POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH
INSTYTUT TELEKOMUNIKACJI

Rok akademicki
2001/2002

PRACA DYPLOMOWA
MAGISTERSKA

Michał Konrad Rój

Wprowadzenie do standardu Parlay/OSA

Praca wykonana pod kierunkiem:
dr inż. Jarosława Domaszewicza

....
Ocena pracy

....
Podpis przewodniczącego komisji

Warszawa 2002

Wprowadzenie do standardu Parlay/OSA

Niniejsza praca w zwarty i przystępny sposób opisuje zestaw interfejsów programistycznych (API) Parlay/OSA. API to, pozwalające na tworzenie usług telekomunikacyjnych niezależnym usługodawcom, zawiera szereg elementów takich jak możliwości tworzenia połączeń w sieci, komunikację z użytkownikami, usługi lokalizacyjne, nalicznie kosztów i inne. Interfejs Parlay/OSA jest zdefiniowany w specyfikacjach ETSI oraz 3GPP.

Pierwsza część pracy jest wprowadzeniem do standardu oraz zastosowanego modelu, a także opisem architektury systemu. W części drugiej opisana jest funkcjonalność wszystkich obecnych w systemie modułów. W części trzeciej pracy pokazano prostą usługę telekomunikacyjną stworzoną przy użyciu Parlay/OSA API. Do zaimplementowania usługi wykorzystano język Java, zaś testy przeprowadzono na symulatorze Parlay/OSA firmy Ericsson.

Praca ta ma w dużym stopniu charakter opisowy. Nie omówiono tu przykładów usług, które można zrealizować za pomocą Parlay/OSA. Sygnalizowana są tu jednak ogromne możliwości integracji sieci telekomunikacyjnych i świata technik informacyjnych (IT), które otwierają nowe możliwości dystrybuowania produktów, usług i informacji.

Życiorys

Urodziłem się 19 lutego 1977 roku w Puławach. Swoją edukację rozpocząłem w roku 1984 w Szkole Podstawowej im. Adama Mickiewicza w Puławach. W roku 1992 rozpocząłem naukę w Liceum Ogólnokształcącym im. Adama Czartoryskiego w Puławach. Po maturze, w roku 1996 zdałem pomyślnie egzaminy i zostałem przyjęty na wydział Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Po czterech latach studiów, w roku 2000, złożyłem pracę inżynierską pt. „Implementation of H.323 Terminal" (poprzedzoną publikacją na Krajowym Sympozjum Telekomunikacyjnym w Bydgoszczy). Po obronie pracy rozpocząłem studia magisterskie, skupiając się na tzw. „otwartych API". Teraz, po dwóch kolejnych latach studiów, składam pracę dyplomową magisterską pt. „Introduction to Parlay/OSA APIs" (poprzedzoną, podobnie jak praca inżynierska, artykułem na bydgoskim KST).

WARSAW UNIVERSITY OF TECHNOLOGY
THE FACULTY OF ELECTRONICS
AND INFORMATION TECHNOLOGY
INSTITUTE OF TELECOMMUNICATIONS

Academic year
2001/2002

MASTER OF SCIENCE
THESIS

Michał Konrad Rój

An Introduction to Parlay/OSA APIs

Scientific Advisor:

Jarosław Domaszewicz, PhD

Warsaw 2002

An Introduction to Parlay/OSA APIs

The goal of this work is to introduce the reader to Parlay/OSA APIs in concise way. The Parlay/OSA APIs is a set of programming interfaces that allow independent service vendors creating telecommunication services. The APIs include (among others): call control part, user interaction part, localization, charging and accounting features. Parlay/OSA interfaces are ETSI and 3GPP standards.

The first part of the work is the introduction to the Parlay/OSA information model. Next, whole the APIs' functionality is discussed. Finally, a simple telecommunications service is described. The service has been implemented in Java and tested with Ericsson OSA/Parlay Simulator.

Contents

1 Introduction **1**
- 1.1 Current Trends in Telecommunications Services . . . 1
- 1.2 New Approach to Service Creation . . . 2
- 1.3 Parlay/OSA information model . . . 4
  - 1.3.1 Functional model . . . 4
  - 1.3.2 Business model . . . 5
- 1.4 Parlay/OSA – History, Presence and Future . . . 6
- 1.5 About this Work . . . 7

2 Supporting technologies **8**
- 2.1 Object-Orientation . . . 8
- 2.2 UML . . . 11
- 2.3 CORBA . . . 13
- 2.4 Java . . . 14

3 The Architecture of Parlay/OSA APIs **16**
- 3.1 API-based Protocol . . . 16
- 3.2 Building Blocks . . . 17
- 3.3 Structure of a typical SCF . . . 18
- 3.4 Fault tolerance and Scalability . . . 19

4 APIs' functionality, SCF by SCF **22**
- 4.1 Framework . . . 24
  - 4.1.1 Framework Access Session API . . . 26
  - 4.1.2 Framework-to-Service API . . . 29
  - 4.1.3 Framework-to-Enterprise API . . . 33
  - 4.1.4 Framework-to-Application API . . . 36
- 4.2 Call Control SCF . . . 40
  - 4.2.1 GCCS . . . 41
  - 4.2.2 MPCCS . . . 44
  - 4.2.3 MMCCS . . . 47
  - 4.2.4 CCCS . . . 50
- 4.3 User Interaction SCF . . . 54
- 4.4 Mobility SCF . . . 57
  - 4.4.1 User Location Interfaces . . . 60
  - 4.4.2 User Location Camel Interfaces . . . 60
  - 4.4.3 User Location Emergency Interfaces . . . 60

