Helsinki University of Technology
Department of Computer Science and Engineering
Telecommunications Software and Multimedia Laboratory

Juhana Räsänen

Broadband service architecture

A Master's Thesis submitted to the Department of Computer Science and Engineering of Helsinki University of Technology in partial fulfilment of the requirements for the degree of Master of Science in Engineering

Supervisor Olli Martikainen
Professor of Network Architectures and Protocol Engineering

Helsinki University of Technology	**Abstract of the Master's Thesis**

Author	Juhana Räsänen
Subject	Broadband service architecture
Pages	72
Date	June 1st 1999
Department	Department of Computer Science and Engineering
Chair	Tik-109 Telecommunications Software
Supervisor	Professor Olli Martikainen

This thesis is about service architectures for service provision in the future broadband networks.The work has been carried out as a part of the Calypso research project at the Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory.

First the evolution of service architectures for different kinds of Intelligent Network paradigms is considered, after which the aims and scope of the Calypso project are described. An abstract, three-layer control model for networks is introduced and an architecture design and requirements for the highest layer (Service Control Layer) is developed. In the practical part of the thesis a prototype implementation of a Java-based software platform for dynamically deployed services is presented with example services and the suitability of Java for broadband service development is considered.

Keywords	Service architecture, Intelligent Networks, Broadband networks, Java

Teknillinen korkeakoulu **Diplomityön tiivistelmä**

Tekijä	Juhana Räsänen
Aihe	Laajakaistaverkon palveluarkkitehtuuri
Sivumäärä	72
Päivämäärä	1. kesäkuuta 1999
Osasto	Tietotekniikan osasto
Professuuri	Tik-109 Tietoliikenneohjelmistot
Valvoja	Professori Olli Martikainen

Tässä työssä käsitellään laajakaistaisessa monipalveluverkossa tarvittavia palveluarkkitehtuureja. Työ on suoritettu osana Calypso-tutkimusprojektia Teknillisen korkeakoulun Tietoliikenneohjelmistojen ja multimedian laboratoriossa.

Aluksi käsitellään erityyppisten älyverkkomallien mukaisten palveluarkkitehtuurien kehitystä, jonka jälkeen esitellään Calypso-projektin tavoitteet ja laajuus. Työssä kuvataan tietoverkoille kolmikerroksinen abstrakti kontrollimalli, jonka ylimmälle tasolle (palveluiden kontrollikerros) esitetään arkkitehtuurimalli vaatimuksineen. Työn käytännöllisessä osassa kuvataan prototyyppitoteutus Java-pohjaisesta dynaamisesti verkkoon ladattaville palveluille soveltuvasta ohjelmistoalustasta sekä analysoidaan Java-kielen ja -ympäristön sopivuutta laajakaistapalveluiden toteuttamiseen.

Avainsanat Palveluarkkitehtuurit, älyverkot, laajakaistaverkot, Java

Preface

Writing this thesis has been a long process and the result became a summary of my work at the Telecommunications Software and Multimedia Laboratory during years 1996-1998. Research work is a collaborative process and the results most often have contributions from several people. This thesis is not an exception and I wish to express my sincere thanks to all people who helped me in my work.

First of all I thank my supervisor, professor Olli Martikainen for the intriguing ideas and continuous support for Calypso project. His exceptional insight of the network architectures and telecommunications business has been the main source of inspiration for this work.

Second, I wish to thank all my colleagues at the University for the numerous discussions on the matters of this thesis - and on the many other matters as well. Especially Petteri Koponen and Jouni Karvo have been the major contributors when the understanding of broadband service architectures was formed in the research group. Also Pekka Nikader has been very kind in helping me and other people learn the principles of the TeSSA security architecture. Many other people have participated, too, and because the list would grow too long to fit it here, I give you a big, collective Thank You.

The financial support from TEKES and the participating companies of the Calypso project is gratefully acknowledged, as well as support from First Hop Ltd. in publishing this thesis.

Finally, I thank my dearest Silja who made me concentrate on this thesis and yet helped me balance my work and other important things in life.

Espoo
June 1st, 1999

Juhana Räsänen

Contents

Abbreviations

ADSL	Asymmetric Digital Subscriber Line
API	Application Programming Interface
B-IN	Broadband Intelligent Network
B-ISDN	Broadband Integrated Services Digital Network
CA	Certification Authority
CBR	Constant Bit Rate
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DFP	Distributed Functional Plane
DPE	Distributed Processing Environment
DSL	Digital Subscriber Line
EJB	Enterprise Java Beans
FCL	Fabric Control Layer
FSR	Frame Synchronized Ring
FTP	File Transfer Protocol
GPRS	Generic Packet Radio Service
GSM	Global System for Mobile communications
GSMP	Generic Switch Management Protocol
HTTP	HyperText Transfer Protocol
HUT	Helsinki University of Technology
IETF	Internet Engineering Task Force
IFIP	International Federation for Information Processing
IN	Intelligent Network
INAP	Intelligent Network Application Part
IP	Internet Protocol
IPSEC	Internet Protocol SECure
ISP	Internet Service Provider
JAR	Java ARchive
JDK	Java Development Kit
JVM	Java Virtual Machine
ISDN	Integrated Services Digital Network
LAN	Local Area Network
LANE	LAN Emulation

MPEG	Motion Pictures Expert Group
MPLS	Multi-Protocol Label Switching
MPOA	Multi-Protocol Over ATM
NCL	Network Control Layer
OMG	Object Management Group
OS	Operating System
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RFC	Request For Comments
RMI	Remote Method Invocation
RSVP	resource ReSerVation Protocol
SCE	Service Creation Environment
SCF	Service Control Function
SCL	Service Control Layer
SCP	Service Control Point
SEE	Service Execution Environment
SPKI	Simple Public Key Infrastructure
SS#7	Signalling System No. 7
SSF	Service Switching Function
SVC	Signalled Virtual Channel
SwC	Switch Controller
TCP/IP	Transmission Control Protocol / Internet Protocol
TeSSA	Telecommunications Software Security Architecture
TINA	Telecommunications Information Network Architecture
TINA-C	TINA Consortium
TML	Telecommunications software and Multimedia Laboratory
UML	Universal Modelling Language
URL	Universal Resource Locator
VC	Virtual Channel
VCI	Virtual Channel Identifier
VoD	Video on Demand
VoIP	Voice over IP
VPI	Virtual Path Identifier
WWW	World Wide Web
xDSL	(some) Digital Subscriber Line

1 Introduction

In the course of the late 1990's the telecommunication and datacommunication worlds have experienced significant expansion, development and convergence. The digital communications is becoming an ever more important part of the everyday life of people all over the world. Two trends can be seen: the usage of digital communications in an increasing number of areas of life and the convergence of all digital communications and media into a single ubiquitous network that provides a wide variety of communications and media services. The vision is that this network and all its services could be reached via any access method by all users: corporate, residential and mobile users all over the world using whatever access network that happens to be available.

The vision is not yet reality. On the contrary, it seems that the whole field of communications is more complicated and distracted than ever, with new technologies, trials, services and service providers emerging everywhere. All stakeholders in the business are trying to be the first ones to see the future, to be ready before the world changes and to change the world themselves. We can see, for example, equipment manufacturers, telecom and datacom operators, software houses, financial institutions, traditional media houses such as newspapers, TV, cinema and music producers as well as the new media houses operating in the Internet, all aiming at the same direction: to get their fair share of the vast communications market in the new millennium. However, this apparent confusion does not mean that there would not be any progress towards a more unified communications world. Some enabling technologies, discussed in Section 1.1 below, can be expected to become the cornerstones of the future communications infrastructure.

1.1 Enabling technologies

For the unified communications world to become reality, a number of technological developments are necessary. Some of these technology requirements are related to the performance issues, such as the available communications bandwidth "per capita" and the capability of the terminal devices to handle the broadband content which has been traditionally handled in the analog media instead of the digital networks. The performance issues are especially critical in the mobile networks where the bandwidth

resource of the air interface is scarce and the computing power of hand-held, battery-powered devices is limited. However, the performance issues seem to be currently the easiest to solve, as the gigabit- and even terabit-capable networks are being developed and the performance of the computing devices is still growing according to the Moore's law. Even the mobile world has kept up with the progress and the 3rd generation mobile networks to be commercially introduced in the next decade will be capable of delivering broadband traffic to terminals that resemble more small hand-held computers than traditional mobile phones.

1.1.1 Internet

A harder problem than the performance is to provide the same services for the users in different types of networks: fixed and wireless as well as residential and corporate networks. The trend that makes this requirement important can be seen in the Internet today. Originated as a research project to connect computers into a global network, the Internet has grown during its history of three decades to a network that joins *people* all over the world together. The communications services that are valuable for the users of the Internet are the reason for its success, not the technological fact that the computers are connected. In fact, the Internet has become so successful that it has started to attract services from the other networks and media. Long-distance voice calls, news services, music and purchase of goods seem to be currently the strongly developing new areas of the Internet. In the future access to all digital content and global delivery of interactive digital television-like broadcasts have potential to become the next wave of services offered in the Internet.

It can be argued that the Internet has already become one of the enabling technologies for the future communications model envisaged in the beginning of this Chapter. The core technology of the Internet - the TCP/IP protocol family - has become *the* protocol suite supported by all computers, all operating systems and all networks regardless of their manufacturer or intended purpose. The last to adopt TCP/IP have been the mobile networks, but even in them the support is being built, for example in the GPRS (Generic Packet Radio Service) for the GSM networks. The common protocol suite is essential for the service development, the *lingua franca* spoken by all devices. The possibility to transfer data between any two terminals or other nodes in the network

potentially makes the same service accessible from any kind of terminal in any access network[1]. Naturally the terminal devices set technical limitations of which services can be accessed, but the principle of all services being available in all networks is very important.

1.1.2 Mobile code

The nature of the Internet and the nature of the traditional networks, such as the telephone network or the analog TV network, is very different. The traditional networks are specialised to transfer one kind of data and the terminal devices are specialised to handle that data in the most efficient way. The Internet, on the other hand, is by nature a multi-service network in which there is no strictly predefined set of data types, user interfaces or applications. Every service defines its own protocols, data formats and mechanisms and eventually creates a number of applications that are able to handle that particular service. This has not been a significant problem in the Internet so far, because the number of services has been limited and they have also been relatively simple in operation. Thus it has been possible to define a limited set of application specific standards for, e.g., e-mail, netnews, FTP or WWW. Although WWW is very versatile and enables the development of a tremendous range of services, it is hardly suitable for all services. The voice-call services are a good example of this: there are many standard proposals on how to carry voice over the Internet and likewise there are many non-interoperable Internet phone applications. When the number of services increases and their internal logic becomes more complex, it cannot be any more assumed that every terminal device and intermediate network node would support the protocols and data types of each service by default.

An interesting solution for this problem lies in code mobility, meaning that the application able to handle a particular service can be transferred as a part of the service data. Applications have been transferred over the Internet since its early days, but they have been delivered either in source code form or in binary form precompiled for a particular hardware and operating system combination. If a user wishes to access a

1. It should be noted, however, that TCP/IP is not ideal for all kinds of services. Especially the real-time services that have strict requirements on delay and delay variation are not well handled in the current Internet. Solutions for this are being developed, and also the Calypso project has its own approach to the problem.

service that requires a certain application, she has to download the application, possibly compile it and install it to her computer. Although cumbersome, this has worked somewhat and as the support in the operating systems for the most common Internet services has become better, this has not become a major problem.

However, the emergence of Java has made it possible to change the style of service deployment in the Internet. Java is a platform independent programming language whose executable programs can be easily and securely transferred in binary form to the terminals. So far Java has mostly been used to create small applications, so-called *applets*, embedded into web pages where they are automatically downloaded and executed by web browsers. This has made it possible to encapsulate service specific protocols and data types into the applets that are able to access the actual service without the user having to manually download and install an application.

1.2 Goals of this thesis

This thesis carries further the idea of using Java to implement communications services. The aim is to extend the concept of downloadable applets to cover also distributed services that have dynamically downloaded service specific components in the intermediate network nodes. This kind of services would be able to manage service specific state information on the network side, such as the QoS state or billing context of the connections that form a service session. The concrete environment in which this kind of service architecture is considered, is the provision of communications and digital media services for the residential users in a fixed broadband access network. Because the service logic processing and service session management can be distributed close to the edges of the network, the services become more scalable than in the two-tier server-client model, which is important when the services have potentially a large number of subscribers. The architecture also enables a flexible business model, in which independent service providers are able to purchase resources from the network operator and use them to deliver the service to the subscribers, thus making new kinds of value chains possible. This thesis is also the documentation of ideas and vision formed during the time the author was working in the Calypso project, beginning from the end of 1996 and continuing through 1998.

1.3 Organisation of the thesis

The organisation of this thesis is following: After this introductionary Chapter the need for broadband service architectures and their evolution are briefly considered in Chapter 2. Chapter 3 introduces the Calypso project, its goals and scope as well as the business model that has been setting the business-related background and assumptions for the project. Chapter 4 introduces a layered control model that was created to partition the functionality in the network to make the actual service architecture more clearly visible. The service architecture itself is introduced at conceptual level in Chapter 5. Chapter 6 describes a prototype implementation of the most important software component of the architecture, the service platform for dynamically deployed distributed services. Finally, Chapter 7 draws some final conclusions of the developed architecture and prototype and suggests further work. In addition, Appendix A gives some technical background to the prototype platform implementation and Appendix B demonstrates service development in the Calypso architecture with examples.

2 Evolution of service architectures

2.1 Need for service architectures

From the service and service architecture point of view two significant trends in the communications world can be seen. One is the attempt of the traditional network operators to maintain their *status quo* in networking by trying to get a hold of the business in the future networks. The other is the Internet becoming a platform for services that have traditionally been handled by the operators, such as voice or real-time video delivery. These two developments meet at a point that is the subject of interest in this thesis, the architecture needed to provide services in a broadband network. When these two trends are studied more carefully, the following observations can be made:

1. The telecom networks (PSTN, ISDN, ATM) are circuit switched, which guarantees the QoS for real-time services such as voice or video. When a connection is made, a physical or virtual circuit is reserved from the network and the connection may use the reserved bandwidth freely during the session. In IP networks on the other hand, the traffic is packet switched which means that guarantees on the QoS are hard to make, because the network does not monitor individual connections and their state. However, some QoS guarantees would be required, if real-time services are to be provided in the Internet.
2. In the Internet there are no common billing or accounting mechanisms. When commercial services are introduced, they require that the service provider and the subscriber make a contract and arrange billing in some service specific way. In the telecom networks, however, it has been possible to introduce services that are billed by the network operator. Only the operator and the service provider need to make a contract to arrange that the service provider gets paid by the operator for the services used by the subscribers.
3. Neither the Internet nor the telecom networks are programmable in the sense that the network could execute arbitrary service specific code. The telecom networks do have the Intelligent Network service paradigm in which service logic can be constructed and executed in the network, but this has only limited possibilities for service development. The Internet has no common service model whatsoever.

When the observations are considered, it may seem that the telecom networks have an edge over the Internet: the Internet cannot guarantee QoS and it does not have any common billing, accounting or service models. However, the power of the Internet is its tremendous flexibility. Many simple but important services do not really need any kind of QoS guarantees, billing or service logic, but require only that data can be delivered within reasonable time from a host to another. These services include electronic mail, WWW, FTP, remote login, etc., and even real-time data can be supported if the network has enough bandwidth. In circuit-switched world the traditional Internet services become heavy due to connection set-up overhead.

