIOP-address of the Terminal Bridge. All GIOP messages will be redirected to this address.
‘MobileTerminalBridge=’: The reference to the MobileTerminalBridge, it is set via ‘-ORBInitRef’

Both applications show status information, which indicates that they the MBU-object is registered at the Registrar. The BANServer then starts to send keep-alive messages at a fixed rate. At the console of the IIOPInterconnect server the strings ‘request’, ‘activate’ or ‘deactivate’ can be typed to invoke the methods ‘requestSubscription’, ‘activateSubscription’ or ‘deactivateSubscription’ on the remote MBU-object. When one of these methods is invoked this is displayed at both sides.
The test-application sends dummy data as parameters for these methods:
**serviceUserEndpoint**: It is supposed that this is a string with which contains an IP-address and a port number, something like “aaa.bbb.ccc.ddd:eeee”. This parameter contains 20 bytes of data.

**Ssid**: The test-data contained in the source-code of the MobiHealth system shows that that the SensorSetId is left empty. Therefore, this parameter contains 0 bytes of data.

**transcodingChain**: The test-data contained in the source-code of the MobiHealth system shows that de transcodingChain contains different values, from zero to 25 characters. In this test-environment, it is assumed this property takes 12 bytes.

### 7.3 Analysis of the GTP messages

When a method is invoked on an object located on the mobile side the corresponding GIOP messages are sent trough the GIOP tunnel.

The GTP messages sent over the connection are analyzed in this chapter. For this purpose the debug-information generated in the test-environment is used (see paragraph 7.1)

Each GTP message consists of a header of eight bytes and a message-body (see paragraph 5.4). This paragraph analyzes the different GTP messages that are used in the test-environment.

**EstablishTunnelRequest**: The structure of the body of this message is specified below in IDL:

```idl
union EstablishTunnelRequestBody switch (RequestType) {
  case InitialRequest: InitialRequestBody initial_request_body;
  case RecoveryRequest: RecoveryRequestBody recovery_request_body;
};

typedef short RequestType;
const short InitialRequest = 0;
const short RecoveryRequest = 1;
struct InitialRequestBody {
  MobileTerminal::TerminalId terminal_id;
  MobileTerminal::HomeLocationAgent home_location_agent_reference;
  unsigned long time_to_live_request;
};
struct RecoveryRequestBody {
  MobileTerminal::TerminalId terminal_id;
  MobileTerminal::HomeLocationAgent home_location_agent_reference;
  struct LastAccessBridgeInfo {
    MobileTerminal::AccessBridge access_bridge_reference;
    unsigned long time_to_live_request;
    unsigned short last_seqno_received;
  } last_access_bridge_info;
  unsigned long time_to_live_request;
};

module MobileTerminal {
  typedef sequence<octet> TerminalId;
};
```
In the test-environment, this message uses 208 bytes including the GTP-header.

**EstablishTunnelReply:**
The structure of the body of this message is specified below in IDL:

```cpp
union EstablishTunnelReplyBody switch (ReplyType) {
    case InitialReply: InitialReplyBody initial_reply_body;
    case RecoveryReply: RecoveryReplyBody recovery_reply_body;
};
typedef short ReplyType;
const short InitialReply = 0;
const short RecoveryReply = 1;
enum AccessStatus {
    ACCESS_ACCEPT,
    ACCESS_ACCEPT_RECOVERY,
    ACCESS_ACCEPT_HANDBOFF,
    ACCESS_ACCEPT_LOCAL,
    ACCESS_REJECT_LOCATION_UPDATE_FAILURE,
    ACCESS_REJECT_ACCESS_DENIED
};
struct InitialReplyBody {
    AccessStatus status;
    MobileTerminal::AccessBridge access_bridge_reference;
    unsigned long time_to_live_reply;
};
struct RecoveryReplyBody {
    AccessStatus status;
    MobileTerminal::AccessBridge access_bridge_reference;
    struct OldAccessBridgeInfo {
        unsigned long time_to_live_reply;
        unsigned short last_seqno_received;
    } old_access_bridge_info;
    unsigned long time_to_live_reply;
};
```

[OMG1 2001]

In the test-environment, this message uses 180 bytes including the GTP-header.