4.4.4 User Status Interfaces 60
4.5 Terminal Capabilities SCF 61
4.6 Data Session Control SCF 64
4.7 Generic Messaging SCF 67
4.8 Connectivity Manager SCF 70
4.9 Account Management SCF 73
4.10 Charging SCF 76

5 Service Design in Parlay/OSA **79**
5.1 Introduction 79
5.2 Testing platform 79
5.3 Application 80

6 Conclusions **89**

A Acronyms **90**

B Glossary **92**

C Source code **94**
C.1 MyAppEvent.java 94
C.2 MyAppEventQueue.java 94
C.3 MyAppInit.java 95
C.4 MyAppLogic.java 97
C.5 AppCall.java 100
C.6 AppCallControlManager.java 102

List of Figures

1.1 Service Logic / Infrastructure separation 2
1.2 Possible locations of applications in Parlay/OSA 3
1.3 The Parlay/OSA functional entities 4
1.4 The business entities defined in Parlay/OSA 5

2.1 A class in UML 12
2.2 Inheritance in UML 12
2.3 Sequence diagram in UML 12
2.4 Calling local and remote operation in CORBA environment 13
2.5 IDL source – definition of modules, data structures and an interface 15

3.1 Application and gateway: how Parlay/OSA APIs are used 16
3.2 How the gateway is built: the framework and SCFs 17
3.3 Communication between objects in Parlay/OSA APIs 18
3.4 A typical SCF: service manager and other objects 19
3.5 Setting and using supplementary callbacks 20
3.6 Division of the application 21

4.1 Central role of the framework in Parlay/OSA architecture. 24
4.2 Framework event chain 24
4.3 Service profiles - subscription assignment groups (SAGs) relation 33
4.4 Dependence among the call and the call service manager interfaces. 40
4.5 The GCCS' call model 41
4.6 The MPCCS' call model 44
4.7 The MMCCS' call model 47
4.8 The CCCS' call model 50
4.9 The Generic Messaging call model 67

5.1 Application logic – pseudocode 81
5.2 Enabling call notification 82
5.3 The IDL definition of the TpCallEventCriteria structure 82
5.4 Java source code for the MyAppLogic.createOrigEventCriteria() method 84
5.5 Java source code for MyAppLogic.monitorNumbers() – registering events . . . 84
5.6 Event notification diagram 85
5.7 Java source code for the AppCall.callEventNotify() method. 86
5.8 Java source code for the application's logic main loop (in MyAppLogic) 86

5.9 Java source code for the MyAppLogic.translateModulo10() method 87
5.10 Java source code for the MyAppLogic.doRouteReq() method – call routing . . 87
5.11 Java source code for MyAppLogic.doDessignCall() – deassigning from the call 88

List of Tables

1.1 Parlay/OSA specifications family 7

4.1 Service capability features (SCFs), as defined in Parlay/OSA 23
4.2 Framework Access Session API application's primitives 27
4.3 Framework-to-service API SCF provider's primitives 30
4.4 Framework-to-service API framework's primitives 31
4.5 Framework-to-enterprise API enterprise operator's primitives 34
4.6 Framework-to-application API application's primitives 37
4.7 Framework-to-application API gateway's primitives 38
4.8 Generic Call Control SCF's API primitives – called by application 42
4.9 Generic Call Control SCF's API primitives - called by gateway 43
4.10 New MPCCS SCF's API application's primitives over GCCS primitives 45
4.11 New MMCCS SCF's API application's primitives over MPCCS primitives . . 48
4.12 New MMCCS SCF's API gateway's primitives over MPCCS primitives 48
4.13 New CCCS SCF's API application's primitives over MMCCS primitives 51
4.14 New CCCS SCF's API gateway's primitives over MMCCS primitives 52
4.15 User Interaction API application's primitives 55
4.16 User Interaction API gateway's primitives 55
4.17 Mobility SCF's API application's primitives 58
4.18 Mobility SCF's API gateway's primitives 59
4.19 Terminal Capabilities SCF's API application's primitives 62
4.20 Data Session Control SCF's API application's primitives 65
4.21 Data Session Control SCF's API gateway's primitives 65
4.22 Generic Messaging SCF's API application's primitives 68
4.23 Generic Messaging SCF's API gateway's primitives 68
4.24 Connectivity Manager SCF's API primitives 71
4.25 Account Management SCF's API application's primitives 74
4.26 Account Management SCF's API gateway's primitives 74
4.27 Charging SCF's API application's primitives 77

Chapter 1

Introduction

1.1 Current Trends in Telecommunications Services

Today's telecommunications industry has become one of the fastest growing ones. This development is mainly due to introducing new, revolutionary services. Data transmission, mobile telephony and e-everything caused a great impact on people's ways of communication.

Recently, the meaning of the term "telecommunications service" has widened. The traditional, speech telephony services (fixed and mobile) constitute a diminishing piece of the telecommunications cake. The new important actors include: SMS services, WAP, user interaction (e.g. virtual banking, voting), micro payment, and location services. People are more willing to use their devices (e.g. mobile phones) to deal with new tasks. Furthermore, they demand for new services.

Additionally, there had been some changes in the whole organization of the telecom world. Started with the breakup of AT&T in the United States, the changes brought about the competitive market. New enterprises appear. They do not always own their own infrastructure (which is already built), but develop their new ways to the existing one.

