MASTER

Parallel processes for telecommunication with Java implementing services in an IN environment

Lutz, M.H.M.

Award date:
1998

Link to publication
Parallel processes for Telecommunication with Java

Implementing services in an IN environment

By M.H.M. Lutz

Coach: ir. J. van der Meer (Ericsson Rijen)
Supervisor: prof.ir. J. de Stigter
Examiner: ir. G.L.J.M. Janssen
Period: November 1997- August 1998
Preface

This report forms the conclusion of my Master's thesis, fulfilled as part of the Information Technology studies at Eindhoven University of Technology. The project was done at the BAC (Business creation and Applied research Company), which is part of the R&D division of Ericsson Telecommunication in Rijen. Looking back working for the BAC has been a great experience for me, especially because the BAC is an experiment of the Ericsson small company approach. It was nice to be part of a big multinational as Ericsson is and to be working within the rapidly evolving telecommunications business. Besides expanding my knowledge on telecommunications in general, I was offered the possibility to investigate the integration of the new Java language into the telecommunications world. I value the many things I learned, both technical and social. I am convinced that this experience forms an excellent starting-point for the rest of my career.

However, all of this would never have been possible without Prof. Ir. J. de Stigter, and Jan van der Meer. Prof. Ir. J. de Stigter took the responsibility for my graduation project. I would like to thank him for the interest and competence in coaching me. Jan van der Meer was my coach within Ericsson. He gave me the opportunity to do my project at the BAC. His feedback on my work was very useful. Especially I want to thank him for always believing in me, and his always supportive and enthusiastic attitude. I am very grateful to my colleagues, for they were always more that willing to help me and for making me feel a complete BAC member. Finally, I would like to thank my parents and sister for their encouragements and support during my study.

August 1998
Michiel Lutz
Summary

This report shows how Java can enrich the telecommunication environment. Java has been coupled to the Intelligent Networks environment for this reason, through the AXE-VM.

The Ericsson's AXE-10 switching machine can be deployed for Intelligent Networks (IN). IN puts the signalling and the logic in separate nodes. The logic is put in a central place while the switching remains in the switching nodes. New services can be integrated at a central place in the network. A virtual machine of this AXE-10 has been built in a Unix environment. It enables code written in C++ and Plex (language of the AXE-10) to co-operate with each other. The AXE-VM platform has been built in order to reduce the implementation time of new services. It builds upon the growing role of IT in telecommunications.

A new computer language that might be interesting in telecommunications and IN in particular, is the Java language. The way the strengths of Java can be used in telecommunications was discussed in this report. The strengths of Java are:

1. Run time environment.
2. Event handling.
4. Garbage collected heap.

Special attention was also given to some database functions, and provided networking facilities.

In order to integrate Java in the existing telecommunication environment, a communication mechanism between the AXE-VM and Java-VM has been initialised. Different communication mechanisms have been examined. The socket interface was found to be best suitable. It was implemented with a protocol to enable transparent data communication at both sides.

Different ways to integrate Java into the existing environment have been examined. The choice was made to integrate Java into the IN network. A special SIB (Service Independent Building Block) has been developed for this purpose. A set of parameters can be passed to a Java process, and a changed set returns to the IN network. A data module coupled to the SIB determines which service must be executed and what parameters are needed.

At both the IN, and the Java side a standard service creation environment for services written in Java has been designed. At the IN side new services are written as Data Modules. These Data Modules are coupled to the SIB that integrates Java into the IN network. To implement new services at the Java side, a set of standard design rules are provided to the service designer. Standard methods must be submitted in a service dependent way.

While installing, removing or upgrading services in the Java environment, both the Java and the AXE-VM can stay in active mode. At the Java side an operation and handling environment has been developed to execute services and error handling.

According to the design rules, a service will be created and integrated into the environment. In this study the created environment is tested for correctness, robustness, and performance by implementing the televoting service at the Java back end.
8.2.1 One thread for service execution ................................................. 67
8.2.2 Create a dynamic buffer of threads where service requests can route to ................. 67
8.2.3 Create a limited dynamic buffer of threads ........................................ 68
8.2.4 Create an event mechanism on top of the services .................... 68
8.3.1 Explanation of the televoting service ........................................ 68
8.3.2 Service creation according to the standard creation design steps .......... 68
8.3.2.1 Look in the ToolBox environment for reusable classes .......... 68
8.3.2.2 Implement standard reusable classes in the ToolBox environment ...... 68
8.3.2.3 Determine the number of threads that run the system .......... 69
8.3.2.4 Implement the abstract methods of the service you want to create .......... 69
8.3.2.5 Create the MML script .......................................................... 69
8.3.2.6 Insert the service in the service Creation and management environment .......... 70
8.4 CONCLUSIONS .............................................................................. 70
9. CONCLUSIONS AND RECOMMENDATIONS ........................................ 71
9.1 CONCLUSIONS ............................................................................. 71
9.2 RECOMMENDATIONS ..................................................................... 71
10. REFERENCES .................................................................................. 73

APPENDIX A: CLASS NUMBERLIST .......................................................... 75
APPENDIX B: MML COMMANDS FOR THE TELEVOTING SERVICE .......... 77
APPENDIX C: THE LABELTEXTFIELDFRAME CLASS .............................. 79
1. Introduction

This chapter provides an introduction to this report. Some background information for the scope of the project, the problem description and an outline of the rest of the report are presented in this chapter.

1.1 Background

The world of telecommunications is nowadays a dynamical world in which developments succeed each other rather fast. The competition is getting harder and harder, since the liberalisation of the telecommunication branch. The national PTTs could formerly, when they had the monopoly, take a few years to implement a new service. Nowadays it is of key importance to be the first with new features, services, etc. to strengthen one’s position.

It is only since the late 80's that new services could be developed in a new way based on Intelligent Networks (IN). The IN place the intelligence at a central place in the network, instead of putting it in every switching machine. A major advantage of this different approach is that the development of new services goes much faster and becomes independent of the brand of the switching machines.

The AXE-10 is the latest switching machine of Ericsson. This switching machine is very robust. Recently a virtual machine of the AXE-10 has been developed in order to reduce the implementation time of new IN services and the growing role of IT in telecommunications. Currently the first internal release of this virtual machine (AXE-VM) exists. AXE-VM is a platform built in a Unix environment to let Unix applications communicate with the AXE-10 and vice versa. The AXE-VM has been developed in a way that reuses AXE-10 PLEX software. It opens the door to build object-oriented services. Building services in an object oriented way makes it possible to reuse the already written code. By making a set of standard classes a service can be built up as a set of standard classes. This approach reduces the implementation time of new services again.

The last couple of years Java has come into prominence. Not only as an Internet tool but also as an object oriented language. The great advantage of Java is that it runs on a virtual machine, and therefore the applications are independent of the operating system.

1.2 Project description

Two major developments can be seen in the telecommunications branch:

1. The creation of new services is becoming more and more object-oriented.
2. The telecommunication and the information technology world is migrating into a new branch called Infocom.

In the scope of these two developments, the position that Java can create in the AXE-VM environment will have to be defined. How can the strengths of Java be combined with the existing environment?

The project is split up in different parts that subsequently lead to the answer of how Java can be combined into the existing environment.

The first part of the project is to create an environment in which the AXE-VM and Java-VM communicate to each other in a predefined way. Three steps must be made to create such an environment:

1. Build up a communication link between the AXE-VM and Java-VM
2. Determine the level at which the AXE-VM and Java-VM communicate to each other.
3. Create a data protocol to enable transparent data communication in a predefined way.

The second part defines a general service environment. An environment must be created in which services can be designed in Java without having any knowledge of the AXE-VM and the way data
communication is handled internally. Services designed in Java must be inserted into this environment without affecting the AXE-VM.

The third part defines and creates a service operation and management environment at the Java back end. Operation is the normal every day running of the exchange, including activities to adapt the exchange to the continuous changing demands made on it. The management function provides fault, configuration, performance and IN service management. At the Java side some basic functions as to insert a new service or update an old version of an existing service into the running environment must be supported.

1.3 Report outline
This section provides a brief overview of the remaining chapters of this report.

Chapter 2 provides the reader with some background information, for understanding the remaining part of the project. This chapter describes three topics:

1. The AXE-10, describing the basic structure of the Ericsson latest switching system as well as the functions it provide.
2. Intelligent Networks, describing the concept of IN and the environment in which it has been implemented.
3. The AXE-VM, describing the concept and the design principles of the AXE-VM.

Readers who are already familiar with the concepts of these topics can omit the concerning sections, since only some background information is given.

In Chapter 3 an introduction to Java is provided. It describes the Java virtual machine together with some important features for the scope of this project. This chapter is partly meant for some background information about Java, partly to see how those Java features can be of benefit for the interworking with the AXE-VM. The explained features are:

1. Events.
3. Database functions.
4. Distributed programming.

Chapter 4 describes some standard ways for network programming. It examines some interface approaches and compares them with each other. At the end of this chapter a networking mechanism is chosen to set up a communication link between Java-VM and the AXE-VM.

Chapter 5 describes the chosen communication mechanism in more detail. Design decisions for:

- the architectural level at which the AXE-VM and Java-VM communicate to each other,
- the reuse of the already available features for this type of network communication,
- the initialisation of the connection, and
- the used data protocol to enable transparent data communication,

will be given here.

Chapter 6 describes the creation of a general service creation environment. New services or service functions must be designed in Java without having to restart or affect the AXE-VM. This chapter also explains a standard way to generate calls to test the environment.

Chapter 7 describes a standard service operation environment in Java. It sets up an operation environment, that can add or remove services in Java at run time. It also provides the possibility to subscribe someone to a particular service at run time. It also describes a standard library to store reusable service functions.
Chapter 8 describes a service that has been created; the televoting service. Throughout the report all the requirements, the service designer has to take into account when designing services in Java, are described. These are listed in this chapter and the televoting service is designed according to these requirements.

Chapter 9 describes the conclusions of the research, and some recommendations for future work. Especially on expanding the created environment in Java.
2. Background

This chapter provides a brief environment analysis of this project. The Ericsson AXE-10 digital switching system supports the physical network. IN and the signalling network overlay the physical network to reduce the development time of new services. A virtual machine of the AXE-10 has been developed to let functions implemented in C++ communicate with the AXE-10 software. All these devices will be briefly explained in this chapter.

2.1 Chapter outline

The Ericsson AXE-10 digital switching system supports the physical network. The structure and the functions of the AXE-10 switching device is briefly explained in section 2.2. IN and the signalling network overlay the physical network to reduce the development time of new services and to transfer the control logic to central points. The environment where the IN services can be built up as standard building blocks is the SSI (Service Script Interpreter). In section 2.3 the IN concept together with the SSI environment is explained. A virtual machine of this switching system has been made in a Unix environment called the AXE-VM. This virtual machine allows Unix applications to communicate with AXE-10 and vice versa. Section 2.4 gives a brief overview of the virtual machine that has recently been developed.

2.2 The AXE-10 digital switching system

This section only gives a brief overview of the AXE-10; the interested reader is referred to [AXE]. AXE-10 supports a wide range of applications like PSTN (Public Switched Telephony Network), ISDN (Integrated Services Digital Network), PLMN (Public Land Mobile Network) and Business Communications. In each of these networks it also provides functionality at different levels. AXE-10 can be regarded as a number of platforms capable of supporting various mixes of applications. These platforms are the so-called "Product Lines". The product lines are for example the Local Exchange, Transit/International Exchange. The Local Exchange is the AXE deployed at local level. The transit/international exchange is the AXE-10 deployed as:

- A (inter)national transit exchange.
- A signalling transfer node in the signalling network.
- An IN node for service switching and/or service control node.
- An operator exchange known as OPAX that provides a PC LAN-based operator system.

The AXE-10 supports two languages: the ASA assembly language and a high-level language named PLEX. PLEX is an Ericsson language especially designed to implement the AXE-10.

2.2.1 Main structure of the AXE-10

Two parts each consisting of a number of hierarchical levels structures the AXE-10. The two main parts are the APT and the APZ. The APT is the switching part. It contains switching hardware for basic functions and traffic handling for more complex functions such as routing, operation and management for the exchange. The APZ is the control part. It handles; processing, file management and man-machine communication. Both parts are divided in subsystems. Each subsystem is designed to have a high degree of autonomy and connects to other subsystems through standard interfaces. Function blocks divides each sub-system. Standardised software signals handle all communications or interworking between function blocks. For security and reliability reasons this interworking is done at a central software level. Function units divide a function block. Such a function unit can be either hardware or software. The hierarchical structure is illustrated in Figure 2-1.
Due to this architecture the system is flexible, easy to modify and extendible. To cope with current and future demands the AXE-10 is continuously developed.

2.2.2 Intelligent Networks implementation in AXE-10

AXE-10 supports network wide access to IN (Intelligent Networks) services. The SES (SErvice provisioning Subsystem) takes care of this in the AXE-10. The SES is divided into two functional elements: the SCF (Service Control Function) to implement the SCP (Service Control Point) and the SSF (Service Switching Function) to implement the SSP (Service Switching Point). Section 2.3 explains the meaning of the SCP and the SSP. Figure 2-2 illustrates the functional structure of the SES and the interworking with other elements.
2.3 Intelligent Networks

2.3.1 Introduction to Intelligent Networks

In the late 80's a quite new approach to building, maintaining, changing and providing services has been developed. This approach is called Intelligent Networks (IN). The IN concept has been defined by the telecommunication industries, in co-operation with the standardisation institutes. This concept separates the service control from the call control. The call switching remains in the switches, while the control is transported to a separate platform.

To transport the call control to a separate platform new services can be implemented without updating the software in every switch. In traditional telecommunication networks introducing a new service would be a long and troublesome process. Not only because the software has to be updated in every switch but also because those switches can again be manufactured by different companies with different systems.

Due to the centralisation of service implementation, the time required to implement new services is reduced from two to five years in the traditionally telecommunication network to less than six months. Another advantage of the centralisation of the service implementation is that not only the operators can now add new services but also other parties like software houses.

To implement services independent of the switch manufacturers it is of key importance that the interface between the switches and the IN platform, and the interfaces within the IN platform itself is standardised. This standardisation of IN has been described in ITU's Q1200 series [Q.1200]. The ETSI however (European Telecommunications Standards Institute) have defined its own IN standards, but also based on the ITU Q.1200. The most important difference can be found in the definition of the INAP (IN Application Protocol).
Parallel processes for telecommunication with Java

IN is not the first concept in the telecommunication system using intelligence, but IN gives it a new approach. It describes a new type of technology, which consists of a centralised service control instead of a local service control.

In the IN environment services can be built up as concatenation of standard functionality's; called building blocks. The SSI (Service Script Interpreter) is the environment that provides these building blocks.

2.3.2 The IN conceptual model

The INCM (IN Conceptual Model) specifies the framework for the design and description of any IN based network. This INCM is intended to remain constant while the architectures will involve with increasing service requirements, and emerging technologies. Figure 2-3 illustrates the INCM. It has a four plane structure with each a different level of abstraction. The four plane structure consists of:

1. A service plane (SP) describing what a service must do.
2. A Global Functional Plane (GFP) describing how this is done without taking distribution aspects into account.
3. A Distributed Functional Plane (DFP) describing how this is done taking distribution aspects into account.
4. A physical plane describing how it is done including the physical aspects.

Specific functions implement the different planes. ETSI and ITU-T creates the conceptual model. The Q.120x series of ITU-T describes these different planes. The ITU recommends a set of Capabilities, named CS1, described in the Q121x series. Currently an improved version of this CS (Capability Set) is developed. Ericsson currently uses an extended version of this CS1 called CS1+. It of course supports the entire set of CS features.
Figure 2.3: The IN conceptual model.
The service plane
The SP (Service Plane) shows services without indicating how they are built up. One or more Service Features (SF) characterises the service. An SF consists of a specific aspect of the service, which can be used in conjunction with other SF to build up the service.

The SP contains, besides telecommunication services, also management related services that provide IN management aspects of the services.

The global functional plane
GFP (Global Functional Plane) is the plane on which network operators create basic new services and on which the SIB's (Service Independent Building block's) are found. It models the network from a global point of view meaning that no distributed aspects are taken into account. A lower level takes the distributed aspects into account. The elements in this plane are used to build up the elements in the SP. A SIB has been defined as "a standard reusable network wide capability residing in the GFP used to create service features"[Thömer]. Several SIB's are chained together to build service features and services in the SP. Figure 2-4 shows a graphical representation of a SIB.