**OpenConnectionRequest:**
The structure of the body of this message is specified below in IDL:

```cpp
struct OpenConnectionRequestBody {
    Object target_object_reference;
    unsigned long open_connection_request_id;
    unsigned long timeout;
};
```

[OMG1 2001]

In the test-environment, this message 76 bytes including the GTP-header.
Assessment of Wireless CORBA in MobiHealth context

**OpenConnectionReply:**
The structure of the body of this message is specified below in IDL:

```c
struct OpenConnectionReplyBody {
    unsigned long open_connection_request_id;
    OpenConnectionStatus status;
    unsigned long connection_id; // 0xFFFFFFFF indicates failure
};
enum OpenConnectionStatus {
    OPEN_SUCCESS,
    OPEN_FAILED_UNREACHABLE_TARGET,
    OPEN_FAILED_OUT_OF_RESOURCES,
    OPEN_FAILED_TIMEOUT,
    OPEN_FAILED_UNKNOWN_REASON
};
```

[OMG1 2001]

In the test-environment, this message 20 bytes including the GTP-header.

**GIOPData:**
The structure of the body of this message is specified below in IDL:

```c
struct GIOPDataBody {
    unsigned long connection_id;
    unsigned long giop_message_id;
    MobileTerminal::GIOPEncapsulation giop_message;
};
```

[OMG1 2001]

When the requestSubscription-method is called in the test-environment this results in a message GIOPData-message of 173 bytes including the GTP-header.
The GIOPData message resulting from the invocation of the activateSubscription method consists of 205 bytes.
The GIOPData message resulting from the invocation of the deactivateSubscription method consists of 157 bytes.
All these methods are one-way methods, therefore no GIOPData-message is sent as reply.

**IdleSync:**
The IdleSync message does not have a body. Therefore, the data sent, only consists of the GTP-header (8 bytes of data).
[OMG1 2001]

**CloseConnectionRequest:**
The IDL below specifies the structure of the body of this message:

```c
struct CloseConnectionRequestBody {
    unsigned long connection_id; // 0xFFFFFFFF denotes all connections for sender
};
```

[OMG1 2001]

This message has a size of 12 bytes including the GTP-header.
Assessment of Wireless CORBA in MobiHealth context

CloseConnectionReply:
The structure of the body of this message is specified below in IDL:

```c
struct CloseConnectionReplyBody {
    unsigned long connection_id; // same as in request
    CloseConnectionStatus status;
};
enum CloseConnectionStatus {
    CLOSE_SUCCESS,
    CLOSE_FAILED_INVALID_CONNECTION_ID,
    CLOSE_FAILED_UNKNOWN_REASON
};
```

[OMG1 2001]

In the test-environment, this message 16 bytes including the GTP-header.

### 7.4 Usage of wireless connection

This paragraph describes how much data is totally sent over wireless connection when each Service Primitive is used by the Service Users. This result is compared with the data needed in the HTTP-based BAN interconnect protocol.

**Setup of Terminal Bridge:**
When the Terminal Bridge starts, it set up a GTP connection with the Access Bridge. The Terminal Bridge sends an EstablishTunnelRequest (208 bytes) to the Access Bridge, which returns an EstablishTunnelReply (180 bytes). After this, the Access Bridge sends an IdleSync message, which the Terminal Bridge returns with also an IdleSync message (2 x 8 = 16 bytes).
The setup of the Terminal Bridge takes 404 bytes (180 + 208 + 16).
The HTTP-based BAN interconnect protocol does not have any setup-overhead, so no comparison can be made. The 404 bytes needed by this action is pure overhead.

**Registration of MBU:**
When the MBU-object invokes the registerMBU method on the Registrar-object, the bridges transfer the following messages over the wireless connection (See paragraph 6.4):
- OpenConnectionRequest: uses 244 bytes.
- OpenConnectionReply: uses 20 bytes.
- GIOPData: uses 352 bytes. (Terminal Bridge to Access Bridge)
- GIOPData: uses 248 bytes. (Access Bridge to Terminal Bridge)
- IdleSync: uses 8 bytes.