Traditionally, the telecommunications world has been quite hermetic. New services are deployed inside telecom domains by a limited number of telecom engineers. Usually, the solutions are not portable and are developed using highly specialized tools. This approach has two major drawbacks:

- **Mass services** Only those services that will be used (and paid for) by a great number of consumers are deployed. Otherwise they are not worth investing in. As such, the services are usually quite simple. No niches exist.

- **Slow deployment** Infrastructure-owning telecom operators are usually huge companies, where any idea requires taking decisions on many levels of management. It takes time.

Until quite recently, there was no simple way to allow independent value added service providers (VASPs), sometimes referred to as independent service providers (ISPs), to offer their services directly through the telecommunications network. It happened occasionally, but the VASP had to integrate its system with the telecom's internal systems (e.g. with service

control points in case of intelligent network). This negatively affected VASP's flexibility in creating new services.

1.2 New Approach to Service Creation

So far, many actions have been undertaken to help the telecommunications world to be more dynamic and flexible. It quickly became obvious that it was not possible without "opening of the network", i.e. allowing entities outside the telecommunications operator's domain controlling the operator's resources. Naturally, to some limited and manageable degree.

The idea described here is the "service logic" located the outside telecom domains. The real service is then based in separate system and (somehow) uses the telecom's infrastructure. This is depicted in Fig. 1.1.

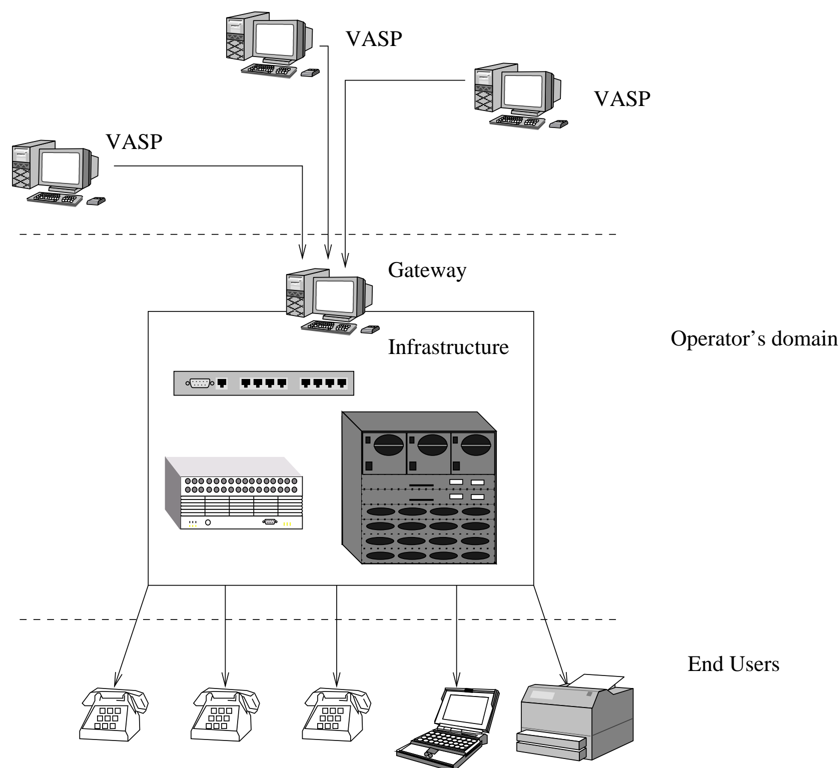

Fig. 1.1: Service Logic / Infrastructure separation

Value added service providers reside at the top part of the figure. They offer their specialized services to telecommunications network users (the bottom part of the figure). The telecom operator is now employed as a kind of broker. To use the operator's infrastructure, VASPs use a certain set of basic features. These features show the VASPs a model, which represents some of telecommunications network functional parts. For example, such a feature can give the VASP opportunity to interact with users willing get access to VASP's databases or localize the user (to find the nearest office, bank, shop, etc.).

The benefits are evident: on one hand, VASPs can offer their services in a new medium, on the other hand, the telecom operator can increase its income from the usage of its infrastructure. Moreover, the services VASPs offer may be too costly or even impossible for the operator to deploy. This seems to be an everyone-wins game. Nevertheless, the major drawback may be the security of the operator's infrastructure. This will be discussed later.

The solution described here, called Parlay/OSA APIs, is now practically the only mature and already-being-deployed standard. The standard, which is also widely supported by major telecommunications players. Service logic in this architecture is located here in so-called *client application* or simply *application*. The application communicates with the telecom operator by means of a special protocol, which is actually a strictly defined distributed application programming interface (API). This API allows the telecom operator to give some restricted access to its infrastructure for authorized applications. Additionally, the API provides a special model of the telecommunications processes, which hides many complex aspects of telecommunications systems architecture. On top of that, the model is portable, i.e. applications can cooperate with various telecommunications networks (e.g. fixed, mobile, 2G, 3G, etc.).

Despite the API described here was mainly developed to "open up" telecom networks, it can be used as well inside the telecom domain. Fig. 1.2 a) and b) compare the two cases.