![Graphical representation of a SIB](image)

Figure 2-4: Graphical representation of a SIB.

The logical starting point is used to trigger the SIB and the one or more logical ending points are used to trigger the next SIBs in the chain. Two types of parameters exist in a SIB:

- **Call Instance Data (CID)**. The CIDs are dynamic data parameters that only exist during the execution of a service feature and are specific to the call instance. It can either be input or an output parameter so CIDs can be transferred between subsequently executed SIBs by means of common CIDs.
- **Service Support Data (SSD)**. The SSDs are static data parameters that are specific to a certain service feature description but fixed for all service instances. The SSDs can only be used as input to a SIB.

As mentioned before several SIBs chained together perform a service. Figure 2-5 gives an example.
2.3 Intelligent Networks

The BCP (Basic Call Process) is a special kind of SIB containing the functionality of handling non IN calls. CSI describes a set of commonly used SIB's. SIBs each perform an intelligent part of the call. The BCP initialises an IN call via the POI (Point Of Initialisation) and is returned through the POR (Point Of Return). The order in which SIBs are chained together to accomplish services is defined by the GSL (Global Service Logic) and is mostly described in a service script. The SSI environment describes this service script, and will be explained later in sub-section 2.3.3.

The distributed functional plane

The DFP (Distributed Functional Plane) provides the distribution aspects of the GFP. Functional Entities (FE) specifies the elements of this block. The FE provides a detailed description and a functional specification to be applied in physical realisations. Information Flows achieve the cooperation between FEs. The FEs, shown in Figure 2-6 are:

- CCAF (Call Control Agent Function), which handles user access to the services. The CCAF can only be found in the LE.
- CCF (Call Control Function), which handles all normal calls and the switching of calls and functions (both IN or non-IN based).
- SSF (Service Switching Function), which handles the communication between the CCF and SCF and the switching of a call or service to a particular location.
- SCF (Service Control Function). The SCF controls the whole process of handling calls or services. This function is the core of the IN.
- SSCF. This is the SSF and SCF joined together in one node.
- SDF (Service Data Function). The CDF provides data about customers and the network to assist the SCF.
- SRF (Specialised Resource Function). The SRF gives and receives data to and from users.
- SCEF (Service Creation Environment Function). The SCEF handles defining, developing, and testing of IN-based services.
- SMAF (Service Management Agent Function). The SMAF handles the human interface with the SMF (for example for the operation and management staff).
- SMF (Service Management Function). The SMF handles provision of IN services, collection of statistics, managing the distribution, control and maintenance.
Because the network can only contain one master there is only one SMF. The functionality of this SMF is that it controls a number of SCFs and SSFs in the network. When mixing traffic from different network operators the co-operation between different SMFs has of course to be considered.

The physical plane
The physical plane allocates the FEs of the DFP to Physical Entities (PEs), or physical nodes. A FE must be found in at least one PE. FEs of the same type cannot be found in the same PE. Figure 2-7 shows a possible mapping of the DFP on the physical entities. The PEs consist of:

- **SSP (Service Switching Point).** This is the node that contains the SSF and CCF. It communicates with the node containing the SCF.
- **SCP (Service Control Point).** This is the node that contains the SCF, so it contains the service logic that controls the IN-based services.
- **SMP (Service Management Point).** This is the node that contains the SMF.
- **SDP (Service Data Point).** This is the node that contains the SDF. It may be accessed from the SCP or the SMP.
- **IP (Intelligent Peripheral).** This is the node that contains the SRF. It can also be used for voice recognition, code receiving, protocol conversion, customised messages, etc., and for communication with one or more SSPs.
- **SN (Service Node).** Node similar to SCP and AD and contains the SCEF, but with a direct point-to-point connection with the SSP for speech and signalling.
2.3 Intelligent Networks

2.3.3 The SSI environment

The SSI environment creates new IN services by connecting and developing standard number of CT’s (Control Types). CT’s is the in Ericsson terminology equivalent to these SIB’s. CT’s can be seen as a sequence of functionality’s that put together in a defined way implement a service. CT’s can be combined to realise the logic function of a service. Logical Modules describes these CT’s. CT functions, that involve the processing of data, are available in Data Modules (DMs). These DM contain service data and are connected to the LMs in the SSL (Service Script Logic). One or more SSLs forms the SL (Service Logic). This sub-section ends with an example of how the different parts in the SSI environment co-operate. It can be seen as an illustration of the theory explained in sub-sections 1 until 4.

2.3.3.1 Logical Modules

All LM’s except for the first one (marked START), must have the input connected, and can have zero, one or several outlets. SSL defines the LM attributes, and consist of:

- Branch or register mode
- Number of outlets
- Input register values
- Start and NTM (Network Traffic Management) Attributes

2.3.3.2 Data Modules

A DM is connected to the LM in the SSI and contains the service data. Three kinds of data can be used [TMOS2]:

1. **Local SA (Service Administrator).** This kind of data is used when the data is specific for the LM and the service. The same data is used for all subscriptions. The SA administers the local SA DM’s for a SSL. In the SMAS application and their user guides the term LDM is used for a Local SA DM.

2. **Local SC (Service Customer).** This kind of data is used when the data is specific for a subscription (Customer Data). For each subscription to a service, the Local SC DM (in SMAS CDM), for all SSI in the service, is contained in the SC.
3. **Global.** Global data is used when the same parameter values are used by several services.

### 2.3.3.3 Call Data Parameters

Call data is a unique set of data that belongs to a specific IN call. The call data is only present during call processing, and not saved between calls. So called Tag-pieces of data with varying lengths and formats separates this data. Three types of tags are distinguished [TMOS1]:

1. **Query tags.**
   
   Query tags holds data that comes in to the SCF (for example B-number and calling party category), either with first invocation or with a hand over. These tags exist and can be used from the start of a service.

2. **Temporary tags.**
   
   Temporary tags consist of temporary numbers and variables available during script execution, and must be created before they can be used in a service script. These tags are the only ones that normally remain unchanged during a service execution.

3. **Response tags.**
   
   Response tags are the tags that are prepared during service processing. They can be translated into operations and parameters. They are sent to the SSF with the RESPONCT (and are lost from the SS point of view)

CTs sort this Call Data in a slightly different way: this is done according to their formats. The format groups are:

- **Kind of Numbers.** These are the typically numbers such as A- and B-numbers with each a unique value. [TMOS1:page 35] gives a list of all KoN (Kind of Number) formats and there usage.

- **Kind of Variables.** These are 16-bit variables including the 7 input registers. KoV (Kind of Variable) is a way to identify pieces of call data. Each value used in Service Processing is identified by a unique value. When a LM needs to work on a variable, this is indicated by the chosen KoV for this LM. [TMOS1:page 40] gives a list of all KoV values.

- **Kind of Long Integer.** These are 32-bit variables, some of them consisting of pairs of the IRR registers. It is also a way to identify pieces of call data and are therefore similar to the KoVs. [TMOS1:page 43] gives a list of all KoV values.

### 2.3.3.4 System Service Scripts

Three System Service Scripts exist, that will have to be defined when creating IN services. These three System Service Scripts are [TMOS1]:

1. **Access Service Script.** This script is the entrance of the database for call handling. It is the first script activated after IN service request from the SSF.

2. **Error Service Script.** This script is activated when SSI detects an error during service processing. When a fault is detected during execution, an internal error code, the SA (Service Administrator), and more information stored in IRR and the ERROR service script is invoked.

3. **Update Service Script.** This script is the entrance of the database for updating of data by other entries. After an update operation this script is activated.

### 2.3.3.5 Example

An example, of how the SSI-environment looks like for a certain service is given here. The freephone service has been implemented for this purpose. A company has three establishments each having its own telephone number. During office hours the service request is routed to the establishment nearest to the caller. Outside office hours and in the weekends two of the establishments are closed and the service
2.4 The AXE-Virtual Machine

A virtual machine of the AXE-10 has been made in order to reduce the implementation time of new IN services and the growing role of IT in telecommunications. AXE-VM is a platform built in a Unix environment to let Unix-applications communicate with the AXE-10 and vice versa.

The Unix environment opens the door to implement new services in an object oriented manner. The reuse of existing classes and the implementation of new services build up by a combination of defined classes reduces the implementation time of new services. The AXE-10 software is also reusable in applications running on the AXE-VM so a minimum duplication of existing software has to be made.

2.4.1 Design principles
The AXE-VM has been developed with the following design principles:

- Real-time behaviour, because the AXE-10 is a real time machine.
- Rather than building a software copy of the AXE-10 the focus is to assure the correct functional behaviour.
- Reuse of terminology wherever possible and applicable.
Parallel processes for telecommunication with Java

- Multi-component architecture development, because the development of the AXE-VM is a cooperation between ETM and UAB where components critical to one or both parties must be replicated.

- Make it easy to extend the application functionality.

- A fault in the AXE-10 software may not crash AXE-VM

- Designed to be an Object-Oriented application in order to reuse the implemented software.

2.4.2 AXE-VM structure

The AXE-VM product structure consists of three levels:

1. **ANT Level.** At this point the AXE-VM structure fits in the overall product structure.

2. **CNT level.** This is the sub-system level.

3. **Software units (CAA's).** These are sets of logically related packages. Packages are used to create a specific executable program. The goal of packages is to have little coupling between packages and high coupling within a package. Two types of packages exist in the AXE-VM: the market specific and the core packages.

The software units are allocated to one of three layers:

1. **Application layer.** This is the highest layer. It contains all the software units having a top-level functionality, such as the mains. This layer is dependent of both the functional layer and the support layer.

2. **Functional Layer.** This is the layer containing the functionality specific to the application. This layer is only dependent of the support layer.

3. **Support layer.** The support layer contains software units not specific to AXE-VM, and is typically used in other software units and application components. This layer is not dependent of any of the other layers.

2.4.3 Event driven

The AXE-VM has an event driven processing structure. This means that the control flow structured in asynchronous applications that interact with the environment. The events are processed by a priority order and managed by an Event Manager, stored by the core. Being an event driven application means that all functions (blocks) that are part of the AXE-VM shall follow the rules for an event driven application. Four groups of events exists:

1. Stream. This occurs if the status of a file descriptor changes.
2. Asynchronous Unix signals
3. Process generated application events
4. Timer events

The Event Manager provides two types of event handlers: the event-loop and the event call-back. The event-loop checks for waiting events and dispatches an appropriate event-handling routine. In case of no waiting events, the event manager will do nothing until an event occurs. The call-back function is a function, registered by the application, that knows how to respond to a certain event.

2.4.4 Communication between blocks in the AXE-VM

Communication between blocks in the AXE-VM takes place at system level. Communication at system level means that communication between two blocks is routed through the AXE-VM platform. Blocks can not access data directly. The AXE-VM takes care of the routing. Communication between PLEX-blocks, between C++ and PLEX-blocks, and even between blocks written in C++ is provided at system level.
2.4 The AXE-Virtual Machine

Signals are the standard way this communication is handled. A signal consist of the following data:

- The block that sends the signal
- The block that shall receive the signal
- A global signal identity called global signal number (uniquely identifies the signal in AXE-VM)
- A local signal identity, called local signal number (uniquely identifies the signal in the block that processes the signal)
- Processor identity, which represents the axe-vm where the signal shall be executed
- A priority level of the system
- A format, indicating the amount of data passed in the signal
- The signal data, an array of maximal 64 32-bit data words

This signal has nothing to do with the standard Unix signal. The available information in this signal is passed in every block call and after a block has been executed this information is updated. These signals can only be updated in the same process.

2.4.5 Signal sending between blocks
Signals contain data that identifies the block that sends the data and the block that shall receive the data. A number of ways exist to send the data:

- Buffered (single signal). Signal is sent via a signal queue named a job buffer. They are handled according to priority in a FIFO way without pre-empting a currently processing signal.

- Combined forward. The sender stops executing after a signal sending until he receives a signal back from the other block. The signal should be executed immediately when sent.

- Delayed. Signals are executed after a specific period of time.

- Distributed. Signals that should be executed on a remote AXE-VM that is running in a different process.

2.4.6 Signal routing between blocks
The event manager starts execution and invokes the scheduler. The scheduler asks the job buffer for the signal with the highest priority. This signal is forwarded to the dispatcher. The dispatcher will, after looking into the signal, forward the signal to the appropriate block. During execution the block can invoke several blocks. These blocks are:

1. The JobTable
2. The TimeQueue for delayed signals.
3. The Scheduler for a buffered signal
4. Back to the dispatcher for a combined forward signal.

In case of distributed execution a ProcessorSelector and SignalTransporter will have to be included. Figure 2-1 shows the routing of the signals through the blocks and the distributed signals between several AXE-VMs.

2.4.7 Initialisation of the AXE-VM

At initialisation time first some Kernel blocks are installed and a dump (back up) is loaded of the actual state of the needed AXE-10 blocks. Further some pre-defined user SimBlocks are loaded. User SomBlocks is the standard format the C++ looks like from the outside. Figure 2-11 illustrates the initialisation of the AXE-VM.
2.4.8 The AXE-VM service gateway

The service gateway is a service platform, on which applications can interwork with IN platforms. It supports a real time processing environment with multi-tasking capabilities and access to all operations of the IN CS1+. Transport between AXE-VM and AXE-10 is done through TCP/IP. Links can be connected in parallel between the front end of AXE-10 and the back end of Unix. The protocol adapter will select these links in a round-robin fashion.

2.5 Summary

This chapter provided the user with some background information about the remaining part of this report. Three topics are explained:

- Some fundamental parts of the AXE-10. The structure is described and consists of two main parts, the control part and the switching part. Each of these parts is divided in sub-systems. Special interest is given to the IN implementation in the AXE-10.

- The Intelligent Networking concept as described in the ITU and ETSI standardisation documents. The conceptual model describes the IN on four levels of abstraction planes: The service, global functional plane, distributed functional and the physical plane. As part of the global functional plane the SIB’s (Service Independent Building Blocks) are standard building blocks, which coupled together create services. The environment in which the IN services are created (as a number of building blocks) is called the SSI (Service Script Environment).

- The virtual machine that has been made of the AXE-10, the AXE-VM. The design principles are listed and the structure is explained. Signals take care of the communication in the AXE-VM. What signals are and their routing is described. An overview of the special set of actions that have to be executed during initialisation time is also given.
3. Java

This chapter provides some information about Java, what Java is and how it can be used in which environment. Important capabilities like, threads, events, database functions, and network programming are given special attention. The end of this chapter lists an overview of capabilities that are of particular interest for the scope of this report.

3.1 Introduction to Java

This section only gives a brief global overview of Java. To know more about Java the interested reader is referred to [Java]. Java is used to create executable content that can be distributed through networks. In order for users to use Java content there must be a key piece of software: the Java interpreter. Executable content means that the downloaded content is not only displayed in the browser, but it can be animated, executed and distributed. To view the content on the web, a user's web browser must be Java enabled. In the first release that was only HotJava, but currently the Netscape Navigator Web browser has been upgraded to interpret Java programs.

The goal of the developers was to create a platform independent code that allows the software to run on any CPU (Central Processing Unit). The developers first focused themselves on C++, but they could not get C++ to do everything they wanted, so they developed a new language later known as Java. For this reason, similarities with C++ are not surprisingly. The main differences between Java and C++ are:

- Java does not contain pointers
- Java does not contain multiple inheritance
- Direct memory access is shielded for the programmer, even at Java-VM level.
- Java is automatically garbage collected.

These points help to keep the language simple and clear. Java has been developed at Sun Microsystems.

The JDK (Java Development Kit) is used for the development of applications and to create applets. It contains what you need to start programming in Java like a compiler, an interpreter, and a debugger. The JRE (Java Runtime Environment) is also a part of the JDK. It consists of the Java core classes the Java-VM (Java-Virtual Machine) and supporting files. Another part of the JDK is beans. A bean is a reusable software component that can be manipulated visually in a builder tool. Beans are interpreted not compiled. Some important features of beans are the support for events as a simple communication metaphor. Beans have to run in design environment and in run-time in a multi-threaded environment. The main difference between beans and class libraries is that class libraries can not be manipulated visually.