This result in a total amount of bytes transferred over the wireless connection of 244 + 20 + 352 + 248 + 8 = 872 bytes.
In the HTTP-based BAN interconnect protocol the HttpRegistrationPDU takes 297 bytes, the HttpResponsePDU takes 116 bytes. The total for the HTTP-Based protocol comes at 297 + 116 = 423 bytes (See paragraph 3.5).
Keep-alive messages:
When the MBU-object invokes the keepAlive method on the Surrogate-object, the bridges transfer the following messages over the wireless connection (See paragraph 6.4):

- OpenConnectionRequest: uses 228 bytes
- OpenConnectionReply: uses 20 bytes
- GIOPData: uses 84 bytes
- Three times an IdleSync: uses $3 \times 8 = 24$ bytes.

The total amount of bytes used over the wireless connection: $228 + 20 + 84 + 24 = 356$ bytes.

In the HTTP-based BAN interconnect protocol, the KeepAlivePDU takes 200 bytes, if an empty OK-response (77 bytes) is sent as reply; one keepAlive procedure takes 277 bytes.

Invocation of a method on MBU:
When the Surrogate invokes one of the methods of the MBU, the bridges send the following messages over the wireless connection (see paragraph 6.4):

- OpenConnectionRequest: uses 76 bytes.
- OpenConnectionReply: uses 20 bytes.
- GIOPData: the data-usage is different for each method:
  - requestSubscription: uses 173 bytes.
  - activateSubscription: uses 205 bytes.
  - deactivateSubscription: uses 157 bytes.
- IdleSync: uses 8 bytes.
- IdleSync: uses 8 bytes.
- IdleSync: uses 8 bytes.

This result following amount of data that the bridges transfer over the wireless connection, separated per method:

- requestSubscription: $76 + 20 + 173 + 8 + 8 + 8 = 293$ bytes.
- activateSubscription: $76 + 20 + 205 + 8 + 8 + 8 = 325$ bytes.
- deactivateSubscription: $76 + 20 + 157 + 8 + 8 + 8 = 277$ bytes.

In The HTTP-based BAN interconnect protocol these messages are sent as reply to a keep-alive message, so there will be no reply message sent over the wireless link for these messages.

As can be seen in paragraph 3.5, the HttpRequestSubscriptionPDU HTTP-message takes 121 bytes, the HttpActivateSubscriptionPDU HTTP-messages takes 140 bytes and the HttpDeactivateSubscription HTTP-message takes 130 bytes.
8 Conclusion

This chapter explains the conclusions that resulted from the analysis and design done in this assignment. In addition, some issues, which can be researched further, are mentioned and described.

The main question of this assignment was if the BAN interconnect protocol is more efficient when using Wireless CORBA instead of HTTP as Lower Level Protocol. Paragraph 7.4 gives answer to this question; it compares the use of the wireless connection when using the Wireless CORBA against the HTTP based BANip. When setting up the Wireless CORBA components there is an overhead of 404 bytes, while there is no setup needed for components, but this set-up is done only once (if no handoffs are made).

The registration of the MBU at the Surrogate host takes 872 bytes of data that the bridges send over the wireless connection, while it only takes 423 bytes when using HTTP. In principle the registration process is also done only once in a session. The subscription and activation needed to let the MBU send sensor-data takes 706 (369 + 337) bytes, while this takes only 261 (121 + 140) bytes when using HTTP. The deactivation of a subscription takes 321 bytes while this takes 130 bytes when using HTTP. In a typical realistic situation, this process occurs only ones in a session.

When registered the MBU starts sending keep-alive messages that use 356 bytes over the wireless link, while this takes 277 bytes when using HTTP. All messages the Wireless CORBA based BANip sends are larger than the corresponding messages send by the HTTP based BANip, which thus uses the wireless connection more efficient in all situations.

Possibly an improvement can be made to the Wireless CORBA based BANip so that the keep-alive messages do not have to be sent over the wireless connection. The Access Bridges detect when the tunnel collapse. When this happens, it invokes the deregister_terminal method on the Home Location Agent; this is also logged when the debug-functionality is turned on. A new process might monitor these logs and while a terminal is connected, invoke the keep-alive method on the Surrogate. Further research is needed to find out if this is possible and how this can be designed best.