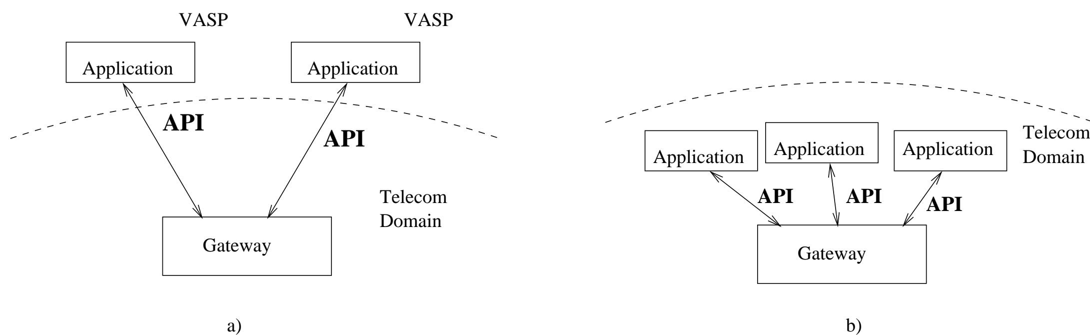

Fig. 1.2: Possible locations of applications in Parlay/OSA

Using a high level programming interface may be much simpler than using lower-level protocols/APIs to create telecommunications services (although they are more powerful). Obviously, services deployed in "vertical" manner, that is, when the infrastructure is altered in many levels/layers, could be, owing to multiple optimizations, very efficient. But it would be also extremely slow, and services would be difficult to extend [1]. Besides, the hardware manufacturers have already started to produce gateways which may be controlled by the use of Parlay/OSA APIs. Therefore, a single telecom operator does not have to develop its own solutions. In addition, the first Software Development Kits (SDKs) are being introduced (e.g. Appium GBox [2]) as well as Parlay/OSA gateways (e.g. Ericsson Jambala [3]) and network simulators (e.g. Ericsson OSA/Parlay simulator [4]). It can make development of the services simpler and more robust, even inside telecom domains.

1.3 Parlay/OSA information model

This section describes the the Parlay/OSA information model in both functional and business layers.

1.3.1 Functional model

The information model in Parlay/OSA consists of four types of actors, which are usually referred as *Parlay/OSA entities*. All entities are shown in Fig. 1.3.

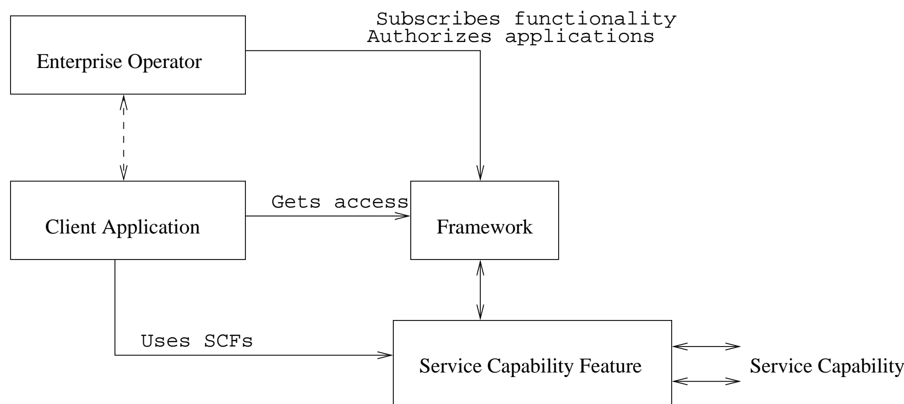

Fig. 1.3: The Parlay/OSA functional entities

Their functionality is as follows:

- **Service Capability Feature (SCF)** This entity is responsible for supplying a certain part of the network functionality. The supplied functionality, or a *service capability*, is a narrow, separated piece of network capabilities, e.g. a user interaction functionality or a call control functionality.
- **Framework** This is the "heart" of Parlay/OSA APIs. It collects one or more SCFs and presents them to other entities. From the other side, the framework is a place where enterprise operators subscribe network functionality and client applications get access to SCFs.
- **Enterprise operator** This entity is responsible for subscribing some network functionality for client applications, and requesting an appropriate quality of service. An enterprise operator does not use the SCFs itself – client applications are entities that use SCF on behalf of enterprise operators.
- **Client Application** This is the entity which uses SCFs and makes use of them to create new value-added services.

1.3.2 Business model

Fig. 1.4 shows one possible business model used in Parlay/OSA. An ordinary font is used for terms denoting business entities, while the Parlay/OSA functional equivalents are written in italics.

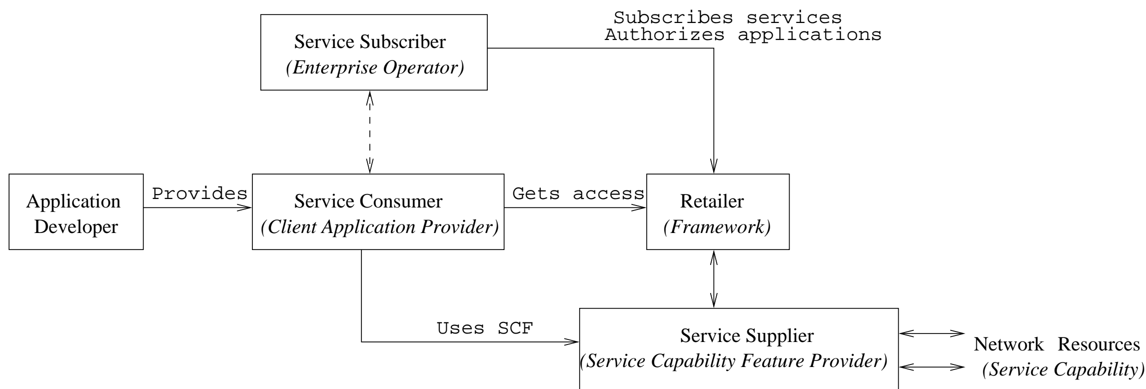

Fig. 1.4: The business entities defined in Parlay/OSA

Each entity in the figure is a separate business but in some cases several entities may be part of a single business (e.g. service supplier and retailer). Specifically, all entities may be part of the *same* carrier. In such case a service subscriber entity may not be needed at all.