3.2 Java Virtual Machine

Java has been developed to be able to run on any operating system. In order to achieve this level of portability, the language and the (run time) environment behaviour must be specified precisely. In order to achieve a run time system Java has invented an abstract computer of its own, where it is running on. This abstract computer is called a Virtual Machine. Java code is compiled into bytecodes. Bytecodes are simply a stream of formatted bytes, with each a specific handling specification. One of the main tasks of the virtual machine is the fast, efficient execution of the Java bytecodes in methods.

3.2.1 The fundamental parts of the Java-VM

The Java virtual machine can be divided into five fundamental pieces:

- **Bytecode instruction set**
  Values of any identifier must be specified in bytecodes or supporting structures. A bytecode instruction consists of a one-byte upcode for the identification of zero or more operands. When operands are more than one byte long the high order byte is stored first. Bytecodes interpret data in the run-time memory areas as belonging to a fixed set of types. Different bytecodes have been
Parallel processes for telecommunication with Java

designed to uniquely handle each of these types. [Java, 692] describes the complete set of bytecodes.

- **A set of registers**
  The registers are used to hold the machine’s state, and affect its operation. They are updated after each bytecode has been executed. The VM defines the different registers to be 32-bits wide. Registers are not used to pass and receive arguments.

- **The stack**
  The stack is used to supply parameters to bytecodes and methods, and to receive results back from them. A stack frame holds the state for a single method call. This frame consists of three sets of data: the local variables for the method call, its execution environment, and its operand stack. An array of 32-bit slots indexed by the variable storage register, stores the local variables. The execution environment in a stack frame helps to maintain the stack itself. The operand stack is a 32-bits FIFO stack, used to store the parameters and return values of most bytecode instructions.

- **A garbage collected heap**
  The heap is that part of the memory where newly created objects are allocated. Objects are automatically garbage collected so the objects in the heap do not have to be de-allocated.

- **An area for storing methods**
  This area stores the Java bytecodes for implementing almost every method in the Java system. The method area is aligned on byte boundaries.

### 3.3 Events

Events is a mechanism to provide state change notifications between a source object and one or more target objects. To standardise the event mechanism in Java, and Java beans in particular, a standard set of interfaces can be created for communication between different objects or to the environment. The Java event model is based on method invocation. The target listeners objects bring forth the event notifications from sources to listeners.

**Event state objects**

An event state object is a set of actions that must be taken, if the event belonging to this object occurs. The event state objects should be considered immutable, therefore the direct public access to the fields is avoided, and instead use assess methods to expose details of event state objects. The access methods should follow standard design patterns. The latter is especially important when bridging events between Java and other component architectures.

**EventListener**

Event listeners interfaces consist of one or several event handling methods that belong together. Each event handling method is defined as a distinct Java method. As argument normally one object of event state objects is passed. Event notifications are forwarded to an external environment it is allowed to endow one or more arbitrary parameters. The event source classes must provide methods for registering and de-registering event listeners, because potential event listeners must register themselves in order to establish an even flow from the source to the listener.

### 3.4 Multi-threading

A thread is a single locus of control for the program. Multithreading programming languages such as Java is, enable several different execution threads to run at the same time inside the same program, in parallel, and without interfering with each other. All different threads will run independently of each other. Many threads can work together at the same time, but it may eventually exhaust the system so that all the threads work slower. There is some overhead associated with the context switching, so creating too many threads more CPU time will be spend changing context than executing the program eventually. When data is shared with not thread safe programs it’s important to protect the data. The Java classes are all thread save, the only thing the programmer has to worry about is the synchronisation and thread-ordering problems. The system itself always has a few so-called daemon-threads running e.g. for garbage collection and for mouse event listening.
3.5 Database-functions

Synchronisation
In Java a synchronisation key word exists named synchronize. This keyword tells Java to make the block of code in the method thread-save. Synchronized allow only one thread in that method at once. Other threads have to wait until the currently running thread has finished execution. The protected area must be defined as small as possible to execute more efficiently. In this case fewer threads get stuck waiting to get into protected areas.

Deadlock
Directly derived from synchronisation issues is the possible existence of deadlock. Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects. If a multi-threaded program locks up occasionally, deadlock is one of the first conditions that must be checked for.

Thread-ordering
The part of the system that takes care of the real-time ordering of threads is called the scheduler. The Java system does not precisely specify the behaviour of the scheduler, and therefore the exact order in which threads execute is not known precisely. A scheduler can have two different ways to look at a job: non pre-emptive scheduling and pre-emptive time slicing. The non pre-emptive scheduler runs the current thread forever. The thread has to notice that it is save to start another thread. This type of scheduling is quite valuable in extremely time critical, real time applications. With pre-emptive time slicing each thread is given a time slice. The current thread is suspended and resumes another thread for the next time slice. Most modern schedulers used the principle of time slicing. The Microsoft NT OS uses the principle of time slicing. Threads can be assigned priorities. Those with a higher priority get to run first. If threads have the same priority the behaviour is not known, due to these different scheduler mechanisms.

3.5 Database-functions
Java is weak on the database front, but it is undergoing rapid development. It has its own database connectivity: the JDBC (Java DataBase Connectivity). The JDBC is loosely based on microsoft ODBC (Open DataBase Connectivity), but different enough that DBMS had to write new drivers. Some individuals have written a database in Java. The jDB is a good example of this.

Java DataBase Connectivity
The JDBC is a powerful API. It offers SQL (Serial Query Language) compatibilities and sophisticated database development capabilities. The JDBC API defines Java classes to represent database connections and result sets. It allows a Java application to issue SQL statements and to process the results. JDBC masks the differences between different data sources and provides a standard interface to any Java application. The JDBC is only usable for not very fast real time database calls.

jDB
The jDB is a database back end written entirely in Java [jDB]. It is especially designed for small databases, and consist of a set of Java classes that allow a programmer to easily integrate a database into their applications. The jDB-package contains classes to create and use databases. There is however a major drawback, because jDB will always be much slower than an equivalent package written in C.

3.6 Java distributed programming
In order to communicate with other processes, data has to be transported over a network. This section gives a brief overview of the standard Java classes that use network support. The Java network support classes are joined together in a package called; Java.net. Raw network interface classes are explained in more detail. A language neutral way to specify an interface between an object and its client on a different platform is the Java-IDL. This section also gives a brief explanation of the IDL.

3.6.1 Java.net package
The package Java.net covers the classes, interfaces and exceptions of common network programming. The classes in the networking package fall into three general categories:

1. Web interface classes. For resources located on the WWW via the HTTP protocol the URL and URLConnection class provides access to it.
2. **Raw network interface classes.** To provide access to plain bare-bones networking facilities the `Socket`, `ServerSocket`, `DatagramSocket`, and `InetAddress` classes exist. These classes are the building blocks for implementing new protocols, and talking to existing servers.

3. **Extension classes.** Some classes are implemented to extend the capabilities of the `URL` class. The extension classes can be user made, if the programmer takes some standard rules into account.

Since the interface to the AXE-VM is not through the WWW, I will not explain the `URL` classes. The interested reader is referred to [Java].

### 3.6.2 Raw network interface classes

The socket interface is one of the raw network interface classes. The socket interface is a common interface for data transport between different processes. Java supports this socket interface by some classes:

- A **Socket** object is the representation of a TCP connection. After a socket has been created, it opens a connection to a certain destination. Stream objects are used to send and receive data to the other end. The constructor takes two arguments into account: the name or the IP-addresses and the port number to connect to.

- A **SocketListener** is the representation of a listening TCP connection. When the incoming connection requests the ServerSocket, it will return a Socket object representing the connection. For further handling of the connection the system creates another thread, to free the ServerSocket to listen for a next connection request. The constructor carries at least one argument. That is the local port number to listen for requests. An additional argument is the maximum time to wait for a connection.

- A **DatagramSocket** is the representation of a connectionless datagram socket. This class works with the DatagramPacket class to provide for communication using the UDP (User Datagram Protocol) protocol. The DatagramPacket objects represent one packet of data. While the transmission is connectionless it provides an unreliable communication path. Unreliable in the sense that the sender does not get an acknowledgement on correct receiving.

- The **SocketImpl** class provides a mapping from the raw networking classes to the native TCP/IP networking facilities to the host. Each Socket or ServerSocket calls this function to access the network. The SocketImpl class is completely shielded for the programmer and can only be accessed through the standard available socket classes.

### 3.6.3 Java-IDL

The Java-IDL is developed to the OMG-IDL (Object Management Group-Interface Definition Language) specifications, as a language neutral way to specify an interface between an object an its client on a different platform. Java-IDL is part of Javasoft's platform API. It provides standard connectivity with CORBA (Common Object Request Broker Architecture). CORBA is the standard for heterogeneous computing. IDL only describes interfaces and not implementations. It is a programming language independent way of describing interfaces to objects. With Java-IDL these IDL definitions can be compiled with the idltoJava stub generator to generate Java interface definitions and Java client and server stubs.

The Java IDL language mapping specification defines the mapping from IDL to Java. IDL allows a Java client to transparently invoke operations in CORBA objects that reside on remote servers, and allows Java servers to define object to transparently invoke operations from remote CORBA clients. CORBA implementations can inter operate transparently if they communicate with the same network protocol. This used protocol is the IIOP (Internet Inter ORB Protocol). IIOP does actually nothing more that provide a communication path between two ORBs. The components of the Java-IDL are:
3.7 Other investigated communication methods with Java

- **Generic ORB core.** ORB runtime that allows Java IDL applications to run either in stand-alone Java applications, or as applet within a Java-enabled browser.
- **IdltoJava.** This is a development tool that automatically generates stub code for specific remote interfaces.
- **nameserv.** This is an implementation of the COS (CORBA Object Service) name service.

### 3.7 Other investigated communication methods with Java

The standard JDK provides standard network communication. This section describes two not standard communication methods; the JavaPP and the Java Native Code.

#### 3.7.1 JavaPP

JavaPP stands for Java Plug and Play and is an international co-operation of researchers over the world. Several universities that are doing research in real-time, embedded systems, and formal methods are putting their theories together in practice, based on the computer language Java [JavaPP]. The project deals with the development of communication channels and compositions construct based on channels. This channel is the replacement for the monitor construct in Java. It is used to avoid deadlock and to free the programmer from synchronisation and scheduling constructs. Channels are objects that pass objects between objects [Hiid]. Also a JEPS (Java Embedded Process Scheduler) is a project of JavaPP. This JEPS is intended to replace the thread class that makes synchronisation, scheduling and the data transfer invisible to the programmer. It also contains building blocks and the availability to add your own blocks. The channels are used for the communication between the building blocks.

This structure has a high level of similarity to the way signals and blocks communicate in the AXE-VM.

#### 3.7.2 Java Native Code

Native means that a language other than Java implements the code. Two reasons exist for using native code:

1. Utilise a special capability of your computer or operating system that the Java class library does not already provide. These features must be implemented in C or any language that can link with C.
2. For speed reasons. At present time, Java releases do not perform as well as for example an optimised C program on many tasks. Time critical parts can be written in native code to gain performance.

The first reason can perhaps be converted into a communication protocol between Java and C. A Java2c translator translates a whole class into a portable C source code version.

### 3.8 Conclusions about Java

Java supports some important features that are obligated in the AXE-VM. These features make it possible to let Java interface with the AXE-VM.

Some features of Java and their relation towards the environment are:

- **Multithreading.** A multi-threaded program runs different processes in parallel. Also a protection mechanism is available to not disturb too time critical operations. In telecommunications this is of vital importance because it supports handling of more than one service request at the same time.

- **Event handling.** This is a communication metaphor. It can be used to connect beans and Java with a different application.

- **Both run-time and design environment.** Since the AXE is a run-time system the language it connects to must also have run-time capabilities.

- **Database function.** Some database functions can perhaps better be implemented in Java. It is not a necessary feature but it can in some cases be an improvement to the system.
Parallel processes for telecommunication with Java

• **Distributed programming.** Java itself is not distributed, but the architecture neutrality and dynamic extensibility make it a good candidate for distributed heterogeneous applications. The designer will need to define a communication path, so that both parties can understand each other.

• **JavaPP channels** cannot be used as a communication mechanism between Java and the AXE-VM, but for communication between different blocks in Java it may very well be suited.

• **Java native codes** will not be used as a way to communicate to C. It is definitely not made for this purpose and it seems difficult and tricky to cope with, if it is possible at all.

Some general features of Java, that may cause some limitations to the environment:

• **Java is relatively slow.** Java has a longer execution time than a similar program run in C++. Even when optimal using native code, the performance is only similar to C++.

• **Thread scheduling is not uniformly done.** Different platforms have different behaviour at this point. Some behave more like a pre-emptive and some more like a non-pre-emptive scheduler. At this moment the situation is not known. Non-pre-emptive scheduling is necessary for time critical real time applications. A thread can be assigned a priority in the range of one to ten. A guarantee that threads with the same priority are handled in a FIFO way cannot be given.

• **Currently Java does not support any network security issues,** as authentication, integrity of data, and so forth.
4. Interfacing the JAVA-VM with the AXE-VM

The goal of the interworking between Java-VM and the AXE-VM is to integrate Java into the existing telecommunication environment. In order to achieve this integration a network connection must be set up to let the Java-VM and the AXE-VM communicate to each other. Different general network connection approaches are explained in this chapter and a choice is made which network connection will be used.

4.1 Chapter overview

The mechanism of sending and receiving data is explained in section 4.2. The approach of making a connection is based on client-server computing. Section 4.3 explains the Client-Server computing principles. The two common TCP/IP APIs: socket and RPC, are explained in section 4.4. A distributed system can come to hand at a higher level. Some of these systems are explained in section 4.5. Section 4.6 gives an overview of the general network connection principles and one principle is chosen to be the best one for the problem described in this report. Another approach is to look what IPC facilities Unix provides, because Unix is the OS for the AXE-VM process. Section 4.7 explains these IPC facilities. This chapter concludes with a section where one method is chosen to set up the communication link between the AXE-VM and Java-VM.

4.2 TCP/IP

The TCP/IP will be used for the connection mechanism between the AXE-VM and Java-VM for two reasons:

1. The connection between the AXE-10 and the AXE-VM is based upon the TCP/IP protocol
2. The TCP/IP connection protocol provides the ability to allow a wide variety of different types of devices, from different vendors, to interoperate with each other.

This section explains the principles of the TCP/IP protocol.

A network-layered architecture has been developed, named the TCP/IP protocol. In order to create meaningful communication at each level, the TCP/IP layers are often compared with the OSI model. ISO has developed a single international standard set of communication protocols that could be used in a computer network. It is called the reference model for Open System Interconnection. OSI forms a basis for an international standard architecture for networking. The TCP/IP layers are illustrated and compared with the OSI model functional layers in Figure 4-1.
For the actual transport two layer protocols exist: the UDP (User Datagram Protocol) and the TCP (Transmission Control Protocol).

The UDP is a connectionless data delivery service. UDP is an unreliable way of sending data through the network, because there is no control flow and delivery acknowledge. Errors can occur during transportation and the application itself must implement any required reliability controls.

The TCP on the other hand is connection oriented, and provides a reliable sequenced data transfer service. The handshake protocol performs the connection establishment. A hand-shake protocol means that first a communication link has been set up before the actual data is transmitted. The TCP data transfer is a full duplex service in which data can flow in both directions between the two communication processes. The TCP also provides error detection and retransmission, flow and congestion control, and a proper connection release [TCP/IP].

IP is a packet switch networking protocol, that takes care of the routing in the network. It allow the interconnected individual networks to give the appearance of a single unified network [TCP/IP].

4.3 Client-Server computing
A client-server system consists of a client and a server component. The client component issues the service request and the server processes the request and returns the result. A service request consists of three subsequently performed steps:

1. Open a communication channel.
2. Send information about the requested service.
3. Accept the service response messages from the server.

Two types of servers exist: Iterative and concurrent servers. An iterative server process handles only one request of one client at a time. A possible second client waits until the first client's request has been executed. Concurrent servers on the other hand can process more client requests at the same time. Issuing a function that creates a separate thread, task or process to handle each request makes it concurrent. The client closes the communication channel and terminates its execution.