An ideal would be that the network could support both connection oriented data with QoS guarantees and packet-switched data as in the Internet today. An additional benefit would be if the services could have easy access to billing and accounting systems provided by the operators and if the services would be able to manage the service sessions on the network side. The purpose of this thesis is to claim that such a network can be developed, if there is an open interface to the network resources and a service architecture that supports development of distributed services.

Service architectures that aim at convergence of the Internet and telecom networks have been introduced. Heinonen describes four different approaches to bring intelligence into the network (Heinonen 1998): broadband IN, TINA, so-called Smart Networks and active networks. These are discussed briefly in the Sections below.

2.2 Broadband IN

Intelligent Network is the service paradigm of the traditional telecom networks. The core idea of IN is to bring service specific intelligence to the call setup process by introducing an external server that is separated from the basic switching infrastructure. The server (SCP, Service Control Point) hosts a number of service logic programs that are activated by a call setup requiring special functionality. The activation is triggered by e.g. the called party number and the service logic is able to control the network to complete the call in a non-standard way, e.g. forwarding the call to another number or directing the billing to the called party. Because the terminal devices (phones) are dumb, the service logic is executed completely in the SCP, excluding possible dialogues with the user with voice instructions and touch-tone dialling.

The broadband version of IN extends the narrowband IN with the concept of session, an association of calls and connections that together form a service session. The service logic can also be distributed to other network devices and intelligent terminals, but the interface to the service logic remains basically the same, based on the signalling protocols. Broadband IN could be used to support e.g. billing, statistics gathering, connection performance monitoring and address translation between the E.164 and IP addresses.

The problems of B-IN lie in its delayed standardisation and relatively rigid service model that does not support Internet-style networking easily. At the same time it seems that the importance of ATM is decreasing as the performance of the IP-based networks increases. ATM has been the cornerstone of the IN-based broadband ISDN network architecture and if ATM fades away, there is little use for B-IN. However, the concept of a SCP as a network element capable of executing service specific functionality is likely to stay alive to support IN-like functions in other architectures.

2.3 TINA

TINA (Telecommunications Information Network Architecture) is an ambitious project by a number of operators and manufacturers joined together into TINA Consortium (TINA-C) in the end of 1992. The goal of TINA is to specify an all-encompassing architecture that would define the structure of the future broadband multimedia networks. TINA design is based on three principles: object-oriented analysis of the environment, distributed processing and conceptual, well-defined separation of architectural components. By these principles three sub-architectures have been defined: the Distributed Processing Environment (DPE), TINA Service Architecture and TINA Network Resource Architecture. DPE is based on CORBA (Common Object Request Broker Architecture) with some adaptation to the telecommunications environment. The service architecture defines the building blocks of services and their interfaces to each other and the other parts of TINA. The network resource architecture defines a generic interface to the network resources to encapsulate the network technology specific details into a single interface that all TINA applications would be able to use. In addition to the architectures, TINA also specifies a role-based business model framework.

TINA has been a massive project to address the problems of the broadband networks in a thorough top-down manner. However, its thoroughness has also been its drawback and the overall architecture has grown so complex that it is unlikely to ever materialise as a whole. However, many principles of TINA are quite applicable, such as the distributed processing and object-oriented analysis. Also the business model has useful concepts and role definitions. Thus it is likely that many ideas of TINA outlive the architecture and will be found in the future network architectures.

2.4 Smart Network

The Smart Network is a concept being developed in the IFIP TC6 (International Federation for Information Processing, Technical Committee 6) Working Group 6.7. The aim is to create a platform consisting of a generic switch fabric infrastructure and an execution environment into which the service logic can be deployed as software modules that can be easily updated, changed or removed. The platform would allow creation of configurable networks in which the same hardware could be easily configured for different purposes with the control software modules. The software modules could be, for example, different kind of signalling or routing modules that would configure the platform to act e.g. as a standard ATM switch or an IP router. In this concept even the individual services can be dynamically downloadable modules, if service deployment functionality is built into the platform.

The smart network is a radically new idea of how broadband networks could be designed. It must be noted, however, that the idea does not exclude the existing network architectures. By installing proper software modules a smart network platform could be easily configured to act as a standard ATM or PSTN switch, for example. The additional value of the smart network concept comes from the possibility to extend or update the functionality, thus extending the lifetime of the hardware. Also the development of interworking devices is easier when the control functions of two different network architectures can be installed on the same platform. For example, a smart network platform could be configured to act as an ATM / Internet / PSTN gateway by installing ATM, PSTN and VoIP (Voice over IP) signalling functions (Raatikainen et al. 1999). The possibilities of implementing individual services as downloadable software modules is studied in more detail in this thesis.

2.5 Active networks

The architectures presented in this Chapter are in an increasing order of flexibility. In active network concept the flexibility is brought to the maximum by letting the data control itself how it is treated in the network. The driving force for active networks is to get rid of the slow process of defining and standardising new protocols and networking mechanisms. If the network nodes were generic purpose computing platforms with a well-defined interface to the resources, and if the data could contain instructions to be executed on those platforms, a new protocol could be simply established by writing a new program that is encapsulated into the data.

This kind of concept is very radical. There are a number of problems yet remaining unsolved, such as security, performance issues and the definition of the active platform that provides the computing environment and interface for the active packets. However, in some respects the active network concept is similar to the smart network, but the functionality is deployed even more dynamically. Currently active networking is purely a research issue, but the research could produce useful results and better understanding of the networking and service architectures in general, so it should be followed by the developers of other architectures.

3 Calypso project

The evolution of the service architectures is clearly visible in the research carried out in the broadband services research group of the Telecommunications Software and Multimedia Laboratory (TML) at Helsinki University of Technology. The work started in 1996 in TOVE project when it seemed that B-IN would be a promising alternative to implement broadband services. However, ideas of a new kind of service architecture started to take form in the end of 1996, when the first experiences of developing services in broadband environment had been gained in the TOVE project in which the author worked during 1996. They can be summarised as the development of concrete services on top of IN-based B-ISDN (Broadband Integrated Services Digital Network) being relatively inflexible. The problems were the lack of applications, programming tools and libraries for the ATM / IN environment as well as the delay in getting the final versions of the standards out from the standardisation process. There was also no clear picture of how the services were to be developed in practise. INAP (Intelligent Network Application Part) and its broadband version B-INAP were available, but it was not well understood how they would relate to the development that could be seen to be gaining momentum on the Internet.

At the same time it seemed that supporting TCP/IP (in other words, the Internet and all its services) on top of ATM infrastructure in the standard way[1] was not very efficient nor scalable. Companies like Ipsilon (later acquired by Nokia) started to emerge and offer controversial solutions in which the signalling overhead of ATM and B-ISDN had been thrown away completely. The purpose of this was to get the best of the both worlds by replacing the heavy ATM signalling stacks with IP routing: the simplicity of the connectionless IP networks on top of the QoS capabilities of the ATM data transport layer. Other solutions for this were, e.g., MPLS (MultiProtocol Label Switching) from IETF and Tag Switching by Cisco. These technologies are analysed in more detail in (Heinonen 1998).

1. The options were LANE (LAN Emulation) or MPOA (Multi-Protocol Over ATM), both defined by the ATM Forum. The core idea of both is to create virtual IP networks using ATM signalled virtual channels and mapping the IP addresses to ATM addresses. When a connection to a host is opened, the LANE or MPOA layer resolves the ATM address from the IP address of the host and if necessary, opens a SVC to that host or to a gateway that is able to forward the IP packets to the host. The purpose is to create “cut-through” virtual channels so that data would not go through the network layer (i.e. get routed) at each ATM switch.

These experiences caused the research in the broadband services research group in TML to split into two parallel tracks (Räsänen et al. 1997). The TOVE project continued on the "evolutionary" track by adopting the ideas emerged in OMG to integrate CORBA with the SS#7 and IN to create a more flexible environment that still would support the legacy systems (OMG 1998). The "revolutionary" track created the Calypso[1] project, whose organisation, core ideas and goals are described in this Chapter. Also the business model of the networks for which Calypso was intended is discussed.

3.1 Project organisation

Calypso project was organised as a partly TEKES-funded (Finnish Government) project with a number of partners from the telecommunications industry, such as Sonera (former Telecom Finland, the biggest operator of Finland), Nokia Research Centre and the Technical Research Centre of Finland (VTT)[2]. The project was led by professor Olli Martikainen of the TML Laboratory of the Helsinki University of Technology and the author started to work as the project manager. The work was officially started in May 1997 and was carried out at HUT. The phases of the project covered in this thesis continue till late 1998, during which time the research staff included also Petteri Koponen, Juha Pääjärvi, Olli-Pekka Auvinen, Kim Lahti, Jukka Aro and Michael Zhidovinov. The project continues in 1999 with mainly new staff.

3.2 Research objectives

The main goal was to develop a service architecture suitable for prototyping new kinds of services that were expected to be provided in the future broadband networks. The focus was set to the fixed access networks for residential areas, because the consumer segment was expected to become the most important market for the broadband networks. This kind of network had also good research opportunities, because the core and corporate access networks were already better covered in the research and also commercially well established. One reason for selecting the access network was also the assumption that the intelligence in the networks needed to be brought closer to the edges of the network, because no centralised model could take the load resulting from

1. There is no special meaning associated with the name in this context, nor is it an acronym.
2. VTT supplied the hardware platform for the project, the FSR (Frame Synchronized Ring) switch which already been used in the TOVE project.

the huge customer base of the residential users. This included also an idea of the network being a *platform* on top of which different kinds of services could be easily implemented using standard programming tools and environments. In other words, the functionality of the network was to be opened as API for the service applications. This idea is also the core concept of the smart networks discussed in Section 2.4.

Other objectives for the project were to gain experience on using Java to implement distributed service functionality in the network nodes and to address the security implications of such a service architecture. An important part was also the prototyping of the actual services that were expected to be provided in the future networks. Another, relatively independent part of the project has been the research and development of a service creation tool that could be used to model service functionality using UML (Universal Modelling Language) and to create the service implementations semi-automatically from the high-level business logic model utilising a repository of ready-made components. This part of the project has been carried out in co-operation with the Peoples' Friendship University in Moscow, Russia.

3.3 Initial assumptions

Having the focus on the residential consumer access networks leads to a number of assumptions about the capabilities of the networks and terminal devices as well as the behaviour of the end users. From the technology point of view there were three assumptions made initially:

1. The bandwidth of the customer access line will be limited to a few megabits per second in most cases. This assumption is based on the expectation that the existing telephone copper pairs are going to be the dominant physical medium over which the network access is provided. The worldwide installed base of the telephone lines is so huge[1] that it cannot be competed by any other medium in penetration, possibly excluding the terrestial TV broadcast network, which is not very suitable for bidirectional network access. The solutions to utilise the telephone cables, the so-called xDSL (Digital Subscriber Line) technologies, provide typically a few megabits per second over reasonable distances from the nearest telephone exchange to the customer (ADSL Forum 1998).

1. 800 million according to the ADSL Forum, see http://www.adsl.com/adsl_status.html.

2. The terminal devices will be some kind of set-top-boxes or personal computers. The device may redistribute the service content to the other devices at home (such as the TV set, phone or PC) via a home network, but the central intelligence is in the terminal device that is connected directly to the network. The device has also processing capability and a general-purpose platform to which executable code of the services can be downloaded.
3. The underlying network technology supports CBR (Constant Bit Rate) streams capable of delivering real-time video. This kind of traffic has strict QoS requirements for the bandwidth as well as delay and delay variation, which in practise requires some connection oriented network architecture. On the other hand, the Internet with its applications and user interfaces (in particular WWW) is very important, so the network should also support TCP/IP level connectivity. As was pointed out earlier in this Chapter, the marriage of connection oriented networks and the Internet is never an easy one, so a feasible technical solution must be found.

In addition to these technology assumptions the user expectations and behaviour were assumed to change more slowly than the technology and be mainly based on the current services. The purpose of the broadband network is to be a multi-service network in which existing services (such as telephony, television and Internet access) could be provided in addition to the new interactive multimedia services. To be commercially successful, the broadband network should maintain or exceed the usage experience of the legacy services, because the customers most likely are not satisfied if the current services are provided with poorer quality without any additional value. An example of such a situation are the video-on-demand trials that have been common broadband pilot services. The problem with these trials has been that VoD cannot compete with the existing service, namely the rental video stores. The selection in the trials has been limited to a few dozen video titles for technical reasons, whereas the rental stores have a selection of thousands of titles. It is clear that the VoD trials implemented as isolated services cannot meet the customer expectations and thus have not been very successful in terms of customer satisfaction.

However, the potential of the broadband access lies in its ability to combine the existing services and new possibilities. If, for example, the VoD service were implemented so that it would be tightly integrated with the Internet, e.g. to get

background information of the films, the service would clearly gain additional value from being provided in the broadband network. Likewise, a digital TV channel could also carry other information than the video stream and the uplink connection (possibly via the Internet) could be used for interactive programs. Thus, it is important that even for the existing services the possibilities of the broadband networking are utilised.

The assumptions have some technical implications to the implementation of the actual services. The assumption on limited bandwidth, for example, implies that the user access line can carry only a few simultaneous good quality video streams. In digital TV service this would mean that only a small fraction of the available channels can be carried over the customer access line at the same time. Consequently the procedure of changing the channel is different: when the user selects a channel, it must change on the network side instead of the terminal device. Assuming that the user behaviour does not change much, a commercial break in a program would cause a sudden peak of channel change requests to the network. This kind of peaks can hardly be managed centrally if the usage experience of the analog TV, in which the channel changes immediately, is to be maintained. Thus, the network side service logic must be distributed effectively and close to the edges of the network.

3.4 Business model

From the beginning of the project the aim has been to create a service architecture that would suit commercial environment in which there are players engaged in different kinds of business relationships. It is worthwhile to study the roles of the players to understand what kind of requirements the business model might set for the technical solution. The scope of this thesis does not allow for a thorough analysis on the business model for broadband networks, but a simple, role-based model can be sketched.

3.4.1 Changing business environment

In commercial environment the reason for an individual or organisation to operate can be seen to gain value by exchanging it to something that is valuable for another player. The roles of the players can be determined by thinking what they have to offer and what they wish to gain. First of all, the fundamental reason for a residential broadband access network to exist is to provide the households and private customers networking

services for which they are willing to pay the service providers. Service provision requires two assets: the actual service content or functionality and the infrastructure needed to bring the service to the customer. Usually these assets are not owned by the same organisation. The infrastructure is typically owned and managed by the traditional teleoperators who have built most of the existing datacommunication and telephone networks, including the customer access lines. The services and their content on the other hand, are most commonly produced by independent service providers.

Traditionally the service providers have either worked in close cooperation with the operators or the operators themselves have assumed the role of service providers. This can be seen especially in the mobile networks such as GSM, in which many of the value-added services are provided by the operator. This is likely to change in the future as the multimedia services become more and more complex and their creation process resembles more movie production than traditional IN-style service creation. This requires skills that are hardly part of the core business of the operators, which forces them to redefine their role in the future communications business.