Notice that in this context the term *service* means "service capability feature" (SCF), i.e. a piece of functionality provided by service suppliers. This convention is widely used in all Parlay/OSA specifications. However, in this work, which is of a broader subject, the term *service* is used in meaning of "value added service" i.e. the result of client applications' work. To avoid ambiguity, apart from this section, the term *service* will not be used in service supplier meaning, and the acronym "SCF" will be used instead.

Business entities are described as follows:

- **Service Supplier** Provides basic telecommunications services (goods) to service consumers. Typically, this will be a network operator.
- **Retailer** Distributes the services among service consumers in accordance with service suppliers, service subscribers and its own policies. This will be run by a network operator or a recognized institution.
- **Service Subscriber** Subscribes services to (somehow related to it) service consumers. Typically, this is a VASP, it may be for instance a company, bank or an institution.
- **Service Consumer** VASP; uses (telecommunications basic) services. It may be a service subscriber's affiliate or a related (e.g. outsourced) company; contains a business logic.
- **Application Developer** Provides IT solutions to service consumers.

1.4 Parlay/OSA – History, Presence and Future

The Parlay Group was formed in March 1998 by five companies (BT, Microsoft, Nortel Networks, Siemens, and Ulticom)[5] planning to create an interface, which could give secure access to internal telecom operator's space. Since the idea of opening the telecommunications networks had been present quite a long time before the Parlay initiative appeared[1], the working group could make use of existing concepts, and the API version 1.0 (Parlay API phase one) was ready in December 1998. It included generic call control (GCC), messaging and generic user interaction interface sets. As it was still a very early version, two more releases of this phase appeared (1.1 and 1.2). Since Parlay 1.2, published in September 1999, was the most mature release, all first Parlay implementations (like Eurescom P909 [7]) were based on this version. Meanwhile, six more companies (AT&T, Cegetel, Cisco, Ericsson, IBM, and Lucent) joined the Parlay Group.

Parlay Phase 2, released in 2000, widened the API's functionality: Parlay's call control was enhanced by multi-party, multimedia, and conference features and the mobility functionality was added. January 2001 brought version 2.1 of the API, now the most common release, which is the base for many contemporary Parlay-compatible products.

In the same time, ETSI and 3GPP started a joint initiative the define an open API for the 3rd generation networks (the idea was roughly the same as the Parlay's). The APIs were called *Open System Architecture* (OSA)[2]. It briefly became clear that Parlay API could be adapted to fulfill the OSA's goals. OSA APIs were based on Parlay Phase 2 with some improvements and additional interfaces.

Fortunately, the Parlay Group, 3GPP, and ETSI met the agreement, and working together under the name of Joint API Group, they released the OSA version 1, which is known as the Parlay 3.0 APIs (phase 3). That is why the interfaces described in this work are called Parlay/OSA APIs. As for today (10 Sept. 2002) the Parlay 3.1 API is ready, but accessible only for the Parlay Group members. According to the Parlay Group's strategy (discussed in presentation "View of the future" [8]) this one is going to include UML–XML mapping (note below – Parlay X). Now, the Parlay Group consists of over 50 member companies and Parlay/OSA APIs appear to be the only significant standard in this area.

Parlay Phase 4 is expected at the end of 2002. This phase is going to be backward compatible with phase 3 specification (until now, the APIs are still being changed). Parlay 4 is also expected to include mapping to popular telecommunications protocols (what it still lacks).

There are also works on a so-called Parlay X ([9, 10]), which is the XML version of the Parlay APIs. These new APIs could be easily integrated with web services and the Internet, and are being designed to be much more simple for programmers. Parlay X will probably open telecommunications networks to the IT industry event broader than "raw" Parlay/OSA APIs.

The Parlay Group, ETSI and 3GPP specification versions are compared in Fig. 1.1.

The specifications are available from: Parlay (`www.parlay.org`), ETSI (`www.etsi.org`), 3GPP (`www.3gpp.org`). 3GPP specification is more simple than the other two: it does not include multimedia and conference functionality (from call control), nor generic messaging, nor connectivity management. Specification names: ETSI (ES 201 919), 3GPP (TS 29.198). In

[1]For example Telecommunications Information Networking Architecture (TINA) [6].

[2]Now, the OSA acronym is translated into Open System Access

Tab. 1.1: Parlay/OSA specifications family

Date	Parlay	ETSI	3GPP
1998/1999	v1.0,v1.1	-	-
Sept 1999	v1.2	-	-
2000	v2.0	-	-
2000/2001	v2.1	v0.1	v3.x (Rel. 99)
2001	v3.0	v1.0	v4.2 (Rel. 4)
2002	v3.1	v1.1	v4.3 (Rel. 4)

addition, OSA includes protocol mappings: ETSI (TR 101 917), 3GPP (29.998).

1.5 About this Work

This work was designed to present Parlay/OSA APIs in a concise and precise manner, and to save the reader hours of digging through the Parlay Group / ETSI standard documents. Comparing to them, this work is less formal but claims to be more easy-readable. On the other hand, this is complex enough to show the functionality of any single service capability feature provided by the API and, additionally, show the general architecture and procedures used in Parlay/OSA standards. In the author's view, it fills the gap between overall journal articles (which, still, were the great source of the information for the author) and the APIs definition.

Chapter 2, introduces the reader to some supporting technologies that are used by the Parlay/OSA standards (e.g. UML, CORBA), and those ones that seem to be essential in applying the API (e.g. Java). This theoretical chapter gives also the quick view of objects-oriented modelling to readers not familiar with objects.