Three basic elements lie at the heart of the client server computing environment [TCP/IP]:

- **Infrastructure functions.** These functions should be available to all systems in the environment. If common functions are available, the infrastructure functions do not have to include processing
logic to implement them. IP provides many of these functions, like multi-session, management, security, naming and timing mechanisms.

- **Client-Server models.** Different types of computing configurations exist for different implemented client-server models. These models fall into one of the three categories: user interface, function, and data access processing. Some model examples are: broadcast and object invocation.

- **Distribution technologies.** In the IP environment three distribution technologies exist: Remote database access, message passing, and remote procedure call mechanisms.

Choices about the client-server models and distribution technologies will be made further in this chapter.

### 4.4 IP APIs to communicate over a network

For distributed programming two general APIs exist to communicate over a IP network: a socket and a RPC (Remote Procedure Call) approach. Both provide a way to access the IP protocol. This section describes both approaches in more detail.

#### 4.4.1 Socket interface

The socket interface is the most commonly used interface for network programming in the IP environment. Three types of sockets exist to support communication in the IP environment:

- **Stream socket.** These sockets support a connection-oriented form of data transfer using the TCP Transport layer protocol in which a stream of data can be sent in a reliable way.

- **Datagram socket.** These sockets support data transfer using the UDP Transport layer protocol in which a stream of data is sent. This data is sent in an unreliable way.

- **Raw socket.** These sockets provide access to the underlying IP and ICMP (Internet Control Message Protocol) processes. They are generally only used for specialised purposes, such as writing network troubleshooting tools.

Figure 4-2 illustrates the structure of the interface of these three sockets with the IP protocol.

![Figure 4-2: Three types of sockets.](image)

Each of the two processes (the client and the server process) is identified by a separate socket address. A 5-tuple completely specifies the two processes. It consists of the following:
Parallel processes for telecommunication with Java

- An identifier to know which protocol is being used for communication.
- The port number the process is identified to.
- The host address (process address if both processes run on the same system) the server is running on.
- The remote port number.
- The remote host address (process address if both processes run on the same system).

Socket call description
The 5-tuple that completely specifies the two processes are implemented in the socket call description. To set up a socket connection, the socket call description consists of a number of calls, that have to be performed subsequently:

- **A socket system call** is used to initialise the socket locally. It carries three integer arguments: a Protocol type family, a Type of the socket, and Protocol associated by the call, and one return integer value. This return integer value will be used to refer to the socket. In case of a failure it returns -1.

- **A bind system call** fills in the endpoint identifier (Internet address and port number). In a connection-oriented application protocol the bind system call can be omitted, the connect system call automatically fulfils this. It carries three arguments: A socket identifier, the local address, and the address length.

- The **connection oriented application protocol consists of another three system calls**:
  1. **A listen system call**, used by clients, tells the communication software that the server process is willing to accept a request. It carries two integer arguments: The socket identifier and the maximum queue length of waiting clients.
  2. **A connect system call** takes care of the handshake procedure that carries all the information necessary to set up the communication path.
  3. **An accept system call** causes the server process to wait for a connection request to arrive from a client. It carries three integer arguments: the socket identifier, the client address, and the address length. The result is a creation of a new socket data structure and a descriptor integer to reference this new data structure. In case of a failure it returns -1.

- **Write and send system calls** are used to send data to the remote process. They carry the following arguments: the socket identifier, a buffer that contains the sent data, the length that contains the number of sent octets. The send system call also contains a flag that specifies special options. The result value is the number of actually transmitted octets, or -1 in case of a failure.

- **The read and recv system calls** accept incoming data. They carry the following arguments: The socket identifier, a buffer and the length. The recv system call carries an additional argument called flag to specify special options.

- **A close system call** closes the socket after completion of the communication. In the connection-oriented approach this system ensures that all queued data has been sent before it releases the connection.

Figure 4-3 illustrates the scenario for the connection oriented protocol. The left side is the server side. It sets up its side of the socket connection and waits for the client to connect to the port number the server is listening to.
4.4 IP APIs to communicate over a network

4.4.2 Remote Procedure Call
This approach makes it possible to implement applications without the need for explicitly issuing requests for communication services. Because procedure calls are a well-understood mechanism for transferring control and data from one procedure to another within the same computing application, it is a logical step to extend this to a set of procedures in a distributed environment. Syntactically it looks just like an ordinary procedure call. Figure 4-4 illustrates the protocol.
A RPC will involve the following 8 steps:

1. Locate the server for the call.
2. Marshall the arguments into a proper format.
3. Transmission of the call through a network protocol (For Example TCP/IP).
4. Unmarshall the arguments in a proper format for the server process.
5. Perform the procedure.
6. Marshall the return value or the exception information.
7. Transmit the response back.
8. Unmarshall the response at the client side.

Three RPC call semantics exist:

1. At least ones. This semantics allows the calling procedure only to determine that the procedure executes at least one time.
2. At most ones. This semantics allows the calling procedure only to determine that if the procedure has executed at all, it has executed only one time.
3. Exactly ones. This semantics allows the calling procedure to determine that the procedure has been called once and only once.

Two major software subsystems exist that typically provide remote procedure call facilities in the TCP/IP environment. One developed by Sun Microsystems, and one by Hewlett-Packard. Both of them provide “at least-ones” and “at most ones” semantics. Only the Sun facility will be explained because the environment is a Sun Unix environment. It consists of the following components:

- **Rpcgen Compiler.** It accepts remote procedure call interface definitions and generates stubs.
- **XDR (eXternal Data Representation).** This is a facility that provides a standard method for encoding argument data and results.
- **Runtime library.** This is a collection of executable routines
- **Portmapper facility.** This provides programs with information about the location of callable services.

A RPC call message has three integer fields: remote program number, remote program version number, and a remote procedure number. These three fields uniquely identify the called procedure. Multi-session is not automatically supported, because a particular request frees the space from the previous request of the same type.

### 4.5 Distributed systems

A distributed object system is a mechanism to deal with objects or procedures on remote systems, as if the objects were locally stored. It has been developed to minimise design, development, and complexity issues. It operates at a higher abstraction level. At this higher abstraction level the handled protocols
can be embedded in platforms. The distributed systems use sockets or RPC facilities to handle the transport at a lower level. This section investigates and explains some distributed systems.

4.5.1 DCOM

DCOM (Distributed Component Object Model) has been developed at Microsoft and it is part of the Windows'95/Windows NT operation system. Therefore it is particularly used in the PC market. A single company develops it and it is not a standard. For these reasons this system is not investigated anymore. In the near future it may become an issue, as a bridge between CORBA and DCOM is currently under construction.

4.5.2 CORBA

CORBA (Common Object Request Broker Architecture) is a part of the OMA (Object Management Architecture) specified by the OMG (Object Management Group). This means that CORBA is a widely used standard of handling exceptions, and protocols. It is directed at distributed, heterogeneous object oriented applications. Distributed in the sense that different objects are hosted across a network of computers or even several networks connected by gateways, and heterogeneous means that the network may contain a diversity of platforms. All these objects communicate with each other without worrying about these differences. The OMA also specifies Common Object Services and Common Facilities. The Common Object Service supplies the ORB (Object Request Broker) with useful features such as naming, persistence, and concurrency control. The ORB is responsible to mediate all the differences between the systems.

CORBA also supports a dynamic method invocation, at run-time, via the DII (Dynamic Invocation Interface) at the client side and DSI (Dynamic Skeleton Interface) at the server side. CORBA passes object references rather than objects to avoid worrying about object migration. CORBA provides a rich set of extensive run-time facilities. Everything that IDL does statically, can be done dynamically.

ORB

The ORB is responsible for taking away the differences between the systems. The request and the response sides pass it. The following 8 steps the ORB must specify:

2. Server location for the object.
3. Transmit the request through for example a socket mechanism.
4. Create a process at the server side to handle the request if necessary.
5. Unmarshall the arguments into the required format at the server process.
7. Transmit the response, if necessary.
8. Unmarshall the response at the client side.

Figure 4-5 illustrates the concept of the CORBA ORB.

![Figure 4-5: The ORB.](image)

The three main features to object oriented programming are supported. These features are:
1. Encapsulation. All API’s support encapsulation
2. Inheritance. Inheritance means that a new class can be created as an extension of another
3. Polymorphism. Polymorphism for example in the sense of overloading methods.

The interested reader is referred to [Corba] for further information

4.5.3 DCE

DCE (Distributed Computing Environment) provides an integrated solution to the problem of building distributed applications across heterogeneous networks. DCE has been designed for procedural programming. Java on the other hand has especially been developed for object oriented programming.

RPC handles the communication at a lower level, and has already been explained in section 4.4.2. It uses IDL for the procedure call protocol. The most important benefits from DCE are the naming and the security of services. The conceptual naming model is based upon hierarchy. It has a tree structured name space similar to the hierarchical file system. The leaves, called resources, consist of files, servers, and users. DCE uses security services to protect all the data structures. It provides authentication, registration, authorisation and privacy in a distributed environment [DCE]. Threads are also supported to handle multiple requests in the application level implementation. It does not yet furnish event notification.

4.6 Inventory of common networking facilities

This section provides an overview of all discussed methods. A decision of which approach is best suited for communication between the AXE-VM and Java-VM is made. Table 4.6-1 gives an overview of the different approaches.

Table 4.6-1

<table>
<thead>
<tr>
<th>Simple (simple interface)</th>
<th>Complex (distributed systems)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster due to little overhead</td>
<td>Slower due to more overhead</td>
</tr>
<tr>
<td>less controlled</td>
<td>more controlled</td>
</tr>
<tr>
<td>No platform it is working on (peer to peer)</td>
<td>More general platform</td>
</tr>
<tr>
<td>All steps to make up the connection are programmer made</td>
<td>The programmer does not have to worry about platform differences</td>
</tr>
<tr>
<td>Must design what a distributed system tries to eliminate (lower level)</td>
<td>Implemented a higher abstraction level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Socket</th>
<th>RPC</th>
<th>DCE</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Oriented</td>
<td>Procedural</td>
<td>Procedural</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>Lack of security</td>
<td>Lack of security</td>
<td>Security issues</td>
<td>Lack of security¹</td>
</tr>
<tr>
<td>Uses TCP/IP at lower level</td>
<td>Uses TCP/IP at lower level</td>
<td>Uses RPC at lower level</td>
<td>Uses sockets at lower level</td>
</tr>
<tr>
<td>event notification</td>
<td>No event notification</td>
<td>No event notification</td>
<td>Event notification</td>
</tr>
<tr>
<td>Supports multi-sessions</td>
<td>Need changes to support multi-sessions</td>
<td>Supports multi-sessions</td>
<td>Supports multi-sessions</td>
</tr>
<tr>
<td>Reliable with TCP</td>
<td>Not reliable because no support of exactly ones</td>
<td>Not reliable because transport over RPC</td>
<td>Reliable</td>
</tr>
</tbody>
</table>

1) Only in the last release of CORBA some security issues are taken into account.

Seen from Java’s perspective: the communication channel will be set up using sockets, and at a higher level CORBA can support this. Since AXE-VM is also object oriented, the RPC and DCE are not investigated anymore.

The simple way to set up the connection has been chosen. Partly because of the items explained in the table but also because with Orbix-Corba, the Corba version that will have to be used, some employees at Ericsson Rijen have encountered some problems with multiple sessions to a server.
4.7 Unix-Inter Process Communication

Another approach to the question of how the AXE-VM and Java-VM can interwork is to look what IPC (Inter Process Communication) facilities Unix provides. For two processes to communicate with each other they must both agree to it and the OS (Operating System) must provide some facilities for the IPC. The IPC methods are each explained in a separate sub-section.

4.7.1 Pipes

Pipes provide a one way flow of data using the stream I/O model. It is a stream of bytes without interpretation by the system. For two way data flow, two pipes must be created for use in each direction.

Remarks about pipes:
- No record boundaries.
- No examination of the data.
- No names.
- In some releases pipes are implemented using a Unix domain socket.

4.7.2 Message queues

Using message queues more structured messages can be built. When the data consists of variable length messages, and when the reader wants to know where the message boundaries are, this mechanism is used. It uses keys to identify the queue on both sides. The pathnames are converted into the same key. All messages are stored in the kernel and have a msqid (message queue identifier). Every message on the queue has a long integer type with the length of the data portion of the message data. A message can be read and or written by the owner, group and world, with the msgget system call. Message queues allow multiple processes to multiplex messages into a single queue, due to having a type associated with each message.

4.7.3 Semaphores

Semaphores are a synchronisation primitive. As a form of IPC they are not used for exchanging large amounts of data, but to let multiple processes synchronise their operations. It uses keys to identify it on both sides. The pathnames are converted onto the same key. Mutual exclusion while accessing a semaphore must be strictly enforced by the operating system (kernel).

4.7.4 Shared memory

Shared memory let two or more processes share a memory segment. Both processes have, read, write, and read/write access to a certain memory segment. The exchanged data is not stored in the kernel as is the case with the other described methods. The synchronisation problems are user solved. This can easily be done using semaphores. It uses keys to identify it on both sides. The pathnames are converted onto the same key.

The advantage of shared memory is that the distributed information does not go through the kernel. The kernel is however involved when semaphores are used for synchronisation issues. If the access is denied for a block, the process goes to sleep, and the kernel can allocate the CPU to other processes that are ready to run.

4.7.5 Sockets

One type of sockets is the Unix domain socket and is used for IPC between processes on a single system. They use the same protocol as described earlier in this chapter, only the communication is not set up to another Internet or host address but to a port number.

Now that the Unix IPC mechanisms have been described it is time to investigate which ones are of interest for the problem described earlier in this report. The two possible communication principles need further investigation. These are the socket and shared memory principles. Sockets have also been chosen as the general network connection method. Shared memory has the advantage of less overhead and no duplication of the data in the kernel, and therefore is very well suited for further investigation.
4.8 A choice to make
The first remarkable aspect is that both the common and the Unix networking facilities support sockets.

Shared memory is unfortunately not suitable for communication with Java, because Java has been shielded for direct memory access. Even at Java-VM level no direct memory can be allocated. The only way is to write your own instance of the Java-VM to allow direct memory access. The last approach will cost a lot of time and one of the design principles of Java is to shield memory allocation and memory access for the user.

The stream Socket variant of the different socket approaches has been chosen to set up the connection between Java and the AXE-VM. A summary of the reasons why is described here:

- Stream sockets are a connection oriented approach as a network connection. Partly therefore it is a reliable way to exchange information.
- It is the common way for both Unix and Java to set up a connection with other processes or servers.
- It supports a multithreaded environment.
- In the future easily extendible for communication at a higher platform.

4.9 Summary
In this chapter a decision has been made which communication mechanism is used for the communication link between the AXE-VM and Java-VM. Different IPC mechanisms are described. The advantages and disadvantages of the different discussed methods is provided.

The AXE-VM and Java-VM use the TCP/IP protocol to transfer data over the network, since this protocol has also been used for the communication between the AXE-10 and the AXE-VM.

The created connection uses the Client-Server computing mechanism. The client component issues a service request and the server processes this request.

Of all the investigated TCP/IP- API's to communicate over the network and the standard distributed systems that can be created on top of it, the stream socket mechanism has been chosen as interfacing mechanism.
5. Set up the connection between the AXE-VM and Java-VM

This chapter provides design decisions of the socket communication interface between the AXE-VM and Java-VM. It will be defined and initialised. Data conversions and a protocol have been created to enable transparent data communication at both sides.

5.1 Design decisions

To actually set up the connection between the AXE-VM and Java-VM, and let the sent data be defined at both sides, some design decisions will have to be made first:

1. Where and how in the processes is the socket interface declared?
2. At what level in the AXE-VM does Java-VM communicate to?
3. How can this socket interface be initialised?
4. How can the data actually be sent and received?
5. In what format is the data sent?

In this chapter all these design questions will be discussed and answered.

5.2 Socket Interface

In the previous chapter Sockets has been chosen as a communication mechanism, this section deals with the question where in the AXE-VM process and how many of these socket connections will be set up. Each approach discussed here has its own advantages and disadvantages. The most important design issues are generality, reusability and standardisation. The different discussed approaches are:

1. **Set up a communication link for every remote request.** Every time a process invokes a remote server, a communication link is set up, and released again after completion of each request. This approach induces a lot of overhead because for every service request the system sets up a communication link. The supported socket implementation classes available in the AXE-VM uses this principle.