Two lines of development can be seen. In centralised model the operators seek allies among the media houses to form communication consortiums that would manage everything from the service production to the infrastructure within a single organisation. Another model can be seen evolving in the Internet, in which a service can be provided basically by anyone who has a web server and is able to develop or purchase the service specific functionality and content. In this kind of diverse model the role of the operator is to be the connectivity provider, whereas the services are a composition of the efforts of software houses, content producers and system integrators and other organisations possibly participating in the service production.

3.4.2 Business roles in Calypso

The Calypso service architecture is designed for the latter kind of business model, in which the valuable asset of the operator is seen to be the infrastructure and not the actual services. However, the architecture is designed so that the operator has better possibilities to offer the resources for the other players, unlike the Internet model in which the operator just connects the customer to the service providers. In addition to the operator and customer Calypso identifies several other players, such as service

providers, content providers, financial institutions, specialised software houses that are capable of creating the services, etc. Some of the roles have been adopted from the TINA business model, but in general the Calypso roles are less strict and there are no predefined reference points (interfaces) between the roles as in TINA. The most important roles are summarised below.

1. **Customer** is in many respects in the most important player of the business model. The reason for the broadband network to exist is to provide the customer a number of services and charge her for the service usage. The customer is the source of revenue that in the end keeps the business financially feasible, so her satisfaction is important.
2. **Network operator** has the infrastructure that is needed to deliver the services. The operator needs to co-operate with the service providers to get profit from the infrastructure, because the infrastructure itself is of little value for the customers.
3. **Service provider** manages a particular service that is valuable for the customers. Service provider is dependent on the operator to provide the connectivity to the customers, but otherwise wishes to remain in control of the service content and its user base.
4. **Service implementor** is a software house that is specialised on service creation in the Calypso network. Even if the service architecture would be simpler to program than traditional architectures, the implementation is likely to require skills that at least the small service providers are not willing to manage. The simplicity of the architecture, however, gives a chance for the smaller software houses to specialise in service creation.
5. **Content provider** is closely related to the service provider. The role of the service provider can be seen as the middle ground between the content provider and service implementor. It can be imagined that in many cases the service provider's role is combined with either the content provider or the service implementor, depending on service provider's orientation to technology or content.
6. **Bank** may offer financial services in the network, such as an electronic monetary system that is respected by all parties to facilitate easy charging and billing. Note, however, that billing services could be provided also by the operator who has a relationship with the customers.
7. **Certification authority** provides a public key infrastructure that is the basis of the security model and the management of trust relationships

between the players. The CA could also perform quality control on the service implementation so that the operator and service provider can be assured that a service is properly implemented. The quality control could also be a separate role as well.

It must be noted that many of the roles can be mixed, for example the operator is likely to offer its own services, thus becoming a service provider.

3.4.3 Implications of the business model

Two important technical requirements stem from the described business model: an open API to the network resources and a solid security model that supports the business model. The assumption that the services are created and provided by several organisations and companies, some of which may be relatively small, requires that the network resources are freely usable once a business contract to provide services in the operator's network has been made. The reason for this is that the number of the service providers is likely to become high and on the other hand the service life cycles become shorter. It is in the operators interests to keep the amount of support needed for service creation and deployment as small as possible - an ideal would be that the operator sells a right to use a certain amount of network resources for a particular purpose, after which the service provider is able to create and deploy the service independently.

The assumption that part of the service code needs to be executed in the network nodes owned by the operator requires a flexible security model. The operator cannot fully trust the code implemented by the third party service providers or software houses. For this reason the operator has to either inspect all code that is going to be executed in the network or be assured that the code simply cannot do more than what the service provider had been authorised to. Inspection is in contradiction with the little involvement by the operator and support for short service life cycles, so the run-time access control mechanism should be capable of preventing the services to exceed their granted privileges. The security model is analysed in more detail in Chapter 6, but it is pointed out that the security model design is derived from the business model requirements.

4 Layered control model

One of the goals set for the Calypso project was to define an architectural framework for service creation and deployment in broadband networking environment. The services in the context of the framework mean the software components that are needed to connect the end user (or her terminal, to be more precise) to a particular server offering digital content, such as the server of a TV broadcasting company offering digital TV channels in the broadband network. The framework can be realised as a software *platform* that offers both the programming tools and interfaces as well as a run-time service execution environment for the service providers. However, before such a service architecture is defined, an overall control model is useful in setting the bounds for the operational environment.

Control model definition was one of the initial tasks of Calypso. Because its purpose was to support the development of the service architecture rather than be a subject for research itself, a simplistic approach was selected. The model extends the idea of separating switching and control functions from each other presented in the TOVE project (Puro et al. 1996) and other similar projects (Lazar et al. 1996, Rooney 1997). In Calypso another conceptual separation was made to separate the service-specific functions from the operation of the underlying network. This resulted in a three-layer control model in which the functionality is divided into fabric control, network control and service control layers (Koponen et al. 1997). This Chapter describes the three-layer control model in detail.

4.1 Three-layer control model overview

Partitioning functionality into layers is a tradition in the computer science and communications. The control model needed for the Calypso is not an exception, but it must be noted that the layering presented here is more conceptual than functional, and in a practical implementation as described in Chapters 5 and 6 the functions on the different layers use each other freely rather than in a strictly layered manner. The model is kept simple compared to the B-ISDN distributed functional plane (DFP), for example, so that it would only give basic guidelines instead of a rigid predefined structure not suitable for a research project. The levels of control defined are control of a single device (switching fabric), the network of such devices and individual

services provided in such a network. Figure 1 gives an overview of the layers in the three-layer model and communication between the elements. The fabric control occurs locally at one switching device, network control is end-to end within the network and service control occurs between components of a single service distributed into the devices of the network.

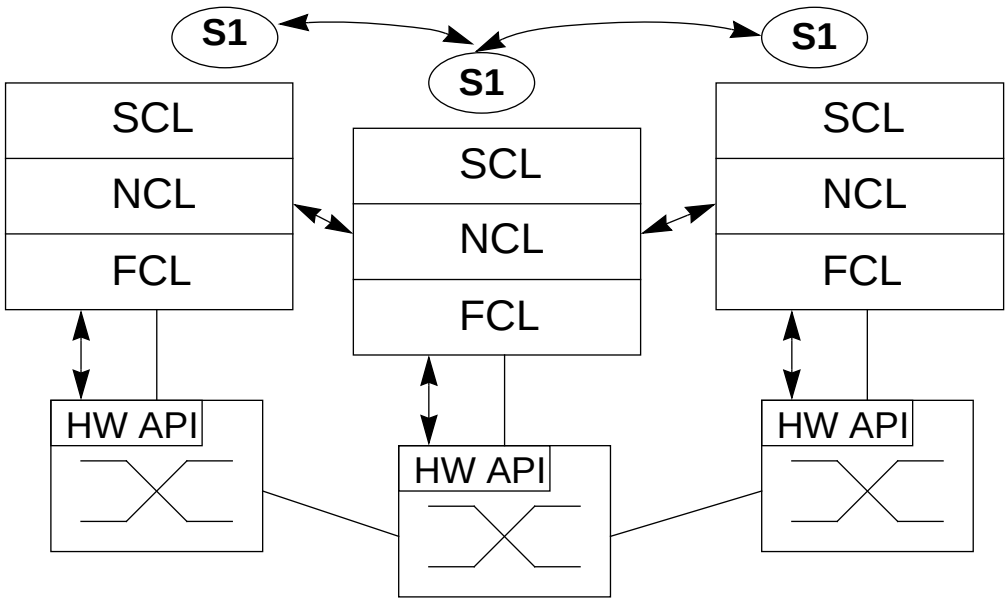

Figure 1 Three-layer control model

4.2 Fabric control layer

Fabric control layer (FCL) is the collection of functions that manage one networking device through which the service data connections are routed. Examples of such devices are ATM, Ethernet and telephone switches, IP routers and firewalls. It should be noted that the devices of interest in this context are active network devices whose operation can be controlled with software running in a possibly separate control processor (workstation). Passive network elements such as Ethernet hubs or repeaters are not thought of as FCL devices.

The purpose of the FCL is to provide a clear separation between the specialised hardware functionality (the switching fabric) and the software functions (such as signalling) that can be run on a general purpose workstation. This separation enables simpler hardware components, because control processors are not integrated as

embedded units into the devices. This in turn makes the hardware more cost-efficient and the software easier to develop and upgrade. Because the software functions are physically separated into an external workstation, the device itself does not need any mechanical parts such as hard disks to operate. This makes the devices more resistant to environmental conditions, which means that they can be more easily placed for example in cellars or telephone poles, while the control processors can be placed into the machine rooms of the network operator. This is important in building the broadband access networks for residential areas, because air-conditioned machine rooms cannot be built everywhere; on the other hand the switching structure may have to be distributed closer to the edges of the network, because increased bandwidth cannot be handled in concentrators as big as the current telephone network exchanges.

One additional benefit gained from the physical separation is the possibility to form logical devices by grouping physical devices under a single control processor or to even let several processors control one high-load device. The processor can also be easily replaced with a more powerful workstation if performance upgrade becomes necessary due to increased traffic, for instance. The idea of a virtual exchange was presented in TOVE project as a possibility to group low-cost ATM switching fabrics into a construction that externally behaves as a a single logical ATM switch (Puro et al. 1996).

The basic requirement for the separation of the hardware switching functions and the software control functions is that the controlled device has a well-defined application programming interface (API) to the device functionality. Typically this includes configuration and connection management as well as statistics collection and event logging facilities. The most important one in the three-layer model is the connection management, which enables opening, closing and monitoring individual connections in the software level, which is a fundamental concept in the control model. Examples of the connection management APIs are General Switch Management Protocol (GSMP) (Newman et al. 1998) and the FSR API for controlling FSR switches (Haatanen and Tuliluoto 1998). In practise the connection management API or its higher level software wrapper (such as an object-oriented class library) can be seen as the representation of the FCL in the software level.

4.3 Network control layer

The FCL defines the lowest level interface to the underlying network and provides an API to the network devices. Even if we assume that the network device and its control processor are separate units, the FCL defines only a local interface between the device and the controller rather than a general network level structure. The network control layer (NCL) covers functionality needed for network operations, such as end-to-end connection management, host addressing and transport layer data security.

The most important function of the NCL is end-to-end connection management. The basic assumption in Calypso has been that the multimedia services need per-session QoS guarantees (such as minimum bandwidth or bounded delay), which in practise requires a connection oriented network, such as RSVP (resource ReSerVation Protocol) capable IP network or ATM. Another issue calling for per-connection management is the accounting and billing of the services. Flat-rate fees make billing difficult in a heterogeneous environment in which several service providers offer their services through the single network and everybody must get their fair share based on how the customers used the services. This requires per-connection statistics for implementing time-based or volume-based billing systems.

Distinction between connectivity and connection management is made in the three-layer control model. Connection state management for QoS or accounting is necessary only for the actual data connections the carry the service content. This makes it possible to establish a logical control network that carries only control data, such as service requests and signalling traffic to set up data connections. Provided that the necessary traffic priorities are set up so that control data has always precedence over user data, the logical control network can be connectionless in nature, which has a profound practical consequence: TCP/IP can be easily used as a generic transport protocol in the control network. This facilitates the usage of all numerous tools, utilities and protocols readily available in the TCP/IP world, which is significant from the software development point of view. Also IP addressing can be used for host identification in the network. TCP/IP is not an absolute requirement, but as pointed out in Chapter 1, it is the de facto standard that is supported in any case.

If an IP based control network is established, it can be used for end-to-end connection management as well, regardless of the lower layer network technology. If the nodes are able to communicate over TCP/IP, it is possible to set up a signalling system that reserves connection resources hop-by-hop by forwarding a connection setup message using standard IP routing. The requirement is that the intermediate nodes are able to intercept setup messages and reserve resources for the data connection before the request is forwarded in the control IP network. This is the mechanism used in the Internet RSVP (Braden et al. 1997), but the same kind of mechanism can be used to set up connections in an ATM-based network as well.

One additional feature resulting from the separation of connectivity and connection management is that TCP/IP is not necessary for carrying the user data, because IP connectivity has been utilised already in the connection setup phase. If the underlying network supports virtual circuits (such as ATM), IP can be omitted in the data connection. This means that the same signalling mechanism can be used in many kinds of networks and if proper gateway functions are implemented in the interconnection points, the data streams may be connected between different types of networks, for example RSVP and ATM.

4.4 Service control layer

The NCL defines the notion of a connection, an end-to-end association between the communication parties engaging in a service session. A connection may have two or more end points, it is associated with a set of QoS parameters and in a commercial network most likely has also a billing context. However, for several reasons a higher level of abstraction is needed to describe the environment in which the actual services exist. First, a service session may consist of one or more individual connections, which may or may not be between same end points. While extending the connection concept to a more general session concept is possible, taking all possible cases into account in a generic session abstraction might be difficult. Second, the service life cycle is short whereas the underlying network architecture tends to remain more stable. For this reason the services should be decoupled from the network architecture, so that the capabilities of services can be easily extended when necessary. Third, the services typically contain executable code, *service logic*, for which an execution environment

is needed. For these reasons the service control layer (SCL) is defined to provide the services a run-time environment in which they operate.

The essential component of the SCL is the service execution environment (SEE) that provides service management and execution in the run-time environment of a network node. The service programming model is a generic object-oriented model with additional APIs provided by the SEE. The services can be seen as programs that are dynamically installed into and executed in the SEE of a network node. A service may also have components that are executed in the end users' terminals or the service providers' servers, and the components are able to communicate with each other over the logical, IP-based control network of the NCL.

The generic transport protocol (TCP/IP) provided by the NCL allows usage of any proprietary or service-specific communication method in the control network, which makes the service implementation very flexible. This is an important difference to the traditional IN service model. Although the IN model makes basically the same separation by identifying Service Switching Function (SSF) and Service Control Function (SCF), the interface through which a service receives requests is fixed in the INAP protocol definition. This kind of model is suitable for the telephone network, in which the terminals (phones) are simple and not able to execute service code. However, the broadband network terminals are most likely going to be full-featured computers or set-top-boxes, which enables the implementation of the services as distributed programs. For this reason the interfaces internal to a service should not be restricted by the architecture, only the interfaces provided *by* the network *for* the services, such as connection management API, should be predefined.

One aspect of the service execution is also the application security, which must be taken care of in the SCL. Because services are dynamically loaded code written by third party service providers and a single network node may execute code from several different sources at the same time, the run-time environment of the SEE should ensure that the service components do only what they are authorised to. It should be possible to grant individual services rights to perform different operations on the network as well as to limit resources (such as bandwidth) that are available for the services. The access rights should be granted prior to the service implementation so that the service

provider would be able to program the service code according to the granted rights and resources, but the rights should be checked at run-time. In addition, the access right enforcement should be as transparent as possible for the service programmers.

4.5 Three-layer control model in Calypso

The three-layer control model does not mandate any particular technology, it merely identifies groups of related functions that can be found in a connection-oriented network in which multimedia services are offered for the end users. In a real network, however, technology choices must be made. The aim of the Calypso project was to experiment with different kinds of broadband services to gain more understanding of the service requirements - thus some kind of operational environment was necessary for implementing the service prototypes. Because the services and their execution environment was one of the central subjects for research, the SCL was left to be developed during the project, but the FCL and NCL were fixed in the beginning (with the exception of the end-to-end connection management mechanisms of the NCL, because no suitable existing signalling protocol was available).