The description of how the API-based systems are built can be found in Chapter 3, while functional issues are discussed in Chapter 4.

Chapter 5 is about the general idea of how Parlay/OSA applications should be created. A working client application implementation is introduced here. The application was designed for the Ericsson OSA/Parlay Simulator and tested within this environment. The source code is provided.

It should be also noted, that this work shows especially the application developer's point of view. Nevertheless, it does not discuss the professional techniques of telecommunication service design and development, which are now a very broad area of knowledge, too broad for the space limitations of this thesis.

Appendices include acronyms (Appendix A), glossary (Appendix B), and the full source code (Appendix C).

Some aspects discussed here were presented in cooperation with Dr. Jarosław Domaszewicz in the article "Service Creation with Parlay/OSA API" [11], which was submitted (and presented) at the National Telecommunications Conference in September 2002.

Chapter 2

Supporting technologies

This chapter describes the technologies to which Parlay/OSA APIs are strictly linked. Although the APIs are not entirely technology-independent, the supporting technologies are ubiquitous, and being broadly used and accepted by the IT community. They are standards with support from many organizations and companies, with free tools and platforms available. Note, that those technologies have not been the case for older service creation approaches.

First, we discuss object-oriented modelling approach (see J. Rumbaugh's et al. work [12] for more details). Note that we focus on those aspects that are present in Parlay/OSA APIs.

Next, elements of Unified Modeling Language (UML) are introduced. Since it is not possible to learn UML from this chapter, the author provides some bibliography here.
To start with UML, the famous "UML Distilled" [13] could be a good choice. The UML authors' "UML: User Guide" [14] and other books of this serial are great sources of information for every UML practitioner as well. For all those who plan to develop applications with UML, the Bernd Oestereich's book [15] could be a good starting point. The UML specification [16], although quite complex, may be also useful.

The section that follows describes CORBA and middleware-related issues. Middleware is an essence of Parlay/OSA APIs, because it allows application programming interface to be employed as telecommunication protocol. For more information about CORBA, refer the OMG's web page [17]. The Orfali's et al. work about distributed objects [18] may be helpful here.

Finally, some features of the Java programming language are discussed. The author referred on-line available Bruce Eckel's "Thinking in Java" [19], Sun's Java 2 documentation [20] and The Java Tutorial [21] while working on this thesis.

2.1 Object-Orientation

To fully understand Parlay/OSA APIs one has to be familiar with the object technology. This section briefly introduces the reader to objects. The object-orientation described here is the "Java OO"[1], which sometimes differs slightly from other "OOs", e.g. the "C++ OO". All examples are given in Java.

[1]OO is the acronym for object-orientation or object-oriented

Objects were introduced to easily identify and control various items (like people, things or phenomena). In any approach to objects there is a number of special features, which are discussed below:

1. **Classification**
   Any object can be a member of a certain class of objects (which determines the object's type). A class has a signature (or a name), which makes it distinguishable from other classes.

   Assume that there are two classes defined: the `Call` class (identified with a call in telecommunications) and the `CallLeg` class (identified with a party in that call). `Call` and `CallLeg` are names (signatures) of classes.

   Now, let us go back to objects. Theoretically, there may exist an unlimited number of objects of a given class. Suppose that the `myCall` object is of the `Call` class. Similarly, `userOne` and `userTwo` are objects of the `CallLeg` class. To state that in Java, we simply put the class's name and then the variable's name:

   ```
   Call myCall;
   CallLeg userOne;
   CallLeg userTwo;
   ```

2. **Instantiation**
   Although `myCall`, `userOne` and `userTwo` were called "objects" above (which is a very common manner), that is not that simple. Variables are rather kinds of handles (or pointers) to objects than real objects.

   Primarily, if they are defined as in the example above, they point nothing. Every object has to be created before being used. To do that, Java uses the `new` operator. The following example shows how it will be used:

   ```
   myCall = new Call();
   userOne = new CallLeg();
   userTwo = new CallLeg();
   ```

   Now all the variables may be called *object references*, because they refer to concrete objects. This example reveals the next important feature of objects: ability to be created in any moment.

   Note that many object references can point the same object:

   ```
   Call hisCall, herCall;
   hisCall = myCall;
   herCall = hisCall;
   ```

   Now all three object references point the object named initially `myCall`.

3. **Object properties**
   A class defines two types of properties attached an object of that class:

   - Behaviour, i.e. what operations can the object do (for instance: the object can route calls)
   - Attributes, or how the object looks like, for example: the call object has the attribute defining a number of call legs attached

   Objects cannot change the behaviour defined by their class. But attributes may be treated as variables of objects, e.g. the `myCall` object may have zero, one or two `CallLeg` objects associated.

4. **Inheritance and Polymorphism**
   Inheritance is an action of applying all the properties of the parent class (the class from which we inherit) to the descendant one. The parent class is usually called *base class* or *superclass* while the child class (the "heir") is called *subclass*, *derived class* or simply *inherited class*. For example, let the `MultiPartyCall` class be a derived class from the previously mentioned `Call` class. Every `MultiPartyCall` is `Call`, so the following action is possible:

   ```
   Call aCall = new MultiPartyCall();
   ```

   Similarly, every method that takes the `Call`-class object as its argument can be also called with the `MultiPartyCall`-class object as its argument. And it must be treated as the `Call`-class object if necessary. The process of using a derived class as its superclass is called *generalization*.