2. **Set up a communication link for every Java-block to all AXE-VM block’s it is able to communicate to.** In this way a socket interface is set up for each possible communication link. A disadvantage is that every block that wants to invoke Java has to interface with it.

3. **A combination of approach 1 and 2.** Use approach 2 for frequently used communication lines and 1 for rarely used ones. This method causes less overhead than approach 1.

4. **A more centralised method is to set up only one communication link between the AXE-VM and Java-VM.** An advantage of this approach is that the conversion between the two message formats is done at a central place and routed separately in both the AXE-VM and Java. This connection method is general, has only little overhead and is a fast mechanism to invoke a remote server. A major disadvantage with this approach is that two service requests that wants to invoke Java with different priority are handled one at a time in one FIFO-buffer.

5. **Set up a communication link between AXE-VM and Java-VM for each level of priority.** This approach overcomes a major disadvantage of approach 4 while the advantage of a central, general and fast communication principle remains.

The approach that sets up a communication link between the AXE-VM and Java for different levels of priority (Approach 5) has been chosen to set up the socket connection. The structure is similar to the way data communication is internally handled in the AXE-VM; Signals with the same priority are threaded in the same way and are buffered in the appropriate FIFO queue. This approach causes little overhead. Ones the sockets have been initialised, they are not released anymore.
Introduced are 4 levels of priority and sockets will be used for one way communication, so eight sockets are needed: Four sockets for communication to Java-VM, and four for the communication back to the AXE-VM (see Figure 5-1). Another advantage of this approach is that it is centralised. With this general, centralised approach, the amount of work and the level of processing issues at the remote server is scaleable. In the beginning only some functional blocks are implemented in Java and the result is passed back to the AXE-VM.

![Figure 5-1: Socket interface between the AXE-VM and Java-VM.](image)

**5.3 Control issues at the AXE-VM side about the chosen mechanism**

The communication mechanism chosen in the previous section uses one way communication. When sending data to the remote side, it loses the control over the service request if no mechanism is provided to handle the control. To overcome this drawback, a buffer that stores a service request identifier, and a time stamp in which the service request must be executed is designed. The design mechanism of this approach is:

- Give each service request a unique identifier. Create a 64K counter for this purpose.
- Store the counter, the absolute creation time, and a relative execution time in a buffer.
- Send the service request identifier to Java together with all other necessary information.
- The AXE-VM removes the service request from the buffer when Java sends back the service request identifier.
- If the service request is not returned after the in the buffer specified time stamp, it generates a timeout, and the error handling takes care of further handling.
5.4 Levels of Communication

This section investigates where and at what level Java and the AXE-VM communicate to each other. Three levels of communication have been examined; Communication through the AXE-VM platform with signals, communication at application level, and direct communication with the SSI (Service Script Interpreter). Figure 5-2 illustrates these three levels of communication. After examination of the different approaches a mechanism is chosen.

5.4.1 Communication at system level

Communication at system level means that the AXE-VM platform routes the signal (see Figure 5-2, line 1). Every block that wants to invoke Java-VM will have to notify this at the AXE-VM platform. Two possible ways to communicate with Java-VM at system level exist:

1. **Signals are routed to a special simblock that takes care of invoking Java.** Signals have been explained in section 2.4.5. Since the blocks executed in Java are part of another process, the AXE-VM signals has no meaning in Java. Either a class in Java must be written to understand this or a protocol between the simblock that takes care of the routing to Java and a Java-class that receives this data must be set up.

2. **In the signal a processor identifier is available.** All messages routed to Java can be identified with this processor identifier. The needed information for further routing in Java must also be stored in this signal.

The block that invokes another block generates the signal. This signal contains the block name the signal is routed to. A major drawback is that all blocks that want to communicate to Java-VM must be updated for this reason.

5.4.2 Communication at application level

Communication at application level means that different blocks communicate directly with each other, without invoking the system level (see Figure 5-2, line 2). In the AXE-VM two blocks written in C++
can communicate with each other without invoking the AXE-VM at system level. Communication at application level can be seen as a method invocation. Communication at application level means that an AXE-VM block directly invokes Java, without a routing through the AXE-VM platform.

### 5.4.3 Communication with the Service Script Interpreter

As mentioned in section 2.3, a concatenation of Service components called SIB's implements IN services. The SSI (Service Script Interpreter) is the environment in which the CT's operate. The SSI environment has already been explained in section 2.3.3. The CT data modules provide all the service subscription data. When designing a CT, the way the physical memory structure of the SSI maps on the data structure of the CT must be specified in the SSI environment.

A CT can be used as a way to communicate with Java. At the Java side a communication object must be present to take care of the communication to and from the SSI environment (see Figure 5-2, line 3). This special CT will be able to communicate with the system just like other CT's. A way to do so is to pass a standard set of CID (Call Instance Data) to the Java side. When the Java side gives control back to the SSI, it returns a possible changed set of CID parameters. The CT does not directly invoke Java, but is routed through a SimBlock in the AXE-VM.

The CDK (Control-type Development Kit) is a kit to design a CT in 100% C++ and still make it a 100% CT from SCF point of view [CDK]. Since IN services are built up as several CT's joined together in a specified way, this CDK can be used as a gateway between the AXE-VM and IN-services in the SSI.

The CDK has standard classes to subscribe a CT to the SCF, classes to store and retrieve data and a class to convert a register into a string. For more information about the CDK the reader is referred to [CDK].

### 5.4.4 Choose the communication level

Communication at application level will not be used for the following reasons:

1. The system level takes care of the routing and communication between blocks in the AXE-VM.
2. It's more general to set up a communication protocol at system level.

A major disadvantage of the communication at system level is that every block that wants to invoke Java must be updated.

A disadvantage of direct communication with the SSI is that Java can only be used for IN services. Since this report is about implementing IN services in a Java environment, the choice to communicate directly to the SSI environment only sets limitations to the generality of the communication protocol between the AXE-VM and Java-VM, but not for the scope of this report.

The approach that invokes the SSI environment will be used. The disadvantage of only using IN services lies outside the scope of this project. An overview of how the communication between the two processes looks like is illustrated in Figure 5-3.
5.5 Standard AXE-VM networking capabilities

In the AXE-VM an IPC (Inter Process Communication) protocol has been set up for socket communication. The AXE-VM IPC is explained in [IPC]. Five classes are supported:

1. **Class INetSocketServer.** This class supports socket servers creation in the AXE-VM. This class deletes the socket after execution in the ‘accept’ function. The socket connection may not be released and therefore the ‘accept’ function can not be reused.

2. **Class INetsocketClient.** This class supports socket clients creation in the AXE-VM.

3. **Class INetSocket.** This class is the base class for both the INetSocketServer and the INetsocketClient class. At lower level the INetSocket class uses the Unix socket class as described in sections 4.4.1 and 4.7.

4. **Class SocketReceiver.** This class has a function that forwards the data. Since the data will have to be converted and forwarded to the SSI environment, this function is not reused. This class uses an instance of class INetSocket at constructor time.

5. **Class SocketSender.** This class has a function that sends a character string to the remote process to whom the socket connects. This class also uses an instance of class INetSocket at constructor time.

5.6 Data sending at a lower level

The way the data is sent between the ToJava simblock and the SocketServer/SocketClient is explained in this section. It is illustrated in Figure 5-4.

![Figure 5-3: Communication between the AXE-VM and Java-VM.](image)

![Figure 5-4: Data conversions.](image)
The data format in which the data is sent is byte format. This means that conversion mechanisms must be present at both sides.

At the AXE-VM side the function that takes care of data sending carries an argument of type DataContainer. This means that this data has to be converted into DataContainer format. Figure 5-5 describes this format.

```c
typedef struct
{
    char* theData;
    unsigned long theSize;
} DataContainer;
```

*Figure 5-5: DataContainer.*

At the Java side standard classes are available to convert a byte-stream into other data formats, so no extra conversion measurements have to be taken.

### 5.7 Initialisation

At initialisation time the socket connections are created so the two processes can communicate to each other. The standard AXE-VM socket implementation is reused as much as possible. Unix Sockets can also be used. However to have two different approaches for the same kind of data transmission is redundant. It is advisable to reuse the existing software written for the socket communication in the AXE-VM.

A special simblock takes care of the communication to Java at start-up time. It has to perform the following actions:

1. Construct the cdk controltype classes.
2. Set up the communication link with Java.

Followed procedure to initialise the connection with Java:

**Java:**
- Set up a socket server that listens to a predefined port number (port number 5000) for input.

**AXE-VM:**
- Create 4 instances of function ‘create’ in the class INetSocketServer. To create these instances also the RemoteIPAddress, and the port numbers to connect to must be available.
- Create a socket client at the AXE-VM side and send the used port numbers to Java.
- Wait for data back from the socket.

**Java:**
- Java reads the port numbers, creates 4 socket clients and connects them to the sent port numbers.
- Java creates 4 socket servers to arbitrary free port numbers.
- Determine the port numbers where the Java socket servers listen to, and send them over the socket back to the AXE-VM.
- Java creates four threads to listen for incoming messages from the socket server.

**AXE-VM:**
- Determine the port numbers where Java has set up servers to, out of the received information over the socket.
- Create 4 instances of function create in the class INetSocketClient. This function also connects the clients to the Java servers.
- Accept the connection for the Java clients.
5.7 Initialisation

Figure 5-6 gives an illustration of the explained principle.

**AXE-VM**

Load user SimBlocks
This loads
CdkKSimblockToJava

Create the Socket
Connection with Java

Create 4 Socket
Servers

Create socket client for
initialisation

sentPortnrs

Create 4 Socket
Clients

Create 4 Socket
Servers

getPortnrs

sentPortnrs

Create 4 Socket
Clients

accept 4 socket
servers

**Java**

Create socket server for
initialisation

getPortnrs

Create 4 Socket
Clients

Create 4 Socket
Servers

sentPortnrs

Create 4 threads
that listen for
incoming calls

**CdkSimBlockToJava** automatically inserts a
CdkControlTypeToJava which constructs the
SocketToJava class

*Figure 5-6: Initialisation.*
5.8 Data sending and receiving
The initialisation of the socket connection has been established. The following step is to support the sending and receiving of data over this socket connection. This section is divided into two parts. One describes the socket sending and receiving at the AXE-VM side. The second describes this at the Java side.

5.8.1 Data sending and receiving at the AXE-VM side
Data sending is already supported in the available SocketSender class. According to the priority of the service request it routes to the corresponding SocketSender.

Data receiving is supported by the SocketReceiver, but this standard method will not be re-used. The reason for this is that the data is automatically forwarded. Socket receiver has to perform several tasks that execute subsequently:

- Detect an executed service request from the remote side.
- Remove the service request ID from the buffer, and check for time outs.
- Translate the data stream into understandable information and invoke depending on this result the right methods.

To detect a returned service request automatically is not supported in the AXE-VM or C++. Different ways to detect a returned service request from Java have been examined:

1. Create a fork that listens at the predefined port numbers. A fork has a lot of overhead because it completely copies the process that was executing.

2. Generate an interrupt to tell the AXE-VM that there is an executed service request, waiting at the socket server. Interrupts can be nested in this approach, and the AXE-VM loses some control.

3. Check periodically for pending data streams at the socket server.

For the time being, the receiving of data is not implemented yet.

5.8.2 Data sending and receiving at the Java side
Data sending to and receiving from the socket uses standard functions in the socket class in the Java.net package of the JDK. Functions to convert the data into a byte stream and to convert a byte stream into the data format are available in the InputStream/OutputStream classes.

5.9 Socket implementation details at the AXE-VM side
Class SocketToJava implements the socket connection and the sending of data to Java. The constructor of this class takes care of the initialisation of the socket interface. This is described in sub-section 5.9.1. Sub-section 5.9.2 handles the sending of the data to Java.

5.9.1 Initialisation of the socket interface
Class SocketToJava automatically constructs when the CdkControltypeToJava is inserted into the CdkSimBlockToJava. The constructor of the CdkSimblockToJava is called at start-up time if and only if there is not already an instance of this class available. The interface of the CdkSimBlockToJava is shown in Figure 5-7.

```java
CdkSimBlockToJava::CdkSimBlockToJava() :
    CdkSimBlockControltype() {
    theSimBlockExt.setBlockName("TOJAVA");
    insert(new CdkControltypeToJava());
}
```

*Figure 5-7: CdkSimBlockToJava constructor.*
5.9 Socket implementation details at the AXE-VM side

The constructor invokes several functions that all together perform the task as described in section 5.6.

The ToJava simblock is especially designed to route the service requests to Java. Without a communication link there is nowhere to route to, and therefore the connection is initialised at start-up time. An illustration of what the class structure looks like is given in Figure 5-8.

![Diagram of class hierarchy to set up the communication link.](image)

Figure 5-8: Class hierarchy to set up the communication link.

The constructor of class SocketToJava subsequently handles the following steps:

1. Set the local and the remote Internet addresses in a predefined format.
2. Set up 4 socket server connections.
3. Set up a socket client connection with Java and send the port numbers that the servers listen to, to Java.
4. Wait until a byte stream from Java has returned. This byte stream contains the port numbers to where Java has set up 4 socket servers.
5. Set up 4 socket clients that connect to these port numbers.
6. Accept the 4 socket servers after Java has connected 4 clients to the servers.

Each of these steps are explained in a separate sub-section.

5.9.1.1 Internet addresses

The definition of the used Internet socket address structure is shown in Figure 5-9.

```c
struct in_addr {
    u_long     s_addr    // 32-bit netid/hostid
};

struct sockaddr_in {
    short     sin_family // AF-INET
    u_short   sin_port   // 16 bit port number
    struct in_addr sin_addr // see in_addr
};
```

Figure 5-9: Socket Internet address structure.

The local internet address has been defined as htonl(INADDR_ANY), which converts the local host address to the network long integer, needed in the in_addr structure.

The remote Internet address has been set to htonl("164.84.81.11"), which converts the remote host named "sjors" with Internet address "164.48.81.11" to the network long integer, needed in the in_addr structure. If the environment is set up at another server, the internet addresses must be changed here.
5.9.1.2 Set up the server connection

Method SetupServerConnection creates a socket server. It carries two arguments:

1. An instance of type InetSocketServer.
2. A port number to connect to.

If the process fails to connect to the predefined port number, it tries to connect to another port number. In case of a failure, it tries to connect to the port number +4 (since four socket servers will be set up). The return value of this method is the port number to where the socket ultimately connects.

5.9.1.3 Create a socket client and send the port numbers to Java-VM

Java has set up a server that listens at a predefined port for a client to connect to this server. The AXE-VM creates such a client. This socket exists for three reasons:

1. Synchronise the initialisation part of both processes.
2. Let the remote side know to which port numbers these servers must listen.
3. In the future this socket can be used for error handling.

The AXE-VM has created four socket servers and connected them to a certain port number. Since Java needs these port numbers, these are sent to Java-VM over the already created socket connection. The existing socket connection will not be used for execution of service requests and is therefore a normal Unix socket connection. The port numbers are converted in the appropriate format before writing them to the remote side. In case of an error this is reported.

5.9.1.4 Receive port numbers from Java

The port numbers Java is listening to are the ones to where the AXE-VM has to connect. The AXE-VM receives these port numbers through the available socket connection. The read function of the socket class waits for these incoming port numbers. The incoming port numbers will first have to be converted from the character string into integers.

C++ provides a standard function called sscanf to convert the character string into other predefined formats.

5.9.1.5 Set up the client connection

The clients are fully implemented as instances of type INetSocketClient. Class INetSocketClient has already been described in section 5.5.

5.9.1.6 Accept the server sockets

The ‘doJob’ function in this class accepts the socket first and after execution closes the socket connection. Method ‘accept’ in the created INetSocketServer can therefore not be used for the ‘accept’ function. Since the INetSocketServer class uses Unix sockets at a lower level to accept the connection this connection uses the Unix socket::accept command.

5.9.2 Sending data to Java

The SocketToJava class implements a method to send the data to Java. This method is called SendData. It actually sends the data over the created socket connection to Java. It determines the priority of the service request and routes it to the remote side. As default the lowest priority will be given to the service request, since otherwise high priority time critical service requests can be slowed down.