The Calypso fabric control was chosen to be based on the API of the FSR ATM switch. The research group had previous experience of the FSR and at that time ATM seemed to be a viable option for bringing broadband access to the homes. The FSR API is able to control a single ATM switch at the virtual connection level and the API also supports both unicast and multicast connections, which made it suitable for the services that were planned to be implemented in the project.

The FSR API does not contain any network functions, however, so another solution for the NCL implementation were needed. Although ATM signalling for the FSR would have been available as a result of the TOVE project, it was not taken into use mainly because there would not have been an easy way to carry arbitrary service control data between the service components distributed into the network nodes and end terminals. Implementing services as distributed applications requires in practise TCP/IP and some middleware technology, such as CORBA, on top of it. Technology to establish TCP/IP networks over ATM using standard ATM signalling exists (ATM Forum 1997), but in Calypso this would have been redundant functionality. Because ATM signalling was not going to be used for the connection management of the actual

services, using it only for setting up the control IP network would have resulted in unnecessary complexity of the software. It was decided that the control IP network would be established as Classical IP over ATM connections (Laubach and Halpern 1998) over default permanent virtual channels - much in the same way as the ATM signalling network is established on default signalling virtual channels. In Calypso the default control IP virtual channel was chosen to be VPI=0, VCI=15, whereas the ATM signalling VC is VPI=0, VCI=5. This left the possibility to use standard ATM signalling and Calypso-style services in parallel in the same ATM-based network.

The logical control IP network of Calypso and connection setup using it are described in Figure 2. The figure has one client (the grey triangle), an access switch and an intermediate switch. All nodes have an IP address and a default VC (0,15) connected to the neighbour node, used to send and receive control IP traffic. The connection manager services (CM) of the switches are able to work as an IP router with the help of a routing service (R). When a client wants to make a connection to an IP address, it forwards a request on the control IP channel. The switch intercepts the request and reserves a data connection (thick solid line) before forwarding the request packet according to the current IP routing tables. Once the connection setup is completed, a response is sent back to the client that is now able to use the data connection for the service data.

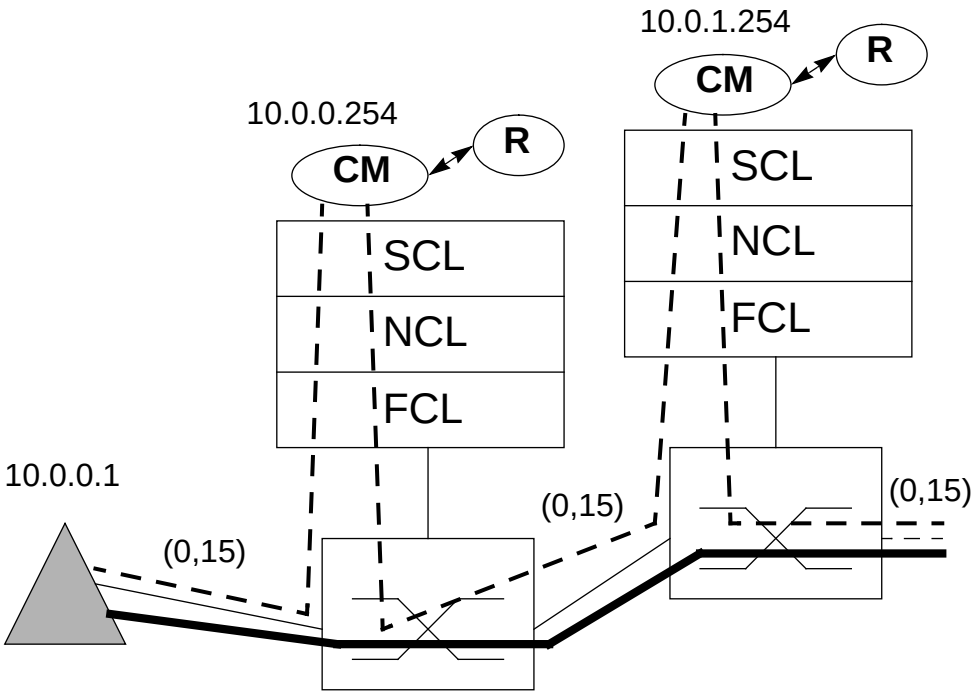

Figure 2 Logical control IP network over ATM

Finally, the SCL was left to be designed during the project and its architecture design is described in Chapter 5. One design choice was made at an early stage, however: Java was selected to be the implementation language of the Calypso service platform. Some features of Java, such as built-in support for TCP/IP and higher-level protocols (HTTP and FTP among others), integrated distribution technology (RMI, Remote Method Invocation), platform independent code and code mobility became the cornerstones of the Calypso service platform prototype described in Chapter 6.

5 Calypso service architecture

The Calypso service architecture defines the software components, security model and programming model for the services and the software needed to execute them in the Calypso networks. The service architecture is the reference on basis of which the concrete implementations of the service environment are built. The purpose of the architecture is to structure the software that is needed to create, deploy and execute end-user services as well as set the rules and conventions for service development. In relation to the three-layer control model introduced in the previous Chapter the service architecture described in this Chapter can be seen as the SCL specific for Calypso networks. However, as will be seen later, in Calypso some of the functions of the FCL and NCL have also their counterparts in the service architecture, so the boundary between those abstract control layers is not strict in Calypso.

This Chapter first identifies the important components of the service architecture and then proceeds by setting a number of requirements for the architecture. Chapter 6 describes the prototype implementation of the Calypso service platform developed as a part of this work.

5.1 Architectural components

The components of the architecture are logical units of software that are executed in the nodes participating in the service sessions: end-user terminals, intermediate network nodes and servers. One logical software component may be distributed between the different nodes, i.e. there can be active parts of one service in several nodes and those components are able to communicate over the network.

5.1.1 Services

The services are the fundamental component of any service architecture. Commonly services are understood to be the end-user services that implement some functionality valuable for the end-user. Examples are providing multimedia content, providing connectivity to other users in the same as well as external networks, providing electronic commerce services, etc. In other words, an end-user service is the collection of software needed to perform a specific task in the network on the behalf of the user.

However, in Calypso the definition of a service is broadened (Magedanz and Popescu-Zeletin) to include a part of the functional components in the underlying system software. These components are not directly visible to the end user (although their effect is most visible as successful operation of the network), but they implement lower-level functions to operate and administrate the network. Examples of such functions are connection management, routing, accounting and billing services, network administration, etc.

Treating the lower-level functions as services is especially beneficial from the programming point of view. When the system level functions are implemented using the same programming model as the end-user services, the underlying software for managing the services, so-called *service platform*, can be implemented as a simple general-purpose component. It is also easy reuse existing functionality to compose services (both end-user and system-level) of other, lower-level services.

5.1.2 Service platform

An analogue to the service platform and services would be the operating system kernel that executes system and application programs. The service platform (kernel) is the generic component and the services (programs) are components designed to perform a specific task. The purpose of the platform is to manage multiple, concurrent services running within a single network node, but it does not itself provide any application specific functionality, such as the APIs of the Calypso FCL or NCL. Thus the platform itself becomes a reusable component that could be used in a variety of other applications. It can be argued that in Calypso the service platform is of the microkernel breed of operating systems, because most of the low-level system functionality is implemented also as services rather than integral parts of the platform.

The platform also manages the shared general-purpose system resources, such as the file system, memory and CPU. Although the interfaces to access these resources can be provided as system-level services (for example, a scheduling service to execute tasks), the management of the resources belongs fundamentally to the platform. This is because the eventually resources are accessed through system calls of the host operating system or by some other mechanism, which is most likely not exposed to the services. If the resources are shown as system services for the end-user services, the

system service acts as a *proxy* that is able to perform access control or serialize access to a resource that can be used by only one service at a time.

5.1.3 Security model

The security model is tightly coupled with the platform and access to the system resources. A wider perspective to security covers two things: the run-time security of the services executing on the platform and the cryptographic security of the data transmitted over the network. From the service architecture point of view the former is more important, whereas the connection-level security is to a large extent internal to the NCL. The API for connection management may include security features on the data transport layer, for example IPSEC functionality, but it is not understood to be a part of the security model in this discussion.

The run-time application security consists of three components: *permissions*, *policy* and *access controller*. The permissions represent individual access rights that can be given to a service, e.g. a permission to read a file or to open a network connection. Policy is the static or dynamic configuration that defines how the permissions are assigned for each service. The access controller is a piece of software that enforces the policy by performing a run-time check every time a service tries to access a resource requiring a specific permission. The access controller, being an active software component, must be a part of the service platform. The permissions and policy, however, are not active components and could be represented with files, for example. This issue is studied in more detail in Section 5.2.1 below and in Chapter 6, which describes the prototype platform.

5.1.4 Programming model

The programming model, although not a software component, is a central part of the service architecture. It consists of the set of rules, conventions and constraints resulting from the design choices of the architecture. The rules must be followed by the service developer when designing and implementing a service. These include, e.g., the choice of programming language, possible limitations to the features and standard libraries of the language that can be used, interfaces (in the sense of object-oriented programming) that the service must implement and other properties that the service must have to be

accepted and executed by the service platform. It may also contain recommended practises for successful service implementation. Many aspects of the programming model can be built into a service creation tool, in which case the rules and interfaces are enforced automatically without the programmer having to bother about them.

5.1.5 Service creation environment

The service creation environment (SCE) is the programming environment for service developers. It supports the service programming model in a high level on which the developers do not need to concern themselves with the low-level details of the overall architecture or the service platform. The services could be represented, for example, as graphical components in the user interface of the tool so that they could be visualised and manipulated in an intuitive manner. The service logic could be represented in a high-level modelling notation, such as UML (Unified Modelling Language), instead of programming language constructs. In this case the SCE could assemble the service by interpreting the model of the service logic to create a composition of predefined library components of the tool (Martikainen et al. 1997). The SCE could also support service testing by providing a run-time environment in which a service can be executed in a simulated network. Services could also be installed into the live network using the same tool, if the network supports dynamic deployment of services.

Although any deeper discussion of the SCE is beyond the scope of this work, it is recognised that the SCE is the concrete service architecture representation visible for the service developers. For this reason the SCE is, like the service platforms of the live network nodes, the component that integrates the other parts of the architecture.

5.2 Requirements

The requirements for the service architecture described in this Section have emerged gradually during the Calypso project. These requirements form the basis for the design of the service platform prototype described in Chapter 6. The requirements are divided into three categories: security, functional and non-functional requirements. Although the security requirements contain both functional and non-functional issues, they have been given a separate category to emphasise their importance and to highlight the fact that they originate to a great deal from the business model described in Chapter 3.

5.2.1 Security requirements

The most important security feature of the Calypso service architecture is the need to execute services from several organisations or vendors concurrently on the service platforms. The organisations are independent of each other and the network operator, who is the owner of the service platforms. Direct consequence of this is that the different parts of software running on a single service platform have different amounts of trust associated with them. The amount of trust is generally based on the business relations and contracts between the organisations. For example, if a service provider has made a contract with the network operator to deliver digital TV channels in the network, the contract does not warrant the provider to relay voice calls in the same network. The amount and nature of trust placed on a service provider results in a different sets of access privileges given to the service code executed in the network node. In other words, the trust associated with the service providers that have services running in a network node translates into the security policy of the node.

The trust is expressed as permissions that are granted to the services. They must be fine-grained enough to support the business requirements, so that service providers can buy only the necessary resources from the network. On the other hand, the network operator should also be able to trust that services operate on the minimum required privileges principle and are able to access only the resources the providers have made contracts for. A coarse-grained model in which the services are placed on different security levels based on the origin or the signature of the service is not enough, because the services would receive unnecessary privileges in the assignment to a trusted level. Each service should be assigned a unique security context that includes only the privileges the service needs during its execution.

Flexibility is the most important requirement for the policy configuration. Because the services originate from several sources and can be deployed into the live network at any time, a static policy configuration (in a file, for example) at each network node is not feasible. The services should carry their credentials with them and the service platform should adjust its policy configuration at the installation time of a new service. This requires that the platform is able to securely verify the credentials of the service by cryptographic means. In practise, the credentials can be carried in *delegation*

certificates bundled with the service code (Aura et al. 1998). A certificate is a cryptographically signed statement that identifies its subject, in this case the service code, and its credentials, or the permissions that the service should have at run-time. When a network node downloads a service and verifies its credentials, it may enforce additional local policy constraints before adding the permissions of the service into the policy configuration. For further reading about security and trust management in agent-based systems (Nikander 1999) is recommended

The security model should allow a service to let other services access it, if the accessing service has been granted proper permission by the owner of the accessed service. This requires that in addition to the service platform protecting the system resources, also the services should be able to perform access control to their own resources. The accessing service should be able to reveal part of its credentials to the accessed service, which should be able to verify the validity of the credentials. The services must also be able to rely on the platform not to reveal the code or even presence of services that do not explicitly make themselves available.

The security model should support time-constrained permissions. This requires that the run-time policy configuration is dynamic, i.e. an expired permission can be removed from the security context of the service at the correct time. Some permissions may also have other type of constraints, e.g. a service could be allowed a 10 Mbit/s quota of network bandwidth, which it is not able to exceed, or that the service is able to use the resources only up to a certain credit limit defined in the certificate.

From the service programmer's point of view the security model should be as transparent as possible, so that the developers are not constrained by the run-time security checks. The developer should be able to write the service code without knowledge of the access control mechanisms and assuming that the service provider is able to acquire the privileges for the service independent of the development process.

The security model should be extensible to the access control of the end users, so that the services could utilise the same security primitives and functionality in the authentication, authorisation and accounting of the service access and usage.

5.2.2 Functional requirements

As with the security requirements, the nature of the functional requirements stems from the underlying business model. It can be expected that the service life cycles become shorter and changes to service content and functionality are frequent, as can be seen in the Internet today. For this reason the service creation and deployment process must not be too bureaucratic or complicated. It should reflect the business model in which the service provider purchases resources from the network and after that is able to use them freely for bringing the service to the customers. The resources include the computing power of the network nodes in which the services are partly executed, and the access to that resource should be as direct as possible.

Direct access to the computing resource can be achieved if two criteria are met by the service architecture: First, the programming model of the services must be similar to the traditional software development models. When the services can be developed with a commonly available programming language without expensive and complicated service creation tools and methodology, the investment needed for the small and medium-sized providers and software houses to enter the business is lower. Second, the network must support dynamic loading and installation of service code into the network nodes. If service providers are able to upload code into the network themselves and put it available for the customers immediately, it will be possible to provide and advertise services that change often. It is possible to enter the market with a minimal functionality service that can be easily extended and upgraded if the user base grows and demands more of the service. Also the ordinary error fixes become easier, thus resulting in better overall customer satisfaction.

In addition to the dynamically loaded services the platform should also support persistent and updatable services. Persistent services are, once downloaded and initialised, stored locally by the platform and they survive restarts of the node. Persistence can be implemented either by saving the service code physically into the node or by saving only reference to the code of the persistent services, in which case the platform downloads the persistent services automatically from the network every time the platform is restarted. Because the network nodes and software running in them should have very little down-time, the platform should support upgrades without

having to shut down the platform. This requires that the code of a service should be possible to be upgraded to a new version smoothly without having to restart the process that executes the code of the services.