   Let the `Call` and the `MultiPartyCall` have a method called `howManyParties()`. The method returns the number of parties involved in th call. It differs in implementation for the `Call` and `MultiPartyCall` class. Now, imagine the situation, shown as follows:

   ```
   Call aCall;
   if(aCondition == true)
   aCall = new MultiPartyCall();
   else
   aCall = new Call();

   aCall.howManyParties();
   ```

   In this case – which method will be called? The `Call`'s or the `MultiPartyCall`'s one? The answer is that it depends on the `aCondition` variable and, directly, on the type of the `aCall` object.

   This is called *polymorphism*, which allows calling operations objects, even if the operation's implementation is not know in the moment of program compiling.

5. **Abstraction and Interfaces**
   Another fundamental idea of object orientation is *abstraction*, i.e. use of nonspecific, abstract ideas. For example a "fruit" is a general idea expressing an apple as well as a plum as well as any other fruit. There are no fruits in the world that are just fruits. In this case a fruit is an abstract item. There are usually numerous concrete items related to one abstract item. Such relations are created by using inheritance.

   Abstract classes cannot be instanced (object of these types cannot be created with the `new` operator. Actually, they cannot be created at all). In consequence, unless abstract classes are employed as base classes they are useless.

   A meaningful concept in the theory of objects is *interface*. Interfaces describe behaviour only - they contain only public methods but not attributes[2].

   Naturally, interfaces can inherit from other interfaces as well as they can be "super-interfaces". Abstract and concrete classes inherit from interfaces to acquire their functionality. In the nomenclature of Java, when a class inherits from an interface, it *implements* that interface, while when a class inherits from another class, it *extends* the class. Interfaces differ from abstract classes in the fact that more than one interface can be a base interface for its derived class.

2.2 UML

The Unified Modeling Language is a graphical language, which is used to specifying, designing, creating and documenting elements of information systems [14]. It uses object-oriented modelling approach and it can be primarily used when creating/documenting object-oriented systems. Because it is impossible to describe even a small subset of UML here (it is a very complex language now), we will discuss only those elements of UML, which are used in this thesis, especially some types diagrams.

UML defines a broad spectrum of various diagrams. Parlay/OSA APIs make use of a few of them – class diagrams, state diagrams and sequence diagrams. This thesis uses even a fewer number – class diagrams and (mainly) sequence diagrams.

Class diagrams show relations among classes and interfaces in information systems: how they are linked, how they can cooperate and whether they are relatives (by means of inheritance). The essential element in such a diagram is *class*. Its graphical representation is a rectangle divided into three parts, as shown in Fig. 2.1.

The top part is the name of the class (`Class name` here). In the middle part, all attributes of the class and their types are enumerated (`attribute 1` of type `type 1` and `attribute 2` of type `type 2`). The bottom part enumerates all operations that may be performed on the class. The plus sign (+) tells that the attributes or operations are public (could be accessed from other objects). In addition, if there are no attributes or operations in the class, the respective parts are left empty.

[2]Unfortunately, there are some inconsistencies among existing object applications. Some of them allow interfaces possess attributes (e.g. OMG IDL), while others do not (e.g. Java)

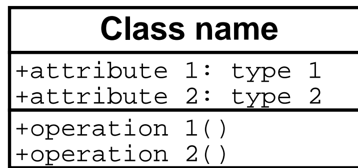

Fig. 2.1: A class in UML

Inheritance between classes is shown by an arrow from the inherited class to the base class, as shown in Fig. 2.2.

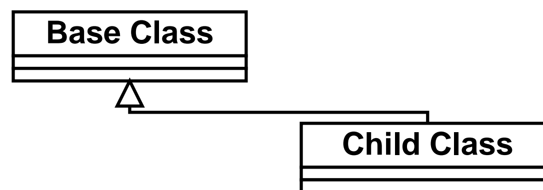

Fig. 2.2: Inheritance in UML

If the class is an interface it can be marked with the keyword `<<interface>>`. Such "«" and "»" bracketed keywords are called in UML *stereotypes* and are used to specify details about items or operations.

The next type of UML diagram is *sequence diagram*. Sequence diagrams were introduced to allow showing what is happening to the system and objects in time. Such diagrams contain objects and show operations undertaken on those objects. The higher an operation is placed the earlier it takes place. An exemplary sequence diagram is shown in Fig. 2.3.

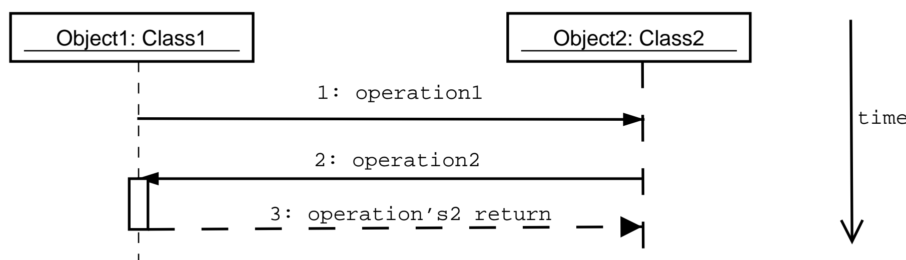

Fig. 2.3: Sequence diagram in UML

Fig. 2.3 illustrates how two objects (`Object1` of the `Class1` class, and `Object2` of the `Class2` class) exchange communicates. They do it be means of calling operations on each other. The first operation, named `operation1()`, is undertaken by `Object1` on the peer object. Although it is not depicted in Fig. 2.3, the operation may be taking parameters or return a value; it can also take some time. Usually, it is enough to place just an operation's name to show how

the system works, other details can be omitted. On the other hand, if there is a need for it, all operation's parameters can be described in the diagram, for instance:

```
String operation1(in String text, in int number, out Class2 reference);
```

instead of simply

```
operation1();
```

The second operation in the diagram, `operation2()`, is called by the `Object2` on `Object1`. This one "blocks" the second object's thread for a while (the blocking is represented by the rectangle on the `Object2` line). After having finished, the returned value is passed to the `Object2`.