5.10 Implementation details at the Java side

This section consists of two parts, one that explains the sockets initialisation and one to send and receive data over the socket connection. Each part is described in a separate sub-section.

5.10.1 Initialisation of the socket interface

At the Java side a class named SocketToAXEVM takes care of the socket connection with Java. This class handles the complete initialisation of the socket interface as described previously in this chapter.
5.11 Data protocol

The SocketToAXEVM constructor consists of the following steps:

1. Create a socket server that waits at a predefined port number (port 5000) for the AXE-VM to connect to this port number.
2. Accept the connection if the AXE-VM client connects to this port number.
3. Wait at the created port number for input. The input consists of four port numbers as a byte stream.
4. Invoke method ClientInitialisation. It carries the byte stream as an argument.
5. Invoke method ServerInitialisation.
6. Send the port numbers to where the server has set up the connection to the AXE-VM.
7. Create four threads that wait for incoming service requests. The threads that listen for incoming requests run until the Sockets are released.

Steps 4 and 5 need further explanation. These steps are both implemented in a separate method. The method that takes care of step 4 is described in sub-section 5.10.1.1, and the methods that handle step 5 in sub-section 5.10.1.2.

5.10.1.1 ClientInitialisation
Method ClientInitialisation carries one argument. The argument is of type Byte and contains the port numbers the client will have to set up the connection to in a predefined protocol. The three tasks subsequently performed in this method are:

1. Set the remote and the local address. The remote address has been set to "164.48.48.76", and the local host can be obtained by asking the local host.
2. Determine the port numbers out of the input parameter.
3. Create four instances of type Socket. These connect to identical port numbers on the remote host.

5.10.1.2 ServerInitialisation
Method ServerInitialisation implements the socket server initialisation at the Java side. The subsequently performed tasks in this method are:

1. Create four instances of type ServerSocket, that set up a socket server to random free port numbers.
2. Determine these port numbers. A standard function getLocalport in the ServerSocket class supported for this matter.
3. Convert the port numbers into a byte stream and send this to the AXE-VM, because the AXE-VM need these port numbers to connect to the clients. Method ConvertPortNumbersIntoByteStreams converts the port numbers into a byte stream
4. Wait until the AXE-VM side has connected 4 clients to the port numbers and accept these socket servers.

5.10.2 Data sending and receiving at the Java side
Four threads have been created to listen for incoming data. These threads react on incoming data, so it does not have to be checked periodically. Instances of class CallThread implement these threads. Incoming data is transformed into a service event and the service listeners will execute the service further.

After execution the return value, if there is one is sent to a thread that returns the data back to the AXE-VM. These threads are implemented as instances of class ClientThread.

5.11 Data protocol
The next step is to define a protocol that enables transparent data communication at both sides. The protocol must be simple, complete, general, and have as little overhead as possible.

5.11.1 Preparing the data protocol.
Before defining the protocol syntax, the needed data at the remote side must be investigated. The pieces of information that are sent in the data protocol are:
Parallel processes for telecommunication with Java

1. A 'UniqueKey' identification, to uniquely identify the service request.
2. A ServiceId that determines what service will have to be executed.
3. All the CID parameters needed for execution in Java. Send the used KoN, KoV and KoL identifiers together with the value pair wise.

5.11.2 The protocol syntax

This section describes the syntax of the protocol.

First Send the length of the message to be sent. Java uses this length to determine the amount of data that must be read from the socket. The length is defined as two characters representing the length. The length of 100 is more than sufficiently enough. If the length is smaller that 10, place a 0 before the number, since two characters will have to be sent according to the protocol.

The CID parameters consist of two pieces of information. A parameter identifier and the value. To uniquely identify the parameters at the Java side both pieces of information are sent. Place a special character (;) between the pairs, and place a special character (=) between the first and the second element of this pair. Only the actual needed variables are sent to the remote process.

To formally specify the protocol the EBNF syntax is used. The symbols of the EBNF syntax are explained in Table 5.11-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>Is defined to be</td>
</tr>
<tr>
<td></td>
<td>Alternatively</td>
</tr>
<tr>
<td>&lt;text&gt;</td>
<td>Non-terminal</td>
</tr>
<tr>
<td>&quot;text&quot;</td>
<td>literal</td>
</tr>
<tr>
<td>*</td>
<td>The preceding syntactic unit can be repeated zero or more times.</td>
</tr>
<tr>
<td>+</td>
<td>The preceding syntactic unit can be repeated one or more times.</td>
</tr>
<tr>
<td>{}</td>
<td>The enclosed syntactic units are grouped as a single syntactic unit.</td>
</tr>
<tr>
<td>[]</td>
<td>The enclosed syntactic unit is optional—may occur zero or more times.</td>
</tr>
</tbody>
</table>

The specification of the protocol according to the EBNF syntax is described in Figure 5-10.

```
<Protocol> ::=<integer>";"<ServiceRequestId>";"<ServiceID>";"{<Item>";"}*;
<ServiceId> ::=<integer>
<UniqueKey> ::=<integer>
<Item> ::="KoN"<integer>"="<Frame>
         | "KoV"<integer>"="16-bit_literal
         | "KoL"<integer>"="32-bit_literal
<Frame> ::=28_digits_literal
<integer> ::=0,1,2,3,....
```

*Figure 5-10: Specification of the protocol.*

5.12 Summary

This chapter defines and initialises the socket connection between the AXE-VM and Java-VM. It defines that 8 one-way socket connections are set up, so that data can be sent bi-directional with 4 levels of priority. The communication is set up at start up time and in a way that the SSI environment can directly invoke Java.

To enable transparent data communication at both sides, a protocol has been created on top of the communication link. The pieces of information in the protocol are subsequently:

1. The length of the data.
2. A service ID which uniquely tells Java what service to invoke.
3. A Service request identification. This tells the AXE-VM which outstanding service requests are still at the Java side.

4. All the needed parameters.
6. The general service creation environment

The created environment supports the sending of arbitrary variables and their values to Java for execution. The next step is to look at a general way to create new services on the remote side and the way the AXE-VM is involved. The goal is to create new services in Java without having any knowledge of the AXE-VM and the communication protocol to the Java-VM.

6.1 Design decisions

The purpose of this section is to explain the design decisions and give the reader an idea of how the environment will look like. It is not the purpose to give implementation details in this section. According to these design decisions the service creation environment has been built.

Design decisions:

1. Create an environment to create new services or service components in Java in a way that the existing Java code used in other services can be reused.

2. The created service or service component must be created in Java without having to recompile the AXE-VM. Several reasons exist for not recompiling the AXE-VM when new services or service components have been created at the Java side:
   - The programmer does not need to have any programming experience in C++, since the code of the AXE-VM remains the same.
   - The programmer does not have to know anything about the programming structure of the CDK and the used protocol to send the needed information to Java.
   - New services can be added during execution time of the AXE-VM.

The major problem to this goal is how the AXE-VM knows which service has been created, when to invoke that service and which parameters the other side needs. As a solution a script of MML (Man Machine Language) commands can be used to load the right commands into the Input Registers in the SSI environment. MML is a way the user can interact with the machine. Section 6.2 explains the script and the protocol to let the script be understandable at both sides.

6.2 Creation of a script

For reasons described in section 6.1 a script of MML commands will be used to load the Input Registers. These input registers tell the AXE-VM what variables to get out of the SSI and gives the ServiceID to send to Java. The MML script is also used to load the service in the SSL environment. Sub-section 6.2.1 explains the way the script is built up. A protocol of how the available information in the Input Registers is translated into understandable information at the AXE-VM is specified in sub-section 6.2.2.

6.2.1 The MML script to load the service script

The MML script loads the service script in the SSI environment, and sets the Input Registers. The Input Registers set the variables for service execution. Erlang, SMAS (Service Management Application System) or the TWR environment can load the script of MML commands in the SSI environment. As a user interface, all three environments may be used, but only the vm_interface class in Erlang and TWR have a direct interface with the AXE-VM and one of the two will therefore be used. Both have exactly the same interface with the AXE-VM, so it is up to the user which one to use.

The creation environment of class BasicCdkControtype supports the reading an writing in the Input Registers of the service request. The general idea is to set the Input Registers to let the CdkControtype know which CID is needed at the Java side, and what service should be executed.

The MML scripts consist of two parts the Service Logic and the Service Administrator part. The service logic part builds up the service according to a standard protocol and the Service Administrator.
The commands for the creation of the service logic are described in [1004]. A script of MML commando's representing a service, and a picture of what this looks like are given in section 8.3.2.5. Appendix B explains the used MML commands.

6.2.2 Protocol specification for the Input Registers

This sub-section is divided into two parts. One part to specify a general specification protocol to tell the environment which parameters should be sent to Java. This primary design specification protocol will not be implemented due to a limitation in the CDK environment. A secondary protocol describes the protocol specification that is implemented here to partly overcome these limitations.

6.2.2.1 Primary designed protocol specification for the input registers

The CT loads two pieces of information to execute the service:

1. A ServiceID. This ServiceID uniquely determines the kind of service that Java has to execute.
2. A piece of information that tells the CT the needed parameters at the Java side.

To send this information 8 16-bits input registers are available. The specification of the protocol of how to fill these input registers is illustrated in Figure 6-1.

![Figure 6-1: Primary design Input Register specification.](image)

Each variable uses an Input register that has the same protocol as IRR2. According to the number of needed variables, which is the sum of the number of input- and output parameters, the system determines the number of needed Input Registers to read.

The Id is a 2 bit identifier that determines the needed kind of parameter at the Java side. The kind of parameters are KoN, KoV or KoL. Table 6.3-1 gives the table of Id values and their meaning.

<table>
<thead>
<tr>
<th>Id</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>KoN</td>
</tr>
<tr>
<td>01</td>
<td>KoV</td>
</tr>
<tr>
<td>10</td>
<td>KoL</td>
</tr>
<tr>
<td>11</td>
<td>undefined</td>
</tr>
</tbody>
</table>

The value gives the instance stated at the value place of this kind of parameter. For example Id=KoN and value=2 means the Java side needs parameter KoN2.

6.2.2.2 Implemented protocol specification for the input registers

One major limitation set to the CDK-environment is that it is only possible to pass IR1 and IR2 to the CDK-environment. This means that using the specified protocol the service designers can only pass one parameter to Java. To extend this to two parameters IR2 only uses one byte to specify which CID is the Java side needs. Figure 6-2 gives an illustration of what this looks like.
6.3 Standard creation environment at the Java side

For each new service the service designer has to create a class. Three parts are distinguished when implementing the class that represents the service:

1. **Make sure that Java knows the meaning of every sent parameter to Java.** The representation in Java for the different kind of parameters sent to Java is not known. Some can best be implemented by a simple integer, while other can best have a class representation. Only a guideline is given here for the likely best implementation solution. Represent KoNs and all the operations that can be performed to these Numbers in a class. Represent KoLs as unsigned integers, and KoVs as unsigned short integers.

2. **Extend the pool of general public classes with reusable objects needed for this service execution.** The goal is to make these classes as general and reusable as possible. These classes are the SIB equivalent in the SSI environment.

3. **A part that builds up the service out of the different service parts in a predefined way.** This part is specific to this service.

For the routing to the right service with the right CID, two additional actions are taken:

1. Create a MML-script that loads the input registers.
2. Support the routing to the service.

A detailed description of the service creation environment is integrated in the operation environment. Chapter 7 explains this environment.

6.4 The call generator

As call generator the AXE-VM uses the user simblock Traffy. Traffy is the latest development in high performing Traffic generator for the IN2.2 environment [Traffy]. Traffy provides advanced capabilities to generate service requests towards the SSF and the SCF. It supports two basic types of traffic:

1. A burst of n calls.
2. Repeatedly send n calls in a specified interval. The calls can be divided fixed or random in the interval.

The way Traffy communicates to the environment is illustrated in Figure 6-3.
Parallel processes for telecommunication with Java

Figure 6.3: The call Generator.

Traffy uses the capabilities of INTSIM. INTSIM provides capabilities to simulate a SSF and SCF function. The essence of Traffy is the replaying of recorded traffic cases (recorded signal flows that were sent by INTSIM). Traffy provides some capabilities:

- Send a burst for a recorded traffic case. Type `tgburst [case=RTCld] num=NUM int=INT`. NUM and INT represent the number of times the record traffic case has to be replayed in the specified interval.
- Replay traffic case n times in a certain time interval.
- Replay multiple traffic cases at the same time.
- Get a report once with `tgreport` or periodic with `tgreport value=INTerval`.

To start a traffic intensity run for a recorded traffic case, type: `tgint [id=ID] [case=CASE] num=NUM int=INT [grow=NUM] [random] [maxtime=time]`. ID is the identifier of the traffic intensity, CASE is the recorded traffic case, GROW specifies the length of the first interval, maxtime gives the maximum time the execution lasts. If random is specified, the calls are divided randomly in the interval.

6.4.1 Creating the recorded traffic case

Before these traffic cases can be loaded, they must be created. To create a recorded traffic case the following actions must be taken:

- Prepare a MML file with INTSIM commands for the traffic case.
- Activate tracing of incoming and outgoing signals for the function block INTSIM.
- Execute the MML command file.
- Stop tracing of incoming to and outgoing signals from INTSIM.
- Save the signal log file under an application file.

The INTSIM-MML commands are described in [1103]. Some general ones are given here:

- PHSDI: This command stands for IN Protocol Handling Traffic Simulation Dialogue Initiate.
- PHSDE: This command stands for IN Protocol Handling Traffic Simulation Dialogue End.

Traffy uses a recorded signal trace of the INTSIM block to generate service requests. Problems have been encountered when creating the recorded traffy case, due to the complex signal tracing of the AXE-VM.

The problem cannot be solved easily, so a workaround has been chosen:
In an earlier release the signal tracing was much more straightforward, and that environment has been used to create the scripts. A disadvantage is that there is only a finite set of available service requests during test or demo time. This set is fixed.

Four scripts have been traced, and therefore only four different calls can be made in this demo environment. An overview of the call possibilities is given in Table 6.4-1.

<table>
<thead>
<tr>
<th>Callers number</th>
<th>Called number</th>
</tr>
</thead>
<tbody>
<tr>
<td>040-12345</td>
<td>1111</td>
</tr>
<tr>
<td>010-12345</td>
<td>2222</td>
</tr>
<tr>
<td>020-12345</td>
<td>3333</td>
</tr>
<tr>
<td>040-23456</td>
<td>4444</td>
</tr>
</tbody>
</table>

### 6.5 Summary

In this chapter a general service creation environment has been created. This environment is general and supports the creation of new services in Java without having to recompile the AXE-VM. A script of MML commands informs the SSI environment and the AXE-VM about the needed information at the Java side.

In this script the service notifies itself in the SSI environment and the Input Registers are set. The Input Registers contain information about which CID parameters are needed at the Java side and an identifier indicating the service to be executed in Java.

A call generator has been used to generate calls. It traces and records the incoming and outgoing signals. The recorded signal tracing file can be loaded into the user interface environment, to generate the calls.

A general note to the service creation environment: At the AXE-VM side, the protocol to let Java create new services is very general other remote processes can easily invoke the AXE-VM in the same way that Java does.
7. Creation of a service operation environment in Java

Service requests can be routed to Java and services created and executed. The next step is to create an environment where a customer can subscribe to a certain service, new services can be added, removed or updated without affecting the environment. Some requirements have been set up to conform to the environment, and some design decisions are made in this chapter. After creation, the environment is looked at from the service designer perspective.

7.1 Requirements
Some obligations to the service operation environment are:

- Add new services at run time.
- Install a new version of an existing service while the service is running.
- Provide the possibility to subscribe to a certain service.
- Provide the possibility to remove services.
- Handle more than one service requests concurrently.
- Make the environment robust and handle the error messages.

7.2 Design decisions
This section explains the design decisions for several sub-tasks. Each design decision takes of course the requirements into account. These sub-tasks are:

1. The execution of a service request.
2. The service class hierarchy.
3. The coupling between the service id and the actual class representing the service.
4. Some database functions.
5. The service event listener.
6. The things to do when adding, removing or updating a service.

All parts are explained in this section. For each part some design mechanisms and a design decision is made.