The service providers should be able to monitor the performance and state of their own service components in real-time over the network. This requires that the service providers are able to connect with their own network management applications directly to the nodes of the network operator. Naturally only data and management functions relevant for the service provider's own services should be accessible.

In case of mobile terminals, migration of services from node to node should be possible. When a mobile user roams in the physical world, the service components in the network should follow the user by moving along on the network side, when the user's terminal initiates a handover. Mobility is not considered in detail within the scope of this work, but it is noted that so-called agent technologies are closely related with this issue.

The network nodes should have a set of core services and other resources available for the service providers. These core services should cover all common functionality that is needed by most applications, so that the service providers would not have to implement those parts themselves. These core services could include, for example, user profile database into which user-specific data of the new services could be added, an HTTP server which the services could use as a method to deliver information and user interfaces to the customers, file system services for temporary storage of data, interfaces to accounting and charging servers if the network operator wishes to provide them as a services for the service providers, etc. The service providers should be able to rely on these services being available in any platform in which the services are deployed, so the set of core services should be at least a operator-wide standard.

5.2.3 Non-functional requirements

Hardware platform independence is an important requirement. Service components are developed by independent service providers and uploaded dynamically into the network. In this case it cannot be expected that the network nodes and all the service providers have environments compatible at hardware or OS level. On the other hand,

because platform independent code is not likely to be as effective as native code optimised for the particular hardware of the network node, it should be possible to have some native parts in the services, too. An example of functionality needing native code performance is real-time processing of live stream data, such as video compression or transcoding, or data encryption. The native code parts should not, however, compromise the security of the platform. For example, in Java the permissions cannot be verified for the native code parts, so the native code libraries must be transferred in a way that the network operator can be assured that the native code is secure.

For the operator's point of view the reliability of the network nodes is essential. When media and communications services are provided for the home and business users, the reliability requirements in terms of service availability must be comparable to those of the current fixed or mobile telephone networks and terrestial TV broadcast network. The operator cannot be held responsible for the reliability or quality of individual services implemented by third parties, but it is in operator's interests to protect the core platform from any programming faults and run-time errors in the third party service code, not to mention deliberate attacks intended to damage the network. The offending services must be, if necessary, shut down and removed cleanly.

Reliability can be increased also by duplication of the hardware. If the operator has duplicated service platforms in which the processes and services are mirrored in real-time to redundant hardware that is designed to take over if the primary hardware fails, it should not affect the service programming model. The developers should not be aware of the duplication and it should not unnecessarily limit the freedom of the service implementation. However, duplication is outside the scope of the Calypso architecture.

5.3 Implementation issues

The architectural components and requirements described in this Chapter set the basic guidelines for the Calypso service architecture implementation. The implementation is not specified in terms of APIs, interfaces, data types, etc., because at this point of the research a standard implementation is not a primary goal. Instead, the architecture is kept in a high level of abstraction to facilitate development of different kinds of

prototypes to gain more experience. However, some implementation-level issues are discussed below to cover the gap between architecture and implementation.

The requirements for the dynamic downloading and platform independence of service code call for a Java-based solution, as discussed earlier in Chapter 1, where Java was named as one of the enabling technologies. Java is the de facto hardware independent programming language and execution environment in which code mobility is inherently supported. Although not strictly required by the architecture, it is believed in Calypso that in practise Java is the only solution that meets the requirements and is widespread enough to be easily taken into use. Although there are other interesting environments that have similar possibilities, such as ERLANG (Armstrong et al. 1996), Java has been selected to be the Calypso implementation language as far as this work is concerned. The discussion in this Section and the next Chapter is from the Java point of view.

As an advanced object-oriented programming language Java is a tool well suitable for implementing complex, distributed and concurrent systems. It has a wide standard API to support the developer, and implementations of the Java Development Kit (JDK) are available for most hardware and OS platforms in common use. Many of its features, such as built-in support for TCP/IP and higher-level protocols (HTTP and FTP for example), thread-based concurrency model, simple memory management with automatic garbage collection, distribution mechanisms (RMI and CORBA), etc., are readily usable in the Calypso implementation.

However, some features needed in Calypso are not found in Java or their implementation is not suitable, which must be taken into account in the implementation. Most notably, because standard Java Virtual Machine (JVM) and the core JDK have a single-process shared memory model and do not have a process-like construct to run several “programs” within a single virtual machine, the implementation must provide its own execution context for the services. This context is the boundary within which a single service executes and it also contains the service’s access privileges to the system resources and other services. Adding this kind of execution model makes the implementation of the service platform a non-trivial task.

Another shortcoming of Java is its security model. Although very advanced compared to other languages, it is still not flexible enough for the Calypso purposes. JDK version 1.1 has a simple sandbox model in which the downloaded code can be given either full access rights to all resources or no rights at all. The decision is based on the code source (expressed as the URL from where the code was downloaded) and signer of the code, which is clearly too coarse in the light of Calypso architecture requirements. The newest version of Java (Java 2 platform, formerly known as Java 1.2), has a fine-grained security model, in which individual permissions can be assigned to the code loaded from different sources and signed by different signers. However, this is only half of the solution needed for Calypso. The static policy configuration of Java 2 based on local configuration files is not usable in Calypso as discussed earlier in this Chapter. There must be a way to determine the permissions belonging to a downloaded piece of code in a more dynamic way. This has been the research topic of the TeSSA project in the Helsinki University of Technology and Calypso relies on its results in the Java security issues (Partanen 1998).

The support of native code libraries is built into Java, but it must be noted that the access rights of the native code cannot be enforced in any way. The native code is able to read and alter any memory location of the JVM process, for example, and there is no practical mechanism to prevent that. For this reason, if native code is needed by a service, it must be delivered to the platform in some other way than bundled with the dynamically downloaded service. In practise this requires that the network operator must review and compile every native code library and install it into the platform independent of the service provider. This kind of procedure does not meet the flexibility requirements of the service deployment, so the operator should always try to provide any necessary native code functionality as standard core services rather than force the service providers implement it themselves.

The next Chapter describes a prototype implementation of the Calypso service platform developed as a part of this work. The prototype does not have a complete set of features described in this Chapter, but serves as a proof-of-concept implementation of the most critical features required from a service platform.

6 Prototype service platform

The most important component of the Calypso service architecture outlined in Chapter 5 is the service platform. Its function is to manage and execute third-party service components that are essentially programs implemented by the service providers. These programs are installed dynamically into the Calypso network by copying the service code to all network nodes where the service is to be provided for the users and letting the platform create an execution context for the service. The platform also needs to manage the security of the system by performing run-time access control on the services that use the resources of the network. These two aspects, the dynamic service downloading and installation into the network nodes and flexible security model required by that were the critical parts studied in the first prototype.

The prototype implementation was carried out in several phases. In the initial phase during summer and autumn 1997 two end-user services were developed without a real platform in place. Experience from these services was used to decide what kind of core services would be needed and how the end-user services would access them. In the second phase these core services were developed along with the design of the platform during the first half of 1998. The first platform prototype was implemented by the author in late 1998. This Chapter describes the design of this prototype, based on the Calypso architecture outlined in Chapter 5.

6.1 Prototype environment

Before the implementation of the prototype could begin, an environment in which the prototype was supposed to be used had to be set. The original idea of the Calypso project as described in Chapter 3 was to implement a service architecture for delivering multimedia services to residential users in a fixed broadband access network. The prototype environment was modelled with this purpose in mind. The network was assumed to be a connection-oriented network, in which data would be carried over ATM virtual circuits to the end users. The prototype network was a simple one, consisting of a single ATM switch, its controller workstation and some client and server machines. The end user terminals were modelled with ordinary PCs, as there were no real set-top-box platforms available suitable for the project. The operating system of all machines in the network was Linux.

Two end-user services were developed into this setup: digital TV delivery over ATM point-to-multipoint virtual channels and an ISP service to provide Internet connectivity for the end users. Both services had a client component in the terminals and a service component in the network node, the ISP service also had a server component in the router that was connected to the Internet. The server component of the TV delivery service was a simpler "data-pump" that only sent the MPEG encoded video streams to the network as ATM point-to-multipoint virtual channels. In both services the client would connect to the network component to request the service, in practise either a TV channel or a connection to the ISP's router. Thus both services needed to access the ATM level connection management interface of the underlying ATM switch, which was clearly identified as one of the core services and was implemented as a Java wrapper to encapsulate the native C-based library for the switch control. The first versions of these services were developed without a real platform as a "hard-wired" server program running in the controller workstation.

In the second phase new core services were developed. One of them was a general-purpose HTTP server with servlet support, which could be used by the other services for adding browser-based user interfaces for the service management. This was demonstrated by implementing such an interface to the ATM switch service so that the switch resources could be managed and monitored using a web browser. Additional core services were the network level connection management, routing and signalling services that could be used to create end-to-end connections instead of local connections within one ATM switch. The existing end-user services were modified to use these network level connection services.

The third phase was the implementation of the actual platform. The goal was to facilitate the development of all core and end-user services using a single programming model as described in Chapter 5. Other goals were also the ability to start and stop services in the network node at run-time (instead of the hard-wired functionality of the 1st and 2nd phase) and to design the fundamentals of the security architecture so that the results from TeSSA project could be added in a later phase to implement true cryptographic authentication and authorisation mechanisms.

Figure 3 describes the set-up of the demonstration prototype. The machines are connected via a logical IP network as described in Section 4.5. The ovals represent the components of the different services. The services that are drawn inside the control layers represent the core services of the platform (HTTP server, connection manager, routing and FSR control) whereas TV and Inet represent the end-user services. All services are implemented using the same service programming model as described in Chapter 5.

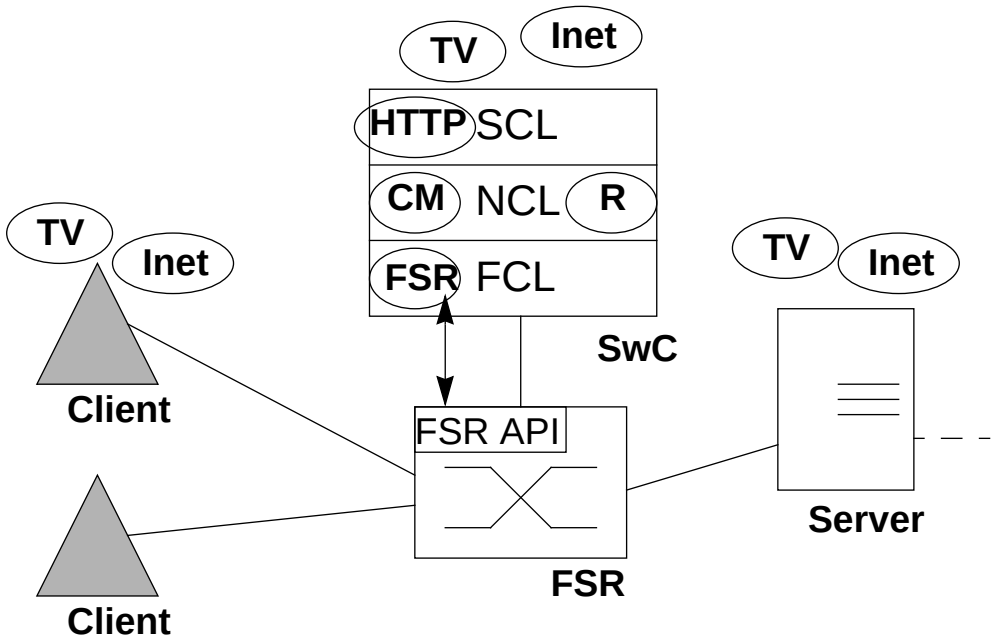

Figure 3 Calypso demonstration environment

6.2 Service execution model

The execution of the services in Calypso resembles the agent technologies that have emerged in the last few years and have also been applied in telecommunications (Albayrak 1998). The distinction between the Calypso service components and agents in general is subtle. The agents can be characterised by them being more independent software components that may move from a node to another on their own initiative, whereas the Calypso service components are intended to stay more permanently in the node in which they are first installed. However, especially when considering the services offered for the mobile users, the Calypso components might also need to move from node to node in the network. In some texts the Calypso service components are also called “service agents”, but otherwise the current platform cannot be considered as a “pure” agent platform because the agent mobility has not been a central

requirement. Instead, mobility could be built as a core service on top of the Calypso platform.

The execution of a service in the network node begins when a Calypso platform downloads the service code, creates an instance of the downloaded service and starts it within a new execution context. For this to be possible, all services must implement a common Java interface[1] defined by the platform API. Once the service has been started by the platform, it can perform its own initialisation to become ready to receive requests from the end users. This initialisation could include, e.g., registering the service interface to the local RMI registry[2] or adding a service specific servlet into the local HTTP server, which is available as a core service. The platform does not have to know about the service specific functionality, it is enough that all services can be started and stopped and otherwise managed via a common interface.

The services are assumed to be bundled into JAR (Java ARchive) files. These files contain all class files of the services and other files that the service might need (e.g. bitmap images). The file must also contain some information for the platform, for example the platform must find out the “bootstrap” class of the service (the one that implements the general service interface) so that the correct class can be instantiated and started. This kind of information can be stored into separate properties files contained in the JAR file. The properties files contain simple name-value pairs in plaintext form, which can be read and parsed by the platform at download time. Furthermore, the JAR file must also contain the certificate of the service, which lists the permissions that belong to the service. The platform must evaluate these permissions and create a security context of them, which will be used at run-time to check that the service has correct permissions for all critical operations it attempts during its execution.

1. A Java interface is basically a class whose methods are all abstract, i.e. without implementations. A concrete class derived of the abstract class must implement these methods, which makes it possible to handle the concrete class instances via the methods of the interface. The user of the class (in this case the platform) need not know the true type of the class.
2. Java RMI (Remote Method Invocation) registry is a repository for objects that can be called from another Java process over the network, for example from the client component running in the end user terminal.

6.3 Platform classes

The prototype platform design consists of four interacting classes: `ServiceManager`, `ServiceClassLoader`, `ServiceContext` and `Service`, each belonging to the package `calypso.platform`. The service manager, according to its name, manages all services downloaded and installed into the system and creates initial contexts for the new services. The context consists of the service context object and a classloader instance, which are strictly one-to-one mapped to the service. The services must extend the abstract base class `Service`, which is an common class for all system and end-user services. To further characterise the roles of these classes it can be said that the service base class and service context class together form an interface to the platform for the service developer, whereas the classloader is an invisible worker behind the scenes, partly managing the security of the platform among other things.

The role of the service manager (together with the context objects) is to define the interface through which services can be installed and managed by a surrounding host application. The host application and the platform are conceptually separated to make it possible to embed the platform in various kinds of other applications. For example, in some applications the services may be installed by the commands given by the user via a user interface, whereas others might download and execute services according to some internal logic or schedule. Yet another possibility, considered especially in the Calypso architecture design, is to have a remote interface provided by a server process for “injection” of new services into the network nodes. For the first prototype, a very simple host application was developed to make it possible to install new services into the system by typing commands on a simple command line user interface. The application and examples of service implementations are described in Appendix B.