The HTTP-based BANip uses the keep-alive message to let the Surrogate Host give subscription commands to the MBU, the protocol uses this piggybacking-mechanism because the MBU has got a private IP-address on the GPRS-network so the MBU is not directly reachable from the Internet. Because of this, the Surrogate Host can only contact the MBU once every 30 seconds. Instantaneous response from the MBU to a request from the Surrogate Host is not possible. When using the Wireless CORBA based BANip the Surrogate host can reach the MBU immediately through the GIOP tunnel between the Access Bridge and Terminal Bridge. This might be a requirement for certain applications; Wireless CORBA can be very useful for these applications.
If wireless CORBA is not necessary for an application, (No fast direct access to MBU needed or no support for handoffs needed) and there is no need for hand-off support, ‘normal’ CORBA can be used as Lower Level Service for the BANip. This requires the use of the piggybacking-mechanism to let the Surrogate Host send commands to the MBU, as described above. To accommodate further research a possible layout of the interfaces (defined in IDL) is included in Appendix B.2.
9 References

[Henning 99] Henning, M, Vinoski,S; Advanced CORBA Programming with C++; 1999


[Knoppel 2003] Performance Improvement of a BAN interconnect protocol


[Kangasharju1 2002] MIWCO – Wireless CORBA extensions for MICO

[Kangasharju2 2002] Implementing the Wireless CORBA Specification
## Appendix A – Time schedule

<table>
<thead>
<tr>
<th>Task/Weeknr</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set-up a MICO CORBA C++ environment</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Learning basics of C++</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Using MICO</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set-up a Wireless CORBA environment</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Set-up of MIWCO</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Testing MIWCO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Compare efficiency of the standard CORBA protocol with the wireless CORBA protocol</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Make and implement MobiHealth interconnect protocol (BAN interconnect protocol) using CORBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Research overhead introduced by current protocol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Research overhead introduced by Wireless CORBA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Compare the results in theory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Compare the results in practice.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Build a demonstrator that visualizes the differences in efficiency.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Implement a demonstrator that visualizes the practical results obtained by the previous task.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Report and presentation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Combine the different documents as one report and deliver this report</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Prepare a presentation.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Deliver presentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B - Interface definitions

The interface definitions are written in IDL.

B.1 IIOPInterconnect.idl

```idl
module BAN {
    interface MBU {
        oneway void requestSubscription(in string ssid,
                                           in string transcodingChain);
        oneway void activateSubscription(in string ssid,
                                           in string serviceUserEndpoint,
                                           in string transcodingChain);
        oneway void deactivateSubscription(in string ssid);
    };
}

module IIOPInterconnect {
    interface Surrogate {
        oneway void keepAlive();
    };
    interface Registrar {
        Surrogate registerMBU(in BAN::MBU mbu);
    };
};
```
B.2 IDL for alternative Wireless CORBA BANip

```c
enum CommandType {request, activate, deactivate};

struct RequestSubscription {
    string ssid;
    string transcodingChain;
};
struct ActivateSubscription {
    string ssid;
    string serviceUserEndpoint;
    string transcodingChain;
};
struct DeactivateSubscription {
    string ssid;
};

union Command switch (CommandType) {
    case request:
        RequestSubscription request;
    case activate:
        ActivateSubscription activate;
    case deactivate:
        DeactivateSubscription deactivate;
};

module IIOPInterconnect {
    interface Surrogate {
        Command keepAlive();
    }
    interface Registrar {
        Surrogate registerMBU(in string initData);
    }
};
```
Appendix C – C++ code

In this appendix the C++ code of the mobile side is shown

C.1 BANServer.h

```cpp
#include "IIOPInterconnect.h"

namespace BAN {
    class BANServer;
    class MBUImpl;
}

namespace BAN {
    class BANServer {
        CORBA::ORB_var orb;
        IIOPInterconnect::Registrar * registrar;
        PortableServer::POA_var rootPOA;

        public:
            BANServer(CORBA::ORB_var, IIOPInterconnect::Registrar *);
            void start();
    };

class MBUImpl : public virtual POA_BAN::MBU {
    CORBA::ORB_var * orb;
    IIOPInterconnect::Surrogate * surrogate;

    public:
        MBUImpl(CORBA::ORB_var);
        void doRegister(IIOPInterconnect::Registrar *) throw(CORBA::SystemException);
        void requestSubscription(const char *, const char *);
        void activateSubscription(const char *, const char *, const char *);
        void deactivateSubscription(const char *);
    }
}

void * sendKeepAlives(void *);
```