7.2.1 Execution of a service request
Incoming service requests enter the Java environment at a predefined port number. Each socket server creates a thread that listens at a port number for incoming requests. After service execution Java routes the result back to the AXE-VM. Each socket client creates a thread that listens for events indicating the service has been executed. The socket client sends a byte stream back to the AXE-VM containing the return values. Figure 7-1 illustrates this mechanism.

![Figure 7-1: Service request routing to Java.](image-url)
The remainder of this section handles the routing of a service request inside the Java service execution part. Two examined possible approaches are:

1. **Create a thread for each service request** and run that thread. Every thread has its own progress and service execution routing.

2. **Create a thread for each service** and let this thread react on service events to where it listens. If a service event occurs, it executes the service and listens for new requests that want to invoke this service. For each incoming message the service to be executed is determined and the message is turned into a service event. For every created service in Java a thread has been made at set-up time. Every service thread listens for pending events. Invoke after an event occurs the invoke function of that service. This invoke function actually executes the service. Translate the return values into a standard configuration that is understandable at the AXE-VM side and create a socket client event (see Figure 7-2).

Approach one has the advantage that it handles every request to a certain service concurrently.

Disadvantages of the first approach are the synchronisation problem and the thread overhead. Problems with synchronisation occur when more than one thread operates on the same data. This problem must be dealt with very carefully, since every service request has its own data part. The thread overhead is definitely non-negligible. To create a thread for each service request has impact on the overall performance of the system.

A design decision is to create a thread for each service. The drawback of non-concurrent execution of the same service can be eliminated and does definitely not outweigh the drawbacks of the first approach. An illustration of what the routing of the service request looks like is shown in Figure 7-2.

![Figure 7-2: Service request routing in Java.](image)

To create more threads that run a particular service eliminates non-concurrent execution of the same service. The latter is particularly important when the service execution takes some time and when waiting events occur during execution of the service thread. Sub-tasks should also be executed in a separate thread.

For each created service, it will have to be examined how many threads should run the service. For the scope of the design decisions in the service creation and operation environment the problem of how many threads will have to be created, is not relevant. It can be seen as only one thread for each service has been created. The service designer will have to take this into account.
7.2 Design decisions

As mentioned before class ServerThread takes care of the incoming service requests and routes these requests to the right service. In section 7.2.5 Service event-listener mechanism Class Server thread will be explained in more detail.

7.2.2 Service class hierarchy

The service creation and operation environment creates an abstract 'BasicService' class. All the service classes are derived from this abstract class, and have its own implementation of it. An abstract class is a class that is only partially implemented. It contains one or more abstract methods. Abstract methods are declared but left un-implemented.

The BasicService class contains:

• The 'run' method. After creation of a thread, a 'start' method actually activates the thread. The 'start' method executes the 'run' method. The 'run' method of the 'BasicService' class overrides the 'run' methods of the thread class. This method subsequently does the following:
  1. Listen for service events.
  2. Get the message out of the buffer.
  3. Execute the service by calling the InvokeService method. As a parameter the message representing the service request is passed.
  4. Listens for the next service event.

• An instance of type Vector, which acts as a buffer. Each service has an instance of this vector to store the service events. Instances of type Vector can grow according the number of pending service requests. An instance of type Vector(100,10) implements the buffer. This means that the buffer has storage for 100 service requests and when the buffer is about to be overwritten it extends the buffer with 10 service requests. Requests are added at the end of the list and removed from the beginning, which makes it a FIFO queue.

• The InvokeService method. This method calls two other methods:
  1. Translateparameters(message)
  2. Invoke()

These methods are both explained in this section.

• The TranslateParameters method. This method translates the incoming message (representing a service request) into a number of parameters and their values. Every time a parameter-value combination is found, method ConvertCallIntoVariable is called.

• The ConvertCallIntoVariable method. This method translates the input parameter into a variable (CallData) and a value (Parvalue). The input parameter has the format: <CallData>=<Parvalue> With CallData={KoNx,KoLx,KoVx} x integer, and Parvalue the value it represents. After splitting the input parameter, abstract method ConvertParameterIntoVariable gives the parameter a meaning, by connecting it to a variable name.

• Abstract methods. These methods indicate that every created service will have to implement these methods in a service dependent way. The service creator is obliged to implement them all. If not all the abstract methods are declared, it rejects the service.

These abstract methods are:

  1. Invoke. The invoke method is the place where the actual service execution takes place. It should implement the complete service request routing and handling.

  2. SubscribeToService. The subscribe method is the place where the sequence of performed actions is implemented, when subscribing to a service.
3. **ShowResults.** The ShowResults method describes the actions that must be performed to show the results of a certain subscriber on screen.

4. **PrepareOutputStream.** This method creates the outputstream to send back to the AXE-VM. It represents the result parameters or an error message. Since every service generated its output parameters, this has to be done in a service dependent way.

5. **ConvertParameterIntoVariable.** This method carries two input parameters. One representing the parameter and one representing its parameter value. The method body connects the variable representing the parameter to the parameter value.

The obliged abstract methods are a minimal requirement set for the service designer. The designer can add its own methods and integrate them into the environment.

### 7.2.3 Coupling between the ServiceID and the actual service

Since newly created services must be inserted without affecting the environment, the coupling between the ID and the corresponding service class is done dynamically. Java provides a class called Hashtable that provides this functionality. An instance of type Hashtable stores the services and their ID's. This Hashtable must have a unique key (in this case the ServiceID) and maps this key to a value. This value can be any object. This value is a reference to the corresponding service name, which means that the right implementation of the abstract class MyService is directly invoked.

Figure 7-4 gives an illustration of the service class hierarchy and the interworking with the Hashtable.

![Figure 7-3: The conversion between the service ID and the corresponding class.](image)

The figure illustrates the references to the services. Every time the ServiceID has been routed to the Hashtable, it returns a reference to the service that should be invoked. This reference is cast to the MyService class and the service request is automatically routed to the right service.

### 7.2.4 Created database functions at set-up time

Two types of database functions are available in the service creation and operation environment.

- **ServiceList.**
  This ServiceList is an instance of class List. Class List is available in the JDK. It represents an indexed list of String values, where elements can be added, selected, removed, updated etc. A complete API of class 'List' is shown in [List]. The ServiceList stores all the serviceID-service name combinations in a String, representing an element of type list. The reason for storing these combinations in a String is, that this is the only text format in which elements can be stored in a List. ServiceList is visible for the user, to know the available services. ServiceList is also used for incoming service requests to test whether or not the service request is a valid one.

- **Numberlist.**
7.2 Design decisions

NumberList is an instance of type 'numberlist'. Class 'numberlist' extends class 'List'. This means that it can invoke all the available methods in class List and some methods are added to the NumberList. A complete set of methods not available in the List class and their meaning is given in Appendix A. NumberList stores all the used numbers in the environment. An element of type 'numberlist' represents 4 pieces of data joined together in one String. These four pieces of data are:

1. The number.
2. The meaning of that number.
3. The subscriber that uses that number.
4. The number of times the number has been dialled.

Both classes are used by only one instance that has been constructed in the 'management' class and should be invoked from this point. The 'management' class is the class invoked by the main method at start up time. If a service want to get the meaning of a known number he has to give the command: Management.NumberList.getMeaning(<number>).

7.2.5 Service event-listener mechanism

The service event and listener mechanism has been implemented as a buffer, to which the events can write, and from which the listeners can read. Figure 7-4 shows this mechanism.

![Figure 7-4: Service Event-Listener mechanism.](image)

As can be seen from this figure, the buffer synchronises the two asynchronous threads.

For each created service the mechanism to store pending service requests in a buffer is available. Every service class creates a thread that listens to the buffer for incoming events.

The thread that listens for incoming service requests stores this request into the buffer. After the message has been stored into the right buffer, 'ServerThread' listens for the next incoming service request.

'ServerThread' subsequently does the following:

1. Determine the length of the message representing the service request.
2. Create a bytestream of that length and get the inputstream from the socket.
3. Determine the service ID out of the service request.
4. Check whether the service represented by the service ID is operational at the Java side.
5. Route the Service ID to the Hashtable that determines the service.
6. Cast the returned Object class to the BasicService class.
7. Stores the service event message in the buffer representing the pending service requests for that service. The given command is the <Vector>.addElement command.

'ServiceThread' subsequently does the following:
1. Listen for incoming service requests with the `<Vector>.getfirst()` function.
2. Invoke the `InvokeService` method of this service. In this method the actual service execution has been implemented. The `InvokeService` method also takes care of the return message to the AXE-VM.
3. Remove the first element in the buffer, since this is not automatically done when getting it.
4. Listen for the next incoming service request.

### 7.2.6 Adding services

When adding a service to the environment some rules will have to be taken into account:

- To add a service at run time the class that represents the service should be added without recompiling the entire environment.
- The service will have to notify itself to the environment, and must be ready to receive messages.

The following actions are required in the service operation environment when adding new services in the environment at run time:

1. Define the service name.
2. Create an object of that name.
3. Cast the object to the `BasicService` abstract class.
4. Create an instance of this class, and start the service thread.
5. Add the newly created service name in the Hashtable.
6. Add the newly created service name and Hashtable key in the `ServiceList`.

### 7.2.7 Removing Services

When removing a service from the environment some rules should be taken into account:

- Pending service events must be executed before the environment removes the service. New service requests entering the environment must be sent back to the AXE-VM with an error message.
- Remove all the subscribers to the service properly.

The following actions are required in the service management environment when removing services from the environment at run time:

1. Remove the service from the `ServiceList` that represents all the available services. By removing this element form the `ServiceList` the incoming message cannot be routed anymore to the service. The request is directly sent back to the AXE-VM with an error message, saying the service ID is not available.
2. Empty the buffer with pending service events.
3. Inform the subscribers that the service is not in use any more. And give them some time to look at the final results.
4. Remove the service id from the Hashtable after a predefined period of time.

Garbage collection is automatically done in Java. For this reason the environment does not have to remove the service class.

### 7.2.8 Installing a new version of an already existing service

When installing a new version of an existing service some requirements should be taken into account:

- All the functionalities that were available in the old version must still be available in the new version.
- No loss of service requests may occur during installation of the new version.
- The service must still be operational during installation of a new version.
According to these requirements a set of actions to be taken subsequently in the operation environment has been set up:

1. Install the new version of an existing service as an extension of the existing one.
2. Start the thread that handles all requests to this service.
3. Link the ServiceId to the class name of this new service.

### 7.3 The flow of a service request

Some design decisions have been made in the previous section, this one gives an overview of how the service request routing is handled in more detail. Figure 7-5 gives an overview of this. The arrows give the flow of a service request.

![Flow of a service request in the service creation and operation environment.](image)

**Figure 7-5:** Flow of a service request in the service creation and operation environment.

As can be seen from this figure the flow consists of several steps that are performed subsequently:

1. An incoming message from the remote host is detected.
2. Determination of the service to be executed, and check whether that service is available.
3. Service is available. Sent the service request to the Hashtable.
3'. Service is not available at the Java side. Sent an error message to the ClientThread.
4. Sent the message to the right buffer.
5. Sent the result to the client thread after execution of the service thread.
6. Sent the result back to the remote host in a predefined way.

Figure 7-6 illustrates the routing of the service request inside the service thread.
Parallel processes for telecommunication with Java

Class myService is an abstract class where all services are derived from.

Figure 7-6: Routing of the service request inside the Service thread.

The figure shows the interworking of the abstract ‘BasicService’ class and the invoked ‘Service’ class. The steps that will have to be performed subsequently are according to the arrow numbers.

7.4 The environment from the service designer point of view

In section 7.2 the design decisions from the service creation and operation point of view have been examined. In this section we look at it from the service designer point of view.

The design decisions from the service designer point of view are:

1. Creating a general set of functions.
2. The steps to be taken for the creation of a new service.
3. The way an already existing service can be updated.

Each of these design decisions is explained in this section.

7.4.1 Create a set of general functions

The concept of IN and of object oriented computer languages is the reuse of already existing functions and methods. Two ways for offering the environment these general functions are considered:

1. Create a list of standard functions, where the designer can look into to see if he can reuse some functionality’s. Static methods in the service classes implement these reusable functions. Another function can explicitly call the method in that service class.

2. Create some kind of library, where all the general functionality’s are implemented as classes. All services can in this approach use a standard and general pool where the standard reusable functions are stored.

Approach two will be used, because it is more general and widely used. The services will be created in a standard directory. This directory is called <classpath>\Services. The library is implemented in the directory classpath\Services\ToolBox. In every created service class that uses one of the ToolBox classes the path after classpath must be imported. When the specified class is loaded at run time of the environment the specified ToolBox is automatically linked to created service. Figure 7-7 illustrates this directory hierarchy.
7.5 User interface

![Figure 7-7: The directory structure of the service creation environment.](image)

### 7.4.2 Creation of a new service
As described in sub-section 7.2.2, all service classes are derived from an abstract class called BasicService. This section deals with the service designers point of view to the service creation and operation environment. What methods must be available in the service class. The parts of the BasicService class that will have to be implemented are:

- **The ‘invoke’ method.** The invoke method is the actual service execution part. It can either construct classes in the standard Library or create its own methods. When the designer creates his own methods, he always has to consider to add this functionality into the Library.

- **The ‘subscribe’ method.** If the possibility exists to subscribe a person, company, or institution to the service, implement the actions that are necessary here. If the service can not be subscribed to, the method body must show that the service can not be subscribed to.

- **The PrepareOutputStream method.** Currently the Java process can either sent an OK or not OK message, but for the generality of the environment it is done in a service dependent way.

- **ShowResults.** This method implements, the sequence of actions that must be performed to show the service results of a certain subscriber. If the results of the service should not be visible for the subscriber, this method creates an error frame indication that it is not allowed to show the results of that service.

- **ConvertParameterIntoVariable.** This method carries two input parameters. One representing the parameter and one representing that parameter value. The method body connects the variable representing the parameter to the parameter value.

### 7.4.3 Updating an already existing service
Service updating is implemented as extension to the old version of the service, all methods and even the invoke function can be reused in this way. The designer has to decide which part of the old version is still usable in the new one.

### 7.5 User interface
From the, in the previous sections designed service operation environment, an easy to use user interface has been made. With this interface the following actions can be taken:

1. Add services.
2. Remove services.
3. Install a new version of an already existing service.
4. Subscription to a certain service.
5. Show the results of a subscriber to a service.

The user interface also lists the created databases (NumberList, and ServiceList) on screen. At initialisation time the user interface is started up and all the actions that the service operation environment have to perform, are performed using this user interface. The begin frame of the user interface is illustrated in Figure 7-8.

![User Interface Screenshot](image)

*Figure 7-8: Start frame when invoking Java.*

### 7.6 Summary

In this chapter a service operation environment has been created. Design decisions are defined of what the environment must look like, according to the requirements.

The routing of a service request inside the Java side follows a predefined road:

1. It enters at the Java server socket.
2. Routed to a dynamic buffer coupled to the service that has to be executed.
3. Service request is executed.
4. Service executed message is created.
5. This message is sent to a dynamic buffer coupled to the socket client that sends the result back.

A common way to add, remove, subscribe to, and update services into the environment has been given. An user interface to easily integrate these items into the environment is supported.

Attention has been paid to look at the environment from the designer point of view. The designer has to take standard rules into account when designing a new service.
8. Creation of a service in Java

A standard environment to create and manage services has been created in Java. The next step is to test the environment for correctness, robustness and performance. General ways to run services are discussed, and a service called televoting service has been created. All the taken design steps to create such a service are explained. This chapter ends with some drawn conclusions about the environment.

8.1 New service creation design steps

This section gives an overview of the steps that are needed to create a new service. Throughout the previous chapters all the steps that should be taken, are already mentioned. An overview is given here:

1. Look in the ToolBox environment for reusable classes.
2. Create as much of the things to do in reusable classes and implement these classes in the ToolBox environment.
3. Determine the number of threads that runs the service.
4. Implement the abstract methods of the service you want to create.
5. Create a script of MML commands.
6. Insert the service in the service Creation and management environment.

When following these steps, the service is inserted into the environment and subscribers can subscribe themselves to the service and the results can be shown in a service dependent way.