Figure 4 is a simple UML class diagram of the class hierarchy of the platform classes. The core classes are denoted with a thicker border line than the base classes of the Java API or the inherited classes of the applications (the concrete service implementations). The names of the abstract classes[1] are written in *italics*. The notation “<<Singleton>>” denotes that the class in question implements the Singleton Pattern. The subsections of

1. Abstract classes in Java have non-implemented methods that must be implemented by the subclasses derived of the abstract class. Compared to interfaces, abstract classes may also have methods with implementations

6.4 describe each of the platform classes on the interface level (the public methods of the classes) and give an outline of the interaction between them.

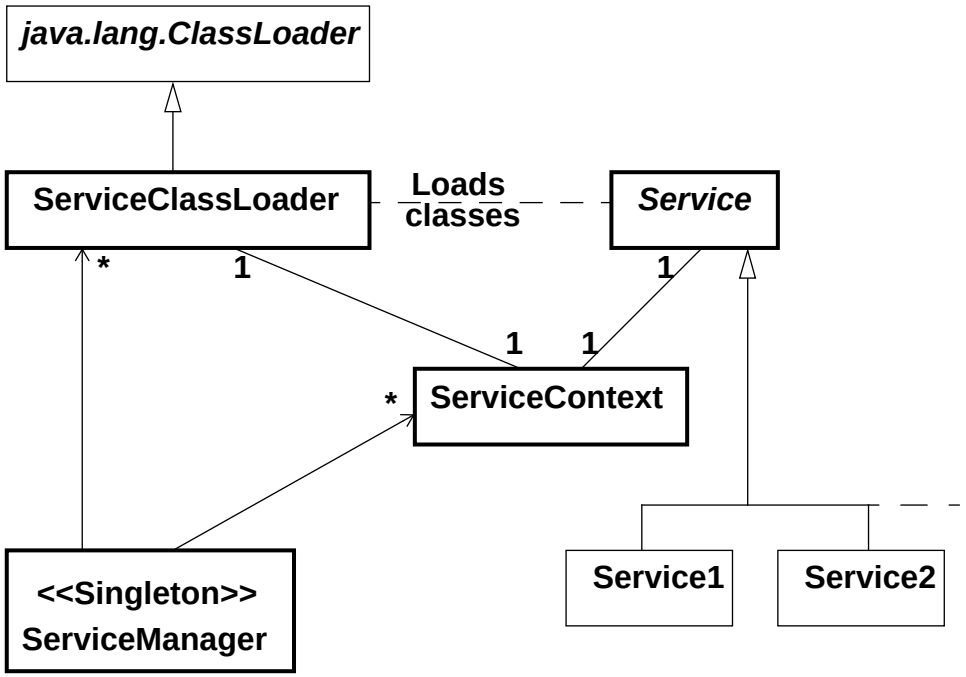

Figure 4 Prototype platform class diagram

6.3.1 ServiceManager

The service manager is a `public final`[1] class implemented according to the Singleton design pattern[2]. The purpose of the class is to keep track of all services that have been downloaded to the system and it is possible to register new services only via the interface of the service manager. The interface contains two public methods that are used to create a new service to the system and to get a reference to a registered service. The methods are defined as follows:

```
public ServiceContext createService(java.net.URL serviceSourceURL_);
public ServiceContext getService(String serviceName_);
```

The `createService()` method takes an URL (as an instance of the Java API class `java.net.URL`) that must point to a JAR file that contains the code of the new service. The method does not, however, download the service at once, but only creates an initial context for the service, returned as a reference to a service context object

1. A “public final” class as a Java idiom means that the class is visible to any other class in the same virtual machine, but it cannot be used to derive new subclasses by inheritance.
2. A Singleton class is a class of which there can be only one instance at any given time in the virtual machine, accessible via a static instance method defined in the class itself.

(which also contains a reference to the associated service classloader instance). The context object can be used to start the service, which causes the code to be downloaded to the platform (see Section 6.3.2 below). The initial context is not yet registered into the service manager, so it is not retrievable via the `getService()` method call until the service has been started, which causes the registration be done.

The `getService()` method can be used to retrieve a reference to an installed and started service in the system. The service is identified by its name and the returned value is a reference to the service context object of the service, if one was found for the given service name. The context object can be used to further manipulate the service, such as to stop its execution or to get a reference to the service implementation.

Because the service manager is a public singleton class, a reference to it can be obtained by any other class (including the classes of downloaded services) and the public methods can be called. This means that in theory the services could be installed by anybody who manages to get service code executed in the system, which can be easily identified as a security risk. It is the task of the security architecture to prevent this (see Section 6.5); however the first prototype does not contain a proper implementation of the security architecture, so it is indeed possible for any service to cause a new service to be downloaded and installed into the prototype system.

6.3.2 ServiceContext

Like service manager, the service context is a `public final` class that is visible to any other class, but can not be used to derive new subclasses. The purpose of the class is to represent a single service downloaded and installed into the system, thus there is always one, tightly coupled service context object for each service[1]. Note that the abstract base class `Service` (see Section 6.3.4 below) also represents a single service downloaded and installed into the system. The difference between these two classes is that the while context represents information about the service used by the platform to manage and manipulate the service, the base class contains common methods and data that the service implementations may call to access the platform resources. The base

1. At the moment there is a strict one-to-one mapping between the context objects and service instances, however it could be possible that one context would host several service instances downloaded from the same JAR file.

class instance is comparable to a process executing on the platform (in the lines of the operating system analogy in Section 5.1.2) while the context object can be seen as an entry in the "process table" of the service manager.

The public interface of the context object is defined by the following methods:

```
public String getName();
public Service getService();
public void start();
public void stop();
```

The `getName()` method simply returns the name of the service as an ASCII string. The name is defined by the service itself and is fetched from the service properties file (see Section 6.4 about the service JAR file contents). No explicit naming hierarchy is defined, so clashes are possible if services are not properly named. A recommended convention is to use the same kind of naming as with Java classes, i.e. starting the name with the inverse domain name of the owning organisation (for example, "`fi.hut.tml.DemoService`").

The `getService()` method returns a reference to an instance of the actual service class that extends the service base class. This reference can be used by other services to access the API of the service. This can be done by *typecasting* the reference from the base class type to the concrete type of the actual service and using the methods of that class to access the service. An example of this is given in Appendix B, where an example of service implementation is given. (Note that the `getService()` method is meaningful only when there is a one-to-one match between the context objects and services. If there were many service instances per context, they could not be referenced in this way.)

The `start()` and `stop()` methods execute and terminate the service, respectively. The `start()` method is meant to be called by the creator of the service after the `createService()` method of the service manager has returned the initial context object. Starting the service creates a new thread of execution to download the service JAR file and to instantiate and initialise the service. The purpose of the thread is to protect the platform from blocking due to a slow or broken network connection or exceptionally heavy initialisation procedure of a service. The thread first downloads

the service JAR file using the service classloader, creates an instance of the service class (the one that extends the service base class), binds the instance to the context object and calls the initialisation method of the service. When the initialisation is complete, the service context object is registered to the service manager, the initialisation thread finishes and the service is ready for use.

The service remains registered in the service manager until the `stop()` method of the context object is called, which causes the service to be informed of the termination via a call to its `doStop()` method. This lets the service perform any tasks it needs to do before stopping. It must be noted that stopping the service does not guarantee that it is really terminated, for example in the current prototype the service may continue executing any threads it chooses to and other services may retain references to the stopped service. The platform only guarantees that the `doStop()` method of the service is called and that no other services are able to load the classes of the stopped service any more, if it had been exporting classes via the service classloader. In particular it cannot be guaranteed that all objects belonging to the service are removed from the system at once (or even eventually) unless the service takes care of freeing its resources itself. After the `stop()` method of the service returns, the service is unregistered from the service manager and it is no more accessible by other services.

6.3.3 ServiceClassLoader

The classloader is probably the most complex of the platform classes. It is a package scope class[1] that is used both by the platform classes (e.g. by `ServiceContext` during the initialisation of a new service) and by the Java virtual machine during the execution of a downloaded service when the classes of it are instantiated (see discussion on Java classloaders and namespaces in Appendix A). The classloader is also used indirectly by the service itself, when it accesses the files that were contained in the service JAR file or the properties that the service was configured with by its author. These tasks are performed via the service base class however, because the classloader is not visible outside the platform package. The classloader also has no public method interface.

1. Only visible to the other classes in the same package, i.e. the other platform classes.

In addition to loading classes and other files from the JAR file the service classloader is a central component in the basic service security. First, it is the task of the classloader to find and verify the certificates contained in the JAR file and grant the service run-time permissions based on these certificates and the local policy configuration. Second, the class loading and definition is a security critical operation that must be done carefully in order not to open up security holes into the system, see discussion e.g. in (McGraw and Felten 1999). The class loading algorithm of the prototype service classloader is defined as follows:

1. If the requested class had been loaded by the current classloader before, use the cached copy of the class. The caching is performed automatically by the default Java classloader functionality, so there is no need to re-implement any caching in the service classloader.
2. If the class is a system class (e.g. it belongs to the `java` or `javax` packages), delegate the loading to the system classloader. If the class is not found by the system classloader, conclude that the class was not found. This is to prevent services from defining their own classes into the system packages to circumvent the member visibility restrictions or to replace classes of the Java API.
3. If the class is a platform class (e.g., `calypso.platform.Service`), use the system classloader to load the class. This is to maintain type compatibility between the platform and the services. The platform classes are the boundary between the system namespace and the service namespace, which must agree on the type of the classes (see discussion on classloaders and namespaces in Appendix A). If the class belonged to the platform package but was not found, the search is not continued for the same reason as in step 2.
4. If the package of the requested class has a classloader registered in the service manager, use it to load the class. This is to maintain type compatibility in a situation where one service uses the classes of another, in which case the namespaces between the two services become partially overlapping.
5. Finally, if the class could not be found anywhere else, try to load the byte code of the class from the service JAR file, and if found, define the class using the system-supplied `defineClass()` method.

The services are able to *export* some of their classes via the service classloader. The service itself does not have to do anything special for this, the only requirement is that is has a permission that allows the classes in a named package (and its subpackages) to become visible for other services. This permission is verified by the service manager at the time when the service is started and registered. The service manager maintains a list of all packages whose classes are exported by service classloaders, and the classloader of a service that is about to load a class of another service automatically consults service manager to get the class from the correct loader. Accessing the classes of other services also requires the accessing service have a proper permission. (See discussion on the security model and permissions in Section 6.5.)

6.3.4 Service

The service class is the base class for all services to be executed on the platform. The class contains methods that the derived service classes may use either directly or by overriding the method in the subclass. For example, the service may use base class methods to get files from its JAR file or override methods that are called during the initialisation or termination of the service.

The base class does not have a public method interface that could be used by any other class to access the service, because this kind of information is stored in the service context object. Instead, the derived services may define their own public method interface into the derived service class to let other services call them. (Most end-user services are likely not to have an external interface, but on the other hand the core services most often do have one).

The method interface for the derived services is defined as follows:

```
protected final ServiceContext getContext();
protected final byte getFileFromJar(String name_);
protected final java.util.Properties getProperties();
protected final String getProperty(String name_);
protected void doInit();
protected void doStop();
```

The first four get-methods are for the service to retrieve information provided by the platform and are defined as final methods to prevent overriding by the derived services. The first one allows the service to get a reference to its own context object. The

`getFileFromJar()` method allows the service to get the contents of files that were contained in the service JAR file (excluding the class files). The method forwards the request to the service classloader that is able to access the JAR file. The two methods for getting properties allow the service get the values of properties[1] that were given by the service developer and were contained in the JAR file.

The methods `doInit()` and `doStop()` are called by the platform during the initialisation and termination of the service, respectively. They are designed according to the Template Method pattern: part of the initialisation or termination code is provided by the platform and cannot be overridden, but service specific functionality is implemented in separate methods. The methods have empty default implementations so the services that do not need any special procedures to start or stop need not override them.

6.4 Service JAR files

JAR is a standard archive file format defined in the Java specification and commonly used to deliver Java applications consisting of several classes packaged into a single file. The format suits well to deliver services to the Calypso platforms, but some additional information not contained in the JAR file needs to be defined for the Calypso services. JAR files are usually created with a command line tool called “jar” or using the functionality of an integrated Java development environment. JAR file is created simply by packaging a full directory hierarchy into a single file that can be later unpacked or used directly by a classloader (such as the service classloader of the Calypso platform). The JAR file contains all class files and other files in the same directory hierarchy.

In Calypso the platform needs some additional information of the services, such as the name of the service and the name of the class that extends the service base class which is instantiated at the installation. The service author may also have defined some default settings that the service needs at its installation or execution time. These kind of settings are supported by Java in form of properties files. Because the platform has

1. The properties in Java are commonly used to read parameters from configuration files and are expressed as name-value pairs in ASCII form.The properties can be manipulated using the Java API class `java.util.Properties`.

to read some properties of the services, there has to be a file with a well-known name that the platform is able to look for. In the prototype implementation this file must be called `service.pro` and it must be in the top level of the JAR directory hierarchy. The file must contain at least `ServiceName` and `ServiceClass` entries for the platform to be able to install the service. The developer may use the same file for the service specific entries or they can be written into separate files accessible via the method `getFileFromJar()` of the service base class. The properties in the default file are accessible directly using the `getProperties()` and `getProperty()` methods of the service base class.

The JAR file also contains the certificates that are used to grant the service its run-time permissions. There are many approaches to use certificates to grant permissions to the services (see discussion in Section 6.5), but in general the certificates are ordinary files that are bundled into the JAR file. The service classloader must be able to find and parse the certificate files in the JAR, so there must be e.g. a well-known file name suffix for the certificates, which the classloader must be aware of. The prototype platform recognises files ending in "`.spki`" as certificates and parses them to grant the service its permissions. Because the certificate handling code is expected to come from an outside source, such as TeSSA project of the HUT, the prototype implementation is very thin in this respect.

6.5 Platform security model

The design of the security architecture of the prototype platform is derived from the security requirements in Section 5.2.1. The goal was to be able to grant each service a unique set of fine-grained permissions, based on the certificates carried within the services and the local security policy configuration of the network nodes. The permission-based fine-grained security is adopted from the Java 2 security model, whereas the policy management scheme is a result of ongoing research in the Helsinki University of Technology.

6.5.1 Java 2 security model

In Java 2 the run-time access control is implemented using a capability based model. Every system resource or critical operation is protected by a *permission*, an object

whose possession indicates ability to perform the operation or to access the resource. The permissions can be very fine-grained, e.g. allowing read access to a named file or allowing to open a network connection to a certain IP address. New application specific permissions can also be easily defined using inheritance. Each class in the system belongs to exactly one *protection domain* that is an object that contains the permissions for all classes in that particular domain. The classes are not able to change or modify their protection domain themselves. The run-time access control is enforced by *access controller*, an object that is called by the system code at the time when a critical operation is invoked by an application class. The access controller checks the current execution context (through which classes the critical method call has been invoked) and checks that all classes in the method call stack have the needed permission by inspecting the protection domains of the classes. If the operation is not permitted by the permissions in the protection domains, a run-time exception is raised and the operation fails. The access controller can be also called from application code to check for application specific permissions derived of the default Java permission classes.