C.2 BANServer.cc

```cpp
#include <pthread.h>
#include "BANServer.h"
#include "IIOPInterconnect.h"

namespace BAN {
    BANServer::BANServer(CORBA::ORB_var tmp_orb,
        IIOPInterconnect::Registrar * registrar) {
        orb = tmp_orb;
        this->registrar = registrar;

        //RootPOA ophalen
```
Assessment of Wireless CORBA in MobiHealth context

```c++
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
rootPOA = PortableServer::POA::_narrow(obj);

// Activate POA manager
PortableServer::POAManager_var mgr = rootPOA->the_POAManager();
mgr->activate();

void BANServer::start() {
    // Starten van MBU-servant
    MBUImpl mbuServant(&orb);
    MBU_var mbuObj = mbuServant._this();
    mbuServant.doRegister(this->registrar);
    // Accepteren van requests
    orb->run();
}

void MBUImpl::MBUImpl(CORBA::ORB_var * orb) {
    this->orb = orb;
}

void MBUImpl::requestSubscription(const char * ssid,
                                  const char * transcodingChain) {
    cout << "MBUImpl->requestSubscription() aangeroepen" << endl;
}

void MBUImpl::activateSubscription(const char * ssid,
                                   const char * serviceUserEndpoint,
                                   const char * transcodingChain) {
    cout << "MBUImpl->activateSubscription() aangeroepen" << endl;
}

void MBUImpl::deactivateSubscription(const char * ssid) {
    cout << "MBUImpl->deactivateSubscription() aangeroepen" << endl;
}

void MBUImpl::doRegister(IIOPInterconnect::Registrar * registrar)
    throw(CORBA::SystemException) {
    MBU_var mbuObj = this->_this();
    this->surrogate = registrar->registerMBU(mbuObj);
    cout << "MBUImpl->doRegister(): Geregistreerd bij registrar, \n    nu KeepAlives instellen" << endl;
    pthread_t thread;
    int code = pthread_create(&thread, NULL, sendKeepAlives, this->surrogate);
    cout << "MBUImpl->doRegister(): \n    ...klaar om subscribes te ontvangen" << endl;
}

void * sendKeepAlives(void * arg) {
    cout << "sendKeepAlives aangeroepen!!" << endl;
    IIOPInterconnect::Surrogate * surrogate = (IIOPInterconnect::Surrogate *) arg;
    while (1) {
        timespec req;
        req.tv_sec = 5;
        req.tv_nsec = 0;
        timespec * reqpt = &req;
```
Assessment of Wireless CORBA in MobiHealth context

```cpp
surrogate->keepAlive();
cout << "KeepAlive verstuurde... " << endl;
nanosleep(reqpt, NULL);
}
}

int main(int argc, char * argv[]) {
  try {
    // Check arguments
    if (argc < 2) {
      cerr << "Usage: client IOR_string" << endl;
      throw 0;
    }

    // Initialize orb
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Destringify argv[1]
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    if (CORBA::is_nil(obj)) {
      cerr << "Nil Registrator reference" << endl;
      throw 0;
    }

    // Narrow
    IIOPInterconnect::Registrar * registrar =
      IIOPInterconnect::Registrar::_narrow(obj);
    if (CORBA::is_nil(registrar)) {
      cerr << "Argument is not a Registrator reference" << endl;
      throw 0;
    }

    BAN::BANServer server (orb, registrar);
    server.start();
  } catch (CORBA::COMM_FAILURE & e) {
      cerr << "Exception: COMM_FAILURE" << endl;
  } catch (const CORBA::Exception & ) {
      cerr << "Uncaught CORBA exception" << endl;
  }
} 59
```
Appendix D – Java code

In this appendix the java code of the fixed side is shown

D.1 IIOPInterconnectServer.java

```java
package IIOPInterconnect;

import org.omg.PortableServer.*;
import org.omg.PortableServer.*;
import org.omg.CORBA.*;
import java.io.*;
import java.util.Properties;

public class IIOPInterconnectServer {
    String[] args;
    org.omg.CORBA.ORB orb;
    POA rootPoa;
    POA poa;

    public IIOPInterconnectServer(String[] args) {
        this.args = args;

        Properties properties = System.getProperties();
        System.setProperty("jacorb.implname", "test");

        // Step 1: Instantiate the ORB
        orb = org.omg.CORBA.ORB.init(args, null);
        try {
            //Get root-POA
            rootPoa = org.omg.PortableServer.POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
            rootPoa.the_POAManager().activate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        IIOPInterconnectServer IIOPInterconnectServer1 =
            new IIOPInterconnectServer(args);
        IIOPInterconnectServer1.startServer();
    }

    public void startServer() {
        try {
            //create a POA policy for the new persistent POA
            // first create a POA policy for the new persistent POA,
            // here the Lifespanpolicy set to persistent

            Policy[] policy = new Policy[2];
            policy[0] = rootPoa.create_lifespan_policy(
                LifespanPolicyValue.PERSISTENT);
            policy[1] = rootPoa.create_id_assignment_policy(
                IdAssignmentPolicyValue.USER_ID);
```