8.2 General approaches to run the service thread(s)

In the previous chapter has been said that the service designer design decides how many threads should run the service. Four general methods are explained:

1. One thread for service execution.
2. Create a dynamic buffer of threads where service requests can route to.
3. Create a limited dynamic buffer of threads.
4. Create an event mechanism on top of the services.

For each of these methods a separate sub-section is devoted. Each of these approaches has its own preferred use for different kinds of services.

8.2.1 One thread for service execution

With this approach the service designer does not have to take extra actions, since one thread is automatically created at installation time for each service.

This approach is interesting if the service execution is fast, without any user interaction or waiting cycles. Examples of one thread service execution parts are: conversion of one number into another one, or count the number of times the number is phoned.

The overall execution is faster if the service request are handled concurrent, than when more threads are created and the service runs virtually simultaneous.

8.2.2 Create a dynamic buffer of threads where service requests can route to

This approach creates some threads and let them wait until a service request is routed to it. The approach explained in section 8.2.1 lets the service request wait until the thread is ready. With this approach it is the other way around, the threads have to wait for service requests.

The buffer can increase and decrease according to the needs. It can increase when new threads are created and put into the buffer. It can decrease if threads are idle for a long time and therefore garbage collected. To store released threads in the beginning in the buffer so the next service request immediately ceases that thread again and the ones at the end of the buffer will eventually be garbage.
Parallel processes for telecommunication with Java

collected when no service request occur for them. Figure 8-1 illustrates the mechanism in which threads wait for service requests.

Figure 8-1: Thread buffer mechanism.

A drawback to this approach is that no upper limit to the number of threads exist.

8.2.3 Create a limited dynamic buffer of threads
This approach uses the same buffer as the one described in section 8.2.2, but the thread buffer has a maximum size. The designer must specify a maximum number of threads running the service. If all threads are handling service requests, the service requests are stored in a buffer and wait until a thread is available again. This approach can be seen as using approach 2 up to a certain number of concurrent service requests. If more service requests are issued the requests are stored in the buffer according to approach 1.

8.2.4 Create an event mechanism on top of the services
With this approach the designer creates a kind of scheduler. The running service request can put itself in hold. Gives the control to another service request and if this request wants the processor it generates an event, and waits until the request is granted. Threads are introduced to eliminate this kind of event mechanisms at lower level, but for some services with a lot of user interaction and waiting moments the self regulating of the process can be an improvement to the performance.

8.3 Creation of the televoting service
According to the design steps the televoting service is created. The reason for creating the televoting service is that it generates a lot of load to check the performance. Also updating the results and statistics real time on screen is provided. After a brief explanation of the televoting service, the service creation steps are explained according to the standard service creation design steps, listed in section 8.1.

8.3.1 Explanation of the televoting service
The televoting service is a service where people can phone to a predefined set of numbers. Every number represents something or somebody you can vote for. By dialling one of the numbers a counter connected to that number is incremented. A subscriber can hire a set of numbers, and can look at the results on screen to see what the voters have been voted for.

8.3.2 Service creation according to the standard creation design steps
According to the service creation design steps the service is built up. This section explains every design step in more detail.

8.3.2.1 Look in the ToolBox environment for reusable classes.
Since the televoting service is the first service created in Java the ToolBox environment is empty and all the functionalities are designer made.

8.3.2.2 Implement standard reusable classes in the ToolBox environment.
After examination of the service the only reusable class is a standard frame which can display meaning :result combinations. This frame class is called 'LabelTextFieldFrame' and is described in more detail in
Appendix C. In the televoting service situation the first label represents the meaning of the called number. The second label represents the number of times that number is called.

**8.3.2.3 Determine the number of threads that run the system**

In section 8.2 some general approaches to thread mechanisms that may run the services are explained. The televoting service only increases a counter according to the dialled number. The service execution is fast, and without any delays. The approach that only uses one thread is best suited for the televoting service.

**8.3.2.4 Implement the abstract methods of the service you want to create.**

Section 7.2.2 in the previous chapter describes a set of abstract methods that the service designer is obliged to implement. Implementation details about each of these methods are explained in this section:

1. **Invoke.** In the invoke method, the actual service execution part is implemented. The only thing that must be done here is to increase the counter of the number that has been dialled. Look in the numberlist for the number and increase the counter of this number.

2. **ConvertParameterIntoVariable.** All the CID parameters that are routed from the SSI environment are connected to a variable name that is understandable for this service. The only variable in the televoting service is the B_Number. It represents the KoN2 parameter in the SSI environment. Since this is the only parameter the parameter value can directly be coupled to the variable.

3. **SubscribeToService.** This method creates an instance of the AddNumberFrame. The AddNumberFrame has already been described in the previous chapter. It asks the user to type the number, and the meaning of that number, and whether or not you want to use another number.

4. **Showresults.** This method carries as argument the String representing the subscriber. It subsequently does the following:
   - Create an instance of the LabelTextFieldFrame to show the results on screen. The LabelTextFieldFrame has been described in section 8.3.2.2.
   - Determine the number of numbers the subscriber uses.
   - Invoke the addltem method for all the numbers the subscriber has subscribed to. As arguments to this method the meaning and the counter representing the number of times the number is called, are passed.

5. **PrepareOutputStream.**

   After service execution, the Java side can sent an OK or Not OK indicator, depending on the validation of the vote.

**8.3.2.5 Create the MML script.**

The service designer has to write a script of MML commands to notify the service in the SSI environment, to set the IR to know which parameters are needed, and to provide the service with an unique ID. In Figure 8-2 this MML script is shown. All the used commands are explained in Appendix B.

```
SSLPISLNAME=teleser;
SSLMMODULE=teleser.CTRTYPE=TOJAVA,IR1=272,IR2=2;
SSLMMODULE=IDLE.CTRTYPE=CHSTATE,IR1=0;
SSLCL:START,MODULE2=teleser;
SSLCLI:MODULE1=teleser-0&&-1.MODULE2=IDLE;
SSLPE;
SSACI:SANAME=access,SLNAME=teleser;
SSABE:SANAME=access;
SSAAC:SANAME=access;
SSANP:SANAME1=access;
```

*Figure 8-2: The MML-script of the televoting service.*
The meaning of the part between the procedure `initiate (SSLPI)` and the procedure `end (SSLPE)` is illustrated in Figure 8-3

![Diagram](image.png)

**Figure 8-3: Illustration of the televoting MML script.**

This MML script creates a module called `teleser`, in the `ToJava` controltype. It sets `IR1` to 272, meaning that the service id = 1, and only one input parameter is passed to Java. It sets `IR2` to 2, meaning that parameter `KN2` must be passed to Java for the service execution. The IR values are set according to the protocol specification in section 6.2.2.2.

`CTRTYPE CHSTATE` is used to change the state of the SSI according to the outlet of the `TOJAVA`. [TMOS2-page 46] describes the `CHSTATE CTRTYPE`.

**8.3.2.6 Insert the service in the service Creation and management environment.**

The following steps must be taken when adding the `TeleVotingService` in the service creation and management environment:

- Make sure the class named `TeleVotingService` is in the Service directory and the `LabelText` field in the ToolBox directory.
- Click on the Add Service button in the ServiceManagement frame and type 'TeleVotingService' in the created frame.

**8.4 Conclusions**

This section has listed the different design steps for creating new services in Java, and a service has been created that is built up according to these design steps.

Different mechanisms can be used to determine how many threads run a service. For each service type a different mechanism is preferable.

The service runs according to the specifications, meaning that:

1. It can be added at run time
2. Subscribers can subscribe themselves to the service
3. The results can be shown on screen and the update button, updates the results.
4. The service can also be removed from the environment.

The socket server stores its incoming requests in a buffer. This buffer is fixed at 50 requests. When creating a high number of service requests, some service requests are ignored. This buffer is not accessible or re-sizable by the programmer (fixed part of the environment). A way to avoid these problems is presented here:

- Copy the complete buffer in a larger one at predefined time stamps, read the messages from this buffer instead of reading the exact number of bytes for each service request separately.
9. Conclusions and Recommendations

This report describes the results of a graduation project, which is focused on integrating Java into the telecommunication environment and IN in particular. This final chapter presents the conclusions of this report and provides some recommendations for future work.

9.1 Conclusions

In this section some conclusions will be presented:

- Network interfaces
  In this report different networking interfaces have been discussed. The main reasons why socket interface has been chosen are:

  1. It is standard provided by the AXE-VM.
  2. It is standard provided by Java-VM.
  3. It is a widely used networking interface.

  This socket interface has been chosen as a raw network interface. On top of this a protocol has been defined to enable transparent data communication between both sides.

- Designing of a Service creation environment
  Java communicates directly to the SSI environment (through the AXE-VM). A special CT has been created that takes care of invoking Java. It routes the service request to Java, together with a set of parameters. For each created service the service designer has to design a script that connects to this CT. The script gives the service a unique identifier, and tells the CT what parameters must be sent to Java. A major advantage to this approach is that services can be added or removed in the designed environment at run-time. The reasons that enables this feature is:

  1. The service dependent data is dynamically linked into the SSI environment.
  2. Java’s standard capability to insert new objects at run-time.

- Creating a standard service operation and handling environment at the Java side.
  In Java a service operation and handling environment has been created, providing the following capabilities:

  1. Subscribers can be subscribed in a service dependent way at run time.
  2. It is possible to install a new version of an existing service, at service execution time.
  3. From the moment the service has been removed, new service requests are routed back to the AXE-VM with an error message. The pending service requests on the other hand, are handled as if the service has not been removed.

  Capabilities to add and remove services at the Java side are also implemented in this environment.

9.2 Recommendations

In this section some recommendations for future work are given.

- The current created environments enables Java to execute services. These service are initialised at the AXE-VM side. Regarding the present capability to phone somebody over the Internet, the situation that Java initialises the service initialisation part must be investigated in more detail.

- Create a configuration file. This file can contain among other parameters: the Internet addresses of the local and remote host, and the port number to where the socket connection is initialised. This instead of hard coding it in the programming code.
• A beginning has been made for a general service environment in Java. It can be extended to a complete operation and management environment, where security and more error detection mechanisms are implemented.

• The connection is initialised at start-up time. A direct consequence of this is that when something goes wrong both processes will have to be restarted. Since this is not preferable, a better look at a proper connection initialisation and release will have to be investigated in more detail. In the case a fault has been detected, the release and automatically re-initialisation of the socket connection should be provided.

• Currently the 'OK' or 'NOT OK' parameters can be routed towards the AXE-VM. Extend the environment with the possibility to send changed parameter values back to the AXE-VM. These values can be sent with the same protocol as used to send data to Java. A parser at the AXE-VM side should for this reason be available to translate the message into understandable information.

• Make more use of the introduced priority levels. Some possibilities are listed here:

1. The socket connection has been set up for 4 different levels of priority. To extract this to the service execution part, 4 buffers for each service should be available to store the pending service requests.

2. The possibility for a subscriber to buy a certain priority at the Java side can be integrated into the environment.

3. Giving a priority to the service in the Input register protocol, one can extend the service installation part. Complete services can be handled with a different priority in this case.

• The Implementation part of the socket listener at the AXE-VM side is not provided yet. The socket connection has been set up, but no mechanism is provided to retrieve and handle the incoming data. Some ways to implement this part are described in chapter 5.

• A possible improvement to the way AXE-VM and Java-VM communicate to each other is presented here:

Instead of creating a MML script for each newly created service, send the service ID to Java. Java tells the AXE-VM which variables will be needed for execution. The AXE-VM determines the variable values and sends them to Java. It is certainly a simplification to the environment, at the expense of performance.
10. References

[AXE] AXE-10 Survey. Advance copy of the AXE 10 survey textbook. EN/LZT 101 1513 PA7

[Q.1200] ITU-T Recommendation Q.1200
Q-series intelligent network recommendation structure
Helsinki: ITU, March 1993

[Thörner] Thörner, J., Intelligent Networks,


[C++Ref] Ellis Margaret A. and Bjarne Stroustrup, The Annotated C++ Reference Manual,


[CDK] Martien Huysmans, CDK The Controltype Development Kid 0.4, Example of how to
design a controltype in C++, Ericsson nr. 1551-CDK-97:0001 Uen

Interprocess Communication, Ericsson nr. 1/1551-APZ_AXE_VM Uen, sept 1997


[1003] Doc no: 82-CNT 249 1003 UEN

[1004] Doc no: 82-CNT 249 1004 UEN

[1005] Doc no: 82-CNT 249 1005 UEN

[1006] Doc no: 82-CNT 249 1006 UEN


73
Appendix A: Class numberlist

This public class extends the List class available in the JDK. List is defined in [List]

A overview of the methods numberlist provides is given here:

- public void AddNumber(String Number, String Meaning, String SubscriberName)
- public String GetMeaningofNumber(String number)
- public void RemoveSubscriber(String Subscriber)
- public int AmountofNumbersForSubscriber(String Subscriber)
- public int getSubscriberIndex(String Subscriber)
- public int getSubscriberIndex(String Subscriber, int i)
- public void AddNumberOfOccurrances(String number)
- public void AddNumberOfOccurrances(int index)
- public int getNumberOfOccurrances(int index)
- public String getMeaning(int index)
- public int getMeaningIndex(String Meaning)
- public String getMeaning(String Number)
- public int getNumberIndex(String number)
- public String getSubscriber(int index)
- public String getSubscriber(int index)
- public String addSpaces(String S, int
Appendix B: MML commands for the televoting service

- **SSLPI.** This command stands for Service logic Procedure Initialisation. It opens a procedure to create a service. The command syntax is: `SSLPI:SLNAME=<slname>;`. It opens a procedure to create a service logic named slname.

- **SSLMI.** This commands stands for Service Logic Module definition Initiate. It is used for defining the modules within a service logic. The syntax of this command is:

  ```
  SSLMI:MODULE=<moduleaname>,CTRTYPE=<ctrtypeaname>[,NOUTLETS=<nro!outlets>],
  ,REGNR=<regnr>,[IR1=ir1[,IR2=ir2[,IR3=ir3[,JR4=ir4[,JR5=ir5[,JR6=ir6[,JR7=ir7[,JR8=ir8]]]])[OR1=or1[OR2=or2[OR3=or3[,OR4=or4[,OR5=or5[,OR6=or6[,OR7=or7[,OR8=or8]]]]]]]]]]
  ```

  The NOUTLETS is only necessary when the controltype does not define the number of outlets. IRx (ORx) are used for two reasons:

  1. It gives the position in the intermediate input (output) register from (to) where the data is read (stored).
  2. It specifies the controltype usage.

- **SSLCI.** This command stands for Service Logic Connection Initiate. It connects one or more outlets from a module to an inlet of another one. A precondition to this command is that both modules have already been defined. The command syntax is: `SSLCI:[START, MODULE1=<module1name>-a&&-b,],MODULE2=<module2name>;`. START points to the start of the SL. The -a&&b- with a and b of type 0..255, connects the outlets a until b of <module1name> to the inlet of module <module2name>.

- **SSLPE.** This command stands for Service Logic Procedure End. The syntax is: `SSLPE;`. It ends the procedure. Note that the SSLPE command is registered in the AXE-VM as a dangerous command. This means that the AXE-VM wants a confirmation that it is the right command. Entering the ';' character gives this confirmation.

The administrator commands:

- **SSACI.** The command stands for Service Administrator Connection Initiate. It connects a service administrator to service logic. This command can only be executed for the NOP (Not OPerating) area. The syntax of the command is: `SSACI:SANAME=SANAME,SLNAME=SLNAME;`

- **SSABE.** This command stands for Service Administrator Blocking End. It de-blocks the service administrator. The syntax is: `SSABE:SANAME=SANAME;`

- **SSAAC.** This command stands for Service Administrator Activity Change. It switches the service logic between OP(operating) and NOP. The syntax of the command is: `SSAAC:SANAME=SANAME;`
Appendix C: The LabelTextFieldFrame class

This frame is used to show a frame of meaning:results combinations on screen.

Two methods are available in this frame:

1. **addItem.**
   Method addItem carries two input parameters both of type String the first represents the meaning label and the second the result label. The method inserts this label:label combination into the already defined rows.

2. **Updatebutton_actionPerformed.**
   This method reacts on clicking on the 'Update' button available on the frame. For all the items in the meaning label field it updates the result labelfield, by getting the counter that counts the calls for the number, represented in the meaning label.