The other half of the Java 2 security architecture is the security policy management, meaning how the classes are assigned to the protection domains. The default way in Java 2 is to identify *code sources* with which the protection domains are associated. A code source is defined by the URL from which the code was downloaded and the signer (or signers) of the code, identified by their public keys. The *policy* of the system is defined in a local configuration file, in which the code sources are given the permissions the user or the system administrator sees as appropriate. To summarise the Java 2 architecture: when a class is downloaded by a classloader, it checks the source URL and signer(s) of the class, matches them to the local policy configuration file and assigns a correct, immutable protection domain object to the class. The run-time system permits or denies operations based on the permissions in the protection domain of the class that invoked a critical operation. See (McGraw and Felten 1999) for in-depth discussion about Java 2 security.

6.5.2 Policy management and certificates

In principle policy management consists of two parts: authentication (i.e. identifying the author or origin of the service) and authorisation (i.e. granting the service the run-time permissions). Authentication is generally easily solved with cryptographic signatures and certificates. The author of the service signs the class files using public key cryptography and lets the receiver of the code (the platform in this case) to verify the signature with the author's public key, which is contained in a certificate delivered with the service or otherwise obtained by the receiver. If the certificate is valid and the signature matches the public key in the certificate, authentication is successful and the service can be authorised to execute on the platform. The validity of a certificate is usually verified through a *certificate chain*[1] that ends to a trusted root, either a public Certificate Authority as in traditional X.509 model or the verifier itself in the newer models. There are two basic approaches to use certificates as a basis to authorise services. The details of these models, the "traditional" model and the "extended" model, are covered extensively in (Nikander 1999), so only brief descriptions are given here.

The traditional model is based on X.509 certificate infrastructure, in which the certificate identifies the author of the service. The author has signed the service code with his private key and includes the signature to the JAR file along with a certificate that identifies the corresponding public key. When the service is downloaded, the signature is verified with the public key in the certificate, the certificate is verified to be signed by a trusted third party (such as a company whose business is to act as a Certificate Authority, e.g. Verisign) and the subject of the certificate (expressed as a domain name, for example) is verified to be the origin of the service. If all these verifications succeed, the platform can conclude that the service can be trusted to the extent of the local security policy and it can be granted the permissions listed in the local policy file. Standard Java 2 security architecture is based on this type of certificate usage.

1. The certificates are chained by their signatures. If the signer (issuer) of a certificate is the subject of another certificate, these two certificates form a short chain, which can be extended with other certificates. Because signing a certificate always expresses some kind of trust placed on the subject, a certificate chain starting from a trusted root can carry that trust to the end of the chain.

The extended model takes advantage of so-called authorisation certificates, such as SPKI certificates. These certificates not only identify the issuer and the subject of the certificate, but also explicitly state in the form of authorisation what kind of trust the issuer has placed on the subject of the certificate. The authorisation field of the certificate can be expressed, e.g., as the set of Java permissions that are to be granted to the service at run-time. When applied to the Calypso platform, the model allows the platform to derive the permissions directly from the certificates without need to consult the local policy configuration (the local policy may be still used to further limit the authorisation). This has the advantage of the local policy being more static, i.e. there is no need to change the policy configuration at each network node every time a new service is to be added. This kind of usage of authorisation certificates is the basis of the TeSSA architecture and thus becomes the model for Calypso as well.

6.5.3 Security implementation in the prototype

In the prototype platform the security is implemented as a mock-up in which there are no true cryptographic certificates or even Java 2 permissions in use. There were two practical reasons for this decision. First, the Java 2 platform was not released for Linux (which was the development platform for Calypso project) at the time of the implementation of the prototype. Second, the code of the TeSSA project to handle SPKI authorisation certificates and the Java permissions within them was not available as well (and would have required Java 2 platform in any case). The decision was made to implement the security in the prototype only in a very lightweight manner to be replaced later with the full TeSSA implementation.

The classloader of the prototype is able to recognise certificates in the service JAR files and handle them as if they were authorisation certificates. However, because Java 1.1 does not have the concept of permissions or fine-grained security, not much can be authorised in the certificate. The mock-up certificates are normal property files which contain name-value pairs. Currently there are two entries recognised by the platform: `LegalPackages` and `ExportPackage`. They are related to the class exporting between services: the former tells what other packages the service is allowed to access and the latter tells from what package the service is allowed to export its classes to the other services. If the service tries to access a package not mentioned in the legal

packages entry, the platform denies access to the class. If there is no export entry for a package in any of the currently loaded services, the classes in that package are not accessible by any other service. No other permissions are recognised or other kind of access control is enforced, so the services on the prototype platform are able to access the system resources quite freely. The prototype also does not contain any kind of code signing or authentication mechanism. It would have been possible to add simple signatures into the mock-up certificates and verify them with public keys locally configured into the platform, but this was not considered worth the effort at this phase.

6.6 Prototype platform implementation

The prototype platform developed as a part of this work is a minimum implementation of the four classes described in Section 6.3. As mentioned in Section 6.5, the security in the prototype is very incomplete due to Java 2 for Linux and TeSSA code not being available, but otherwise the prototype is a functional platform that can be used to develop dynamically downloadable services. The service development on the prototype is described more closely with example code in Appendix B.

The prototype is a compact platform for service development, the size of the source code is roughly 1000 lines of Java and the size of the compiled class files is 15 kBytes. Also, as can be seen in the examples in Appendix B, the overhead for the service implementations due to the platform is small, typically only a few dozen lines of code in addition to the actual service functionality. These figures are due to the fact that the platform was designed to be a simple and reusable component, on top of which application specific functionality could be built as services (such as the Calypso core services).

The prototype is by no means a full-functional application development environment and lacks a number of important features such as service persistency, dependencies between the services, updatable services, etc., found in the commercial platforms such as the Java Embedded Server of Sun Microsystems. Instead, the platform was intended to be a proof-of-concept implementation that demonstrates the feasibility of the development and management model for the services in Calypso-like environment.

7 Conclusions and further work

7.1 Summary of the Calypso project

Calypso started as a project to learn more about the service environment of the future broadband networks. The goal was to design an architecture model for services and their execution environment that could be used in service provision in the residential broadband access networks. The method to achieve the goal was to “learn by doing”, by creating prototypes of services and the service environment and testing them in an environment that had all the essential components: an ATM-based network, terminals, servers and control workstations managing the switches.

The work started by defining a layered control model that partitioned the functionality of the network into conceptual layers by the locality of the control functions. The lowest layer was the control of an individual switching fabric, e.g. an ATM switch. The functions at the FCL handle low-level connections of the underlying network transport. The transport was assumed to be connection oriented, because the strict QoS requirements of the multimedia services were not seen to be achievable with connectionless transports. The next layer was the network control layer, where end-to-end connectivity was managed. The separation of the network layer from the fabric layer enabled the usage of TCP/IP protocol suite as the network model using ATM as a QoS-capable data transport layer. The existing models of IP over ATM were abandoned. A simple, semi-permanent IP control network over default virtual channels was designed to carry signalling and service control data using standard IP routing and addressing. The bulk service data could be carried over native ATM virtual channels.

The third layer of the control model was the service control layer, which became the main research subject for the project. The SCL contains functionality for the management of the actual end-user services implemented as distributed, component-based programs. The project proceeded by defining a service architecture that was the Calypso-specific SCL. The architecture defines the components of the service environment: the services, service platform (or the execution environment), programming and security models and finally the service creation environment. The focus was set to the service programming model and the service platform, of which a prototype implementation was made as a part of this thesis.

7.2 Results obtained

The most important results of the project are summarised below.

1. **Component-based service model** was developed as a part of the service architecture. The service platform takes care of the management of the services and provides an execution context for them. However, the platform itself does not contain any specific functionality, but is a general, operating system -like component. In addition to the end-user services, all system level functionality can also be implemented as services. This makes the platform itself a simple component that could be used in other kinds of applications and not only in the Calypso network nodes.
2. **Security requirements and security model** of the Calypso-style service environment is a critical issue. The services are components implemented by third parties and executed on the service platforms owned by the network operator. It is in the operator's interests to protect the platform from faulty or rogue services. However, if there are a lot of service providers, it should be possible without much trouble, such as inspection of all service code. Dynamic installation of service components into the network nodes requires that the platform is able to determine the run-time access privileges for the services at download time and create a run-time security context based on that. The fine-grained security of Java 2 and the delegation-based policy management model adopted from the TeSSA project seem to meet the needs of Calypso.
3. **Suitability of Java for service development** was demonstrated in the platform and service prototypes. The dynamic code loading and built-in security features of Java became useful in the platform implementation and the ease of use and productivity as a well-defined object-oriented programming language make service creation a relatively easy task. Java was demonstrated to be suitable for the system-level services as well, such as the ATM switch control service, HTTP service and connection management service.
4. **Usage of TCP/IP** as a generic networking protocol suite made it possible to reuse existing protocols, tools and applications. TCP/IP was used as a signalling data transport protocol, which enabled the definition of service specific signalling mechanisms.This was seen as an advantage over the traditional ATM / IN-based architectures in which the signalling is implemented as a predefined and relatively inflexible protocol not suitable for all possible applications.

When the work in TOVE and Calypso was started, the situation in the field of networking was not the same as it is now. In the beginning of 1996 it seemed that ATM was the solution for providing quality of service needed by the new multimedia applications. However, the need for ATM is being questioned more and more as the new gigabit- and terabit-class IP routers are becoming available. In the meantime the QoS capabilities and signalling protocols are also being specified for the TCP/IP protocol stack by IETF. In this light it may seem that the results of Calypso are losing significance as there is less need for service specific control functions on the network side if there is no need for a connection oriented data transport layer below the IP layer. Distributing the service functionality to the network nodes has no clear purpose, if the network does not provide any mechanisms for resource reservation that should be done near the edges of the network.

However, the results of the project are best considered as discoveries of new concepts for service provision in broadband networks in general. The distribution of service functionality may be reasonable for other purposes than controlling the network resources. For example, in mobile networks some services may need network side proxies that process the service data before it is sent over the air interface to the mobile terminal. Compression and filtering to save radio bandwidth are examples of such processing needs. Service specific components could also be used to control access to the services, for which there are two possibilities. One is that an ISP (for example) could offer the 3rd party service providers a platform which controls the routing of the ISP clients. Route to a particular service could be opened by the service specific components that authenticate and optionally charge the clients using the method preferred by the service provider. The second possibility is that the routing is controlled at customer end: the set-top-boxes of the customers could have a service platform where the service components could be downloaded to control access to the services. In this case the set-top-box would act as an inverted firewall that would limit the user's access to the network instead of (or in addition to) protecting the user and her home network from outside intruders. In both these cases a Calypso-style service platform could be used to host the service components.

The developed software prototypes demonstrated the technical feasibility of the ideas, but are not of production quality. However, one indication of the concept being on the

right track is that the component-based service model seems to be emerging elsewhere, too. The most notable incarnations of the same idea as the Calypso platform are the Java Embedded Server and the Enterprise Java Beans architecture. The former is a product of Sun Microsystems, a platform for dynamically downloading functionality into small devices. The idea is to install only the components that are needed and uninstall the components that are not used any more to save the limited memory of the devices. Despite of the slightly different application domain, the idea of downloadable services and a thin, general purpose service platform is exactly the same as in Calypso. EJB on the other hand is an application server concept in which the EJB server contains functionality common to all server applications and the application specific functionality is installed dynamically as components. Again, the idea of having a general-purpose platform and application specific components is in principle the same as in Calypso.

7.3 Suggestions for further work

From the technology point of view there is clearly one area that needs more experimentation and prototyping. The principles of the security architecture are clear, but the TeSSA code still needs to be integrated to the Calypso platform. The security model is also one reason why the prototype platform development could be continued, because the integration of the security architecture to the commercial platforms might prove complicated. Otherwise the functionality of the commercial platforms is much more complete and it is doubtful whether it is worth the effort to implement the more advanced features into the prototype platform, such as the persistent services or service dependencies.

Another area that needs review is the business model. It has not been given much thought after the initial considerations and it is unclear whether the assumptions still hold. The decreasing importance of ATM also has effect on the business model. The Internet becomes more dominant even in the real-time services, which means that the services might not need so much functionality in the intermediate network nodes as was assumed in the beginning of the project. The effect of this should be considered and the assumptions readjusted. However, the platform concept might still become useful in different contexts, as was envisaged in the previous Section.

References

ADSL Forum 1998

ADSL Forum. *General Introduction to Copper Access Technologies*. URL: http://www.adsl.com/general_tutorial.html, 1998.

Albayrak 1998

Albayrak S. (Ed.) *Intelligent Agents in Telecommunications Applications - Basics, Tools, Languages and Applications*. IOS Press, Berlin, June 1998.

Armstrong et al 1996

Armstrong J., Virding R., Wikström C., Williams M. *Concurrent Programming in ERLANG*. 2nd Edition, Prentice Hall, 1996.

ATM Forum 1997

ATM FORUM Technical Committee. *LAN Emulation Over ATM Version 2 - LUNI Specification*. July 1997.

Aura et al 1998

Aura T., Koponen P., Räsänen J. Delegation-based access control for intelligent network services. In *Proceedings of the ECOOP Workshop on Distributed Object Security*, Brussels, Belgium, July 1998, pp. 51-56.

Braden et al 1997

Braden R. (Ed.), Zhang L., Berson S., Herzog S., Jamin S. *Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification* (IETF RFC 2205). September 1997.

Haatanen and Tuliluoto 1998

Haatanen T., Tuliluoto J. *FSR switch Application Programming Interface*. Technical Research Centre of Finland, May 1998. URL: http://www.vtt.fi/tte/tte23/fsr/fsr_switch_api-2.0.pdf.

Heinonen 1998

Heinonen V.-P. *Broadband Network Intelligence*. Master's Thesis, Helsinki University of Technology, Espoo, February 1998.

Koponen et al 1997

Koponen P., Räsänen J., Martikainen O. Calypso Service Architecture for Broadband Networks. In Gaiti D. (Ed.) *Proceedings of the 2nd IFIP Conference on Intelligent Networks and Intelligence in Networks*, Paris, France, September 1997, pp. 73-82.

Laubach and Halpern 1998

Laubach M., Halpern J. *Classical IP and ARP over ATM* (IETF RFC 2225). April 1998.

Lazar et al 1996

Lazar A. A., Lim K. S., Marconcini F. Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture. In *IEEE Journal of Selected Areas in Communications 14,7*. September 1996, pp. 1214-1277.

Martikainen et al 1997

Martikainen O., Samouylov K., Zhidovinov M. A Toolkit for Component-based Development of Telecommunication Services. In Gaiti D. (Ed.) *Proceedings of the 2nd IFIP Conference on Intelligent Networks and Intelligence in Networks*, Paris, France, September 1997.

Magedanz and Popescu-Zeletin 1996

Magedanz T., Popescu-Zeletin R. *Intelligent Networks - Basic Technology, Standards and Evolution*. International Thomson Publishing, 1996.

McGraw and Felten 1999

McGraw G., Felten E. *Securing JAVA: Getting Down to Business with Mobile Code*. Wiley, January 1999.

Newman et al 1998

Newman P., Edwards W., Hinden R., Hoffman E., Ching Liaw F., Lyon T., Minshall G. *Ipsilon's General Switch Management Protocol Specification Version 2.0* (IETF RFC 2297). March 1998.

Nikander 1999

Nikander P. *An Architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems*. Doctoral Thesis, Helsinki University of Technology, March 1999.

OMG 1998

Object Management Group. *Interworking Between CORBA and TC Systems*. Revised (final) RFP Submission, October 1998.