Assessment of Wireless CORBA in MobiHealth context

// now create the persistent POA with the POA policy.
// It is a daughter if the rootpoa
// The rootpoa is always transient, activate as with rootpoa
poa = rootPoa.create_POA("childPOA", rootPoa.the_POAManager(), policy);

byte[] objectId = Registrar.getBytes();
RegistrarImpl registrarServant = new RegistrarImpl(rootPoa, orb);
poa.activate_object_with_id(objectId, registrarServant);
org.omg.CORBA.Object registrator = poa.servant_to_reference(registrarServant);
String stringRef = orb.object_to_string(registrator);
poa.the_POAManager().activate();
orb.run();
}
}

D.2 RegistrarImpl.java
package IIOPInterconnect;
import org.omg.CORBA.ORB;
import org.omg.PortableServer.POA;
import BAN.MBU;
import BAN.MBUHolder;

public class RegistrarImpl extends RegistrarPOA {  
    private ORB orb;
    private POA poa;

    public RegistrarImpl(POA poa, ORB orb) {
        this.orb = orb;
        this.poa = poa;
    }

    public Surrogate registerMBU(MBU mbu) {
        try {
            System.out.println("RegistrarImpl->registerMBU aangeroepen");
            final SurrogateImpl surrogateServant = new SurrogateImpl(orb, poa, mbu);
            org.omg.CORBA.Object surrogateRef = poa.servant_to_reference(surrogateServant);
            Surrogate surrogate = SurrogateHelper.narrow(surrogateRef);
            //Surrogate returnen
            return surrogate;
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
D.3 SurrogateImpl.java

```java
package IIOPInterconnect;

import org.omg.PortableServer.POA;
import org.omg.CORBA.ORB;
import BAN.MBU;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.lang.Thread;

public class SurrogateImpl extends SurrogatePOA implements Runnable{
    private MBU mbu;
    private ORB orb;
    private POA poa;

    public SurrogateImpl(ORB orb, POA poa, MBU mbu) {
        this.orb = orb;
        this.poa = poa;
        this.mbu = mbu;
        System.out.println("Surrogate gestart");

        SurrogateImpl t = new SurrogateImpl(mbu);
        new Thread(t).start();
    }

    public SurrogateImpl(MBU mbu) {
        this.mbu = mbu;
    }

    public void run() {
        System.out.println("Run aangeroepen in Surrogate");
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        boolean stop = false;
        while (!stop) {
            try {
                String input = reader.readLine();
                if (input.equals("exit") || input.equals("quit")) {
                    stop = true;
                } else if (input.equals("request")) {
                    System.out.println("SurrogateImpl->run: request Subscription gestart");
                    mbu.requestSubscription("", "transcoChain");
                    System.out.println("Subscribe sent!");
                } else if (input.equals("activate")) {
                    System.out.println("SurrogateImpl->run: activate Subscription gestart");
                    mbu.activateSubscription("", "aaa.bbb.ccc.ddd:eeee", "transcoChain");
                    System.out.println("activateSubscription sent!");
                } else if (input.equals("deactivate")) {
                    System.out.println("SurrogateImpl->run: deactivate gesture");
                    mbu.deactivateSubscription("" retainAll: true");
                    System.out.println("deactivate sent!");
                } else {
                    System.out.println("Unknown command: "+input);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```
Assessment of Wireless CORBA in MobiHealth context

```java
...}
    }
    catch (IOException ioe) {
        System.out.println("***read error");
        stop = true;
    }
}

public void keepAlive() {
    System.out.println("SurrogateImpl->keepAlive aangesproken");
}
...}
...}
```