Partanen 1998

Partanen J. *Using SPKI certificates for access control in Java 1.2*, Master's Thesis, Helsinki University of Technology, August 1998.

Puro et al 1996

Puro V.-M., Koponen P., Räsänen J., Nummisalo P., Martikainen O. TOVE in Universal Mobile Telecommunications System. In Spaniol O., Slavik J., Drobnik O. (Eds.), *Proceedings of the 2nd Workshop on Personal Wireless Communications*, Frankfurt am Main, Germany, December 1996, pp. 103--111.

Raatikainen et al 1999

Raatikainen P., Martikainen O., Räsänen J., Koponen P. A Control Architecture for a Multidiscipline Switch. In *Proceedings of the 4th IFIP Conference of Intelligent Networks and Network Intelligence*, Moscow, Russia, January 1999.

Rooney 1997

Rooney S. An Innovative ATM Control Architecture. In *Proceedings of the Fifth IFIP/IEEE International Symposium of Integrated Network Management*. May 1997.

Räsänen et al 1997

Räsänen J., Koponen P., Martikainen O. Broadband Network Architectures. In Boyanov K. (Ed.) *Proceedings of the Network Information Processing Systems '97 Conference*, Sofia, Bulgaria, October 1997, pp. 202-214.

Appendix A

This Appendix describes the details of dynamic class loading in Java, which is an important feature utilised in the demonstration platform implementation.

A.1 Java executables

An executable program in Java, unlike many other programming languages, is not a single binary file read into the memory of the computer at once when the execution begins. Java binary consists of a number of individual class files, each containing the code of a single application class, which the Java runtime environment *loads* one by one at the time they are first needed during the execution of the program. This technique is called *late binding*, meaning that the data types (classes) are bound to the executable code modules at run-time instead of compile and link time. Thus different executions of a Java program may cause a different set of classes to be loaded into the memory, which in some cases saves unnecessary work of loading classes that are not actually needed during the program execution.[1]

A.2 Dynamic class loading

In Java the basic idea of late binding has been carried further by including dynamic class loading, which means that a Java program is able to load and instantiate classes that did not even exist at compile time. This feature allows the functionality of the Java programs to be extended at run-time, which is exactly what is being done when a service is downloaded and executed by the Calypso platform. The only compile-time requirement is that some base class or interface type of the dynamically loaded class is known, so that a reference to the class can be created.[2] In Calypso platform this well-known base class for all dynamically downloaded services is the class `calypso.platform.Service`. Even more powerful feature is the ability to create a class by a Java program “on the fly” without it ever being contained in a file. This technique could be used to create e.g. proxy classes transparently.

1. Modern CPUs and operating systems achieve the same result with native code using virtual memory to map an executable file on a disk to a memory address. The file gets loaded into the main memory in small increments during the execution as memory addresses of the program actually get referenced.
2. Note that all Java classes extend, if nothing else, at least the common base class for all Java classes, `java.lang.Object`. Consequently, any Java class can be dynamically loaded and instantiated and references to the objects of the class can be created.

A.3 Classloaders and metadata

In Java the task of loading the class files has been delegated to classloader objects. The classloaders themselves are ordinary classes that extend the Java API base class `java.lang.ClassLoader`. The classloader interface is very simple: a classloader has to implement only one method, whose signature is:

```
public Class loadClass(String name, boolean resolve);
```

The Java virtual machine calls this method of the currently active classloader object at the time the execution tries to create an instance of a class that has not been referenced before. The method takes the class name as a parameter and another parameter that tells whether the class has to be resolved or not (meaning that all classes the class to be loaded references directly, i.e. its base classes, interfaces and the classes of its member variables). It is for the classloader to decide based on the class name, how and where the class can be loaded. The classloader could, for example, load the class via network, load it from a database or even create it on the fly. After the classloader has found the raw byte code of the class (in form of a byte array) it may let the virtual machine *define* the class by calling classloader method `defineClass()`.

During the definition of the class its byte code is verified to be legal Java byte code that conforms to the rules of the Java virtual machine and Java language specifications. From the security point of view this is an important step, because illegal byte code could break the fundamentals of the Java security, which is not acceptable. For this reason the class definition mechanism cannot be overridden by the application developer even if the class loading itself can be extended.

The result of the class definition is a *metadata* object, an instance of API class `java.lang.Class`, whose instances describe loaded classes. The resulting object is returned by the classloader method and it can be used to create instances of the new class or to get information of the methods and fields of the class using the Reflection API of Java. Normally the class instantiation is performed by the virtual machine, but in case of dynamically loaded classes the application code must create the instances by using the methods of the class `java.lang.Class`.

A.4 Namespaces

Each object within the Java virtual machine is associated with the `Class` object that was created during the class definition. Likewise, each `Class` object is associated with the classloader instance that called `defineClass()` method for the class in question. Finally, when a class causes a new class to be loaded, it is loaded *with the same classloader instance as the original class*. The result of these associations and class loading rule is the basis of the *namespaces* in Java.

The `Class` object determines the run-time type of the objects to which it is associated: two objects are of same type if and only if they are associated with the same `Class` object. When the classloaders are considered, it can be seen that if two classloader instances load and define the same class, the objects created of these classes are of different type, because they have a different `Class` instance associated with them. Two classloader instances in the same virtual machine may have different sets of classes in their namespaces, because it is the responsibility of each classloader instance to find and define its classes. Because classes referenced by a class are always loaded with the same classloader instance, a class is able to see only the classes that its own classloader instance is able to see. It is said that a classloader instance defines a namespace, a set of classes visible in the same context. This feature can be used to separate parts of a Java program from each other, which is also how the services in Calypso are separated from each other. If a service tries to load a class of another service, it is not able to do that unless the classloader of the service is allowed to find the class from the other service.

Namespaces do not have to be completely disjoint. Overlapping namespaces can be created by delegating the class loading from a classloader to another. For example, if a Calypso service tries to load a class form another service and it is allowed to do so, the class is *not* loaded and defined directly by the classloader of the accessing service, but it asks for the `Class` object from the classloader of the accessed service. This is necessary to make the two services agree on the common types. If the services communicate by exchanging objects, it would not be possible if the services had independently defined the classes in their private namespaces.

Appendix B

The service development on the prototype platform is demonstrated with two services: a HTTP service capable of supporting Java servlets and a simple test service. The HTTP service can be used by the other services to add their own servlets, which is demonstrated with the test service. The example also contains a host application into which the platform is embedded and which offers a command line user interface for the user to add new services. Because the services themselves are very simple, the source code of the classes is presented here in its whole (excluding the actual functionality of the HTTP server, which takes about 3500 lines of Java source code).

B.1 HTTP service

The service class is derived from the `calypso.platform.Service` base class to make it available as a Calypso service. The rest of the HTTP functionality is implemented elsewhere and is omitted from this example. The service contains an implementation of the `doInit()` method and another method, `addServlet()`, which is the public interface that can be used by the other services to access the HTTP service functionality. The service classes are bundled into a JAR file together with the "certificate" file for the service (`http.spki`) and a properties file (`service.pro`) that contains the entries needed by the platform plus some additional entries for the service specific configuration.

The HTTP service was originally implemented in the earlier phases of the project when the platform or even the design for it was not available. However, it turned out that very little needed to be changed to make the service compatible with the prototype platform. Basically the only new code that was needed is the `HttpService` class shown here - all other functionality remained virtually unchanged. This demonstrates the simple concept of the platform: the platform provides a way to dynamically download and install new services, but the functionality of the services is not constrained by the platform. Very little code is required in the services to interact with the platform.

The source code for the service class (`HttpService.java`) and the configuration files are found below:

HttpService.java:

```
package calypso.core.httpserver;

import javax.servlet.Servlet;
import java.util.Properties;

/**
 * This class binds the Calypso standalone HTTP server to the
 * Calypso platform by extending the platform Service base class.
 *
 * @author Juhana Räsänen / TML / HUT
 */

public final class HttpService extends calypso.platform.Service {
  /**
   * Reference to the real HTTP server. This is static to maintain only
   * one server even if several instances of this service were created.
   */
  private static HttpServer real = null;

  /**
   * Implements the template method of the service base class.
   * Called when the service is initialised by the platform.
   */
  protected void doInit() {
    if (real == null) {
      real = new HttpServer();
      real.setProperties(getProperties());
      real.doInit();
    }
  }

  /**
   * This method can be called by the other services when they
   * wish to register a new servlet into the HTTP server.
   *
   * @param name_ Name of the servlet
   * @param servlet_ The servlet instance
   * @param props_ The servlet configuration parameters
   */
  public void addServlet(String name_, Servlet servlet_,
                         Properties props_) {
    try {
      real.getServletLoader().addServlet(name_, servlet_, props_);
    } catch (Exception e) {
      System.err.println("HttpService: Could not add servlet");
    }
  }
}
```

service.pro:

```
# Configuration parameters for the platform
ServiceName = fi.hut.tml.calypso.core.HTTP
ServiceClass = calypso.core.httpserver.HttpService
# HTTP specific configuration parameters
defaultServerPort = 8000
defaultDocRoot = /home/httpd/docroot
servletRoot = /home/httpd/servlets
servletPrefix = /servlet/
handlerCount = 30
maxQueueSize = 10
serverBacklog = 10
defaultServerHost = localhost
defaultTimeout = 5
mimeTypeFile = mimeTypes
```

http.spki:

```
# HTTP service may export its classes to the other services
ExportPackage = calypso.core.httpserver
```

B.2 Example service

The example service is a trivial test service that uses the functionality of the HTTP service to register a servlet to become available for the end users via WWW. In this scenario the test service could be, for example, a new service offered for the end users. The platform in the network node offers the HTTP service for the other services to register their own servlets and web pages through which the users are able to get information of the service or even access its functionality. When the new service is deployed into the network, it registers its own web interface for advertising the service in a place that is frequently visited by most users (i.e. the local web server in the access switch nearest to the customers).

The service implementation consists of two classes, the service itself and the servlet class that will become accessible in the web. The service overrides only the `doInit()` method of the service base class and within that method fetches a reference to the HTTP service using the service manager interface and registers the servlet to it. Note that the HTTP service must be registered to the platform before the test service is installed, otherwise it will fail to register the servlet. The prototype platform does not have any dependency mechanism for the test service to wait for the HTTP service to become available and then continue the service initialisation.

The implementation of the test service is following:

TestService.java:

```
package calypso.testservice;

import calypso.platform.*;
import calypso.core.httpserver.HttpService;
import java.util.Properties;


/**
 * A simple test service to demonstrate how to create calypso services
 * and how to use other services in the new platform. This service
 * simply instantiates and registers a servlet to the HTTP service,
 * which must be up and running on the platform.
 *
 * @author Juhana Räsänen / TML /HUT
*/

public final class TestService extends Service {
  /**
   * Implements the template method of the Service base class. This
   * is called when the service is started by the platform.
   */
  protected void doInit() {
    // Try to get a reference to the context object of the HTTP
    // service. If the it is not found, the HTTP service is not there.
    ServiceManager sm = ServiceManager.getInstance();
    ServiceContext sc =
      sm.getService("fi.hut.tml.calypso.core.HTTP");
    if (sc != null) {
      try {
        HttpService hs = (HttpService) sc.getService();
        System.out.println("Got a reference to the HTTP service");
        TestServlet ts = new TestServlet();
        System.out.println("Created the test servlet");
        hs.addServlet("Test", ts, new Properties());
        System.out.println("Registered the test servlet");
      } catch (ClassCastException cce) {
        // This exception will occur if the typecast on the first line
        // of the try block fails, in which case the service with the
        // name "HTTP" was something else than waht we expected.
        System.out.println("HTTP service of wrong type!");
      }
    } else {
      System.out.println("HTTP service not available!");
    }
  }
}
```

The test service registers the following servlet to the HTTP service (derived of the examples in the Java Servlet Development Kit by Sun Microsystems):

TestServlet.java:

```
/*
 * Copyright (c) 1996-1997 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE
 * FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING
 * OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * CopyrightVersion 1.0
 */

package calypso.testservice;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * A simple servlet to print output in response to HTTP GET request.
 */
public class TestServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();
    // set content type and other response header fields first
    res.setContentType("text/html");
    // then write the data of the response
    out.println("<HEAD><TITLE>TestServlet</TITLE></HEAD><BODY>");
    out.println("<h1>Calypso TestServlet Output</h1>");
    out.println("<P>If you see this, the service WORKS!");
    out.println("</BODY>");
    out.close();
  }

  public String getServletInfo() {
    return "A simple servlet";
  }
}
```

Finally, the configuration files for the test service are following (note in the certificate how the service must be granted a permission to access the classes of the HTTP service, otherwise it would not be able to typecast the reference to the HTTP service instance to its correct type of `calypso.core.httpserver.HttpService`):

`service.pro:`

```
# The name of the service
ServiceName = fi.hut.tml.calypso.test
# The class instantiated by the service platform
ServiceClass = calypso.testservice.TestService
```

`test.spki:`

```
# The test service is allowed to access the classes of the
# HTTP service package.
LegalPackages = calypso.core.httpserver
```

B.3 Host application

As mentioned in Chapter 6, the platform is not an independent program, but it must be always embedded into a hosting application. The purpose of the host application is to deploy the new services to the platform according to some internal logic or commands given by users via an user interface. When Calypso network nodes are considered, the host application would most likely be some kind of server program running in the control workstations of the network nodes. The server would register a number of locally stored core services (such as the HTTP service or the connection management service) for the other services to use. In addition to that relatively statically configured set of core services the server program would have a remote interface through which the service providers would be able to deploy their end user services to the network nodes, and naturally the network operator would also be able to manage the platforms remotely. Note that the remote management functions and the remote interface for the service providers would also be easily implemented as core services on top of the platform.

The example host application has only an extremely simple command line based user interface, which the users can use to download new services by giving the command “load” to the command prompt followed by the URL from which the JAR file of the new service can be downloaded. The code of the host application in its whole is (excluding the `calypso.util.Shell` class):

Main.java:

```
package calypso.main;

import java.net.URL;
import java.util.*;
import calypso.platform.*;
import calypso.util.*;

/**
 * This class is the main class for Calypso platform.
 * The services are downloaded from a given URL as JAR bundles that
 * contain the service code, its certificates and other files. The
 * platform code takes care of the service downloading, installation
 * and starting. This class is only a simple command-line front-end
 * to the platform.
 *
 * @author Juhana Räsänen / TML / HUT
 */
public class Main {
  /**
   * The main method for the Calypso platform host application.
   * Starts the command line shell that is used to manage the
   * platform (to download new services in this version).
   */
  public static void main(String [] argv) {
    // Create a new Shell instance
    Shell sh = new Shell("Calypso SwC > ", false);
    // Register the command for loading new services onto the platform
    sh.registerCommand("load",
      new ShellCommand("Start a new service from the given URL") {
        public void execute(StringTokenizer st_, Shell sh_) {
          String urlString = st_.nextToken();
          try {
            URL url = new URL(urlString);
            ServiceContext context =
              ServiceManager.getInstance().registerService(url);
            context.start();
          } catch (Exception e) {
            // Catch all possible exceptions. If any of them
            // would not be catched, the shell would be stopped
            // and the platform possibly be left in unstable state.
            sh_.getOut().println("Error loading service: " + e);
          }
        }
      });

    // Finally run the shell forever
    sh.run();
  }
}
```