Although many commercial UML editors support code generation from UML diagrams, there are no formal mapping from UML to programming languages. Consequently, two different tools could generate non-compatible source code. That is why UML diagrams in Parlay/OSA APIs have rather descriptive purposes.

2.3 CORBA

In quite a narrow sense, the common object request brokerage architecture (CORBA) is a standard of distributing applications across multiple platforms. The standard allows objects to be located on separate systems (processes) but cooperate and communicate as if they were all implemented locally. The CORBA mechanisms are transparent, what means that whole the protocol and network complexity is hidden from the developer's eyes.

To illustrate it in an example, Fig. 2.4 compares calling two operations. Both are called by the object `mainObject` (which is located on `System 1`), but the first is called on the local object (`System 1`) and the other is called on the remote one (`System 2`).

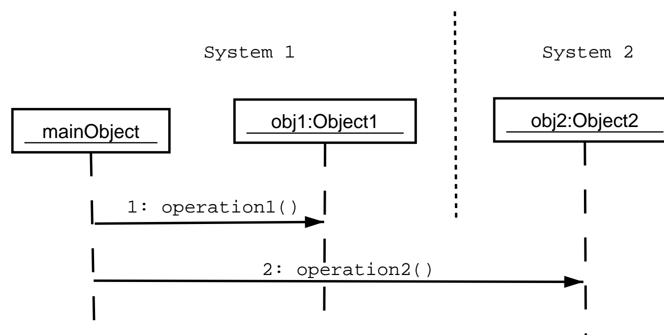

Fig. 2.4: Calling local and remote operation in CORBA environment

For a CORBA programmer both cases may be treated in the same way, no difference where they are located. A Java procedure (`operation()`, a method of `mainObject`), which calls the two operations, may look like below:

```
public void operation (Object1 obj1, Object2 obj2)
{
    obj1.operation1();
    obj2.operation2();
}
```

Looking at this simple program it is even not possible to guess whether the program is distributed. Actually, the difference is in how the `Object2` is defined. This is not shown above since the implementor of the `operation()` procedure does not have to know the application's details.

The architecture of the presented system is as follows: `Object2` and its methods must be implemented on the `System's 2` side. None of objects located on `System 1` need to know *how* the `System's 2` objects are implemented. They are only users of these objects. But there is something the user need to be familiar with. It is a proper interface definition. One of the CORBA essential features is a special language used to define interfaces.

Interface description language (IDL) allows to create prototypes and data types, i.e. all that is needed to define APIs. Since IDL is purely a descriptive language [22], it does not provide any features to write procedures, not at the client's nor at the server's side.

An IDL program must be compiled into a specified programming language (C, C++, Java or any other). As a result, appropriate source files are generated (e.g. `.h` and `.c` files for C), which must be incorporated into the program that uses CORBA objects [3]. Moreover, owing to IDL, the client and server objects may be created using different programming languages.

A sample of an IDL source (taken from Parlay v3.0 IDL) is shown in Fig. 2.5.

The figure shows all the most important features of IDL syntax. Namespaces are created using the `module` keyword (lines 1,2,3), which is translated into `package` in Java. Structures, which are defined with the `struct` keyword (line 4), are simply classes with public elements in Java. Next, `enum`, which is used to define enumeration types (static class in Java) and the `exception` keyword to define exceptions. Finally, an interface is defined in line 17 with the keyword `interface` (also `interface` in Java), with the base interface specified by colon (the `extends` keyword in Java) and a method with one `in` parameter is declared (lines 18,19). The method might raise an exception (line 20).

In addition to what was said before, every program that uses the CORBA system has access to a so-called `ORB` object. This object provides some procedures that may be very useful for CORBA programmer, e.g. naming service (means of resolving objects by textual name and acquiring their references) and many other auxiliary functions.

2.4 Java

Java has been chosen as the programming language for all the examples and the application presented in this work. Since Java programming language is not discussed here, the motivations of using this language are enumerated below:

[3] There are also means to use remote objects without knowing the interfaces. This dynamic way in not used in Parlay/OSA APIs and will be discussed here

```
1  module org {
2    module csapi {
3      module termcap {
4          struct TpTerminalCapabilities {
5              TpString TerminalCapabilities;
6              TpBoolean StatusCode;
7          };
8          enum TpTerminalCapabilitiesError {
9              P_TERMCAP_ERROR_UNDEFINED,
10             P_TERMCAP_INVALID_TERMINALID,
11             P_TERMCAP_SYSTEM_FAILURE
12         };
13         exception P_INVALID_TERMINAL_ID {
14             TpString ExtraInformation;
15         };
16
17         interface IpTerminalCapabilities : IpInterface {
18             TpTerminalCapabilities getTerminalCapabilities (
19                 in TpString terminalIdentity)
20         raises (TpCommonExceptions, P_INVALID_TERMINAL_ID);
21         };
22     };
23   };
24 };
```

Fig. 2.5: IDL source – definition of modules, data structures and an interface

- Java is an object-oriented language that perfectly suits the UML's Parlay/OSA APIs definition.
- Java is very portable. Its virtual machines implementations and compilers exist for every significant platform. The examples given here may be run by broad community of developers.
- CORBA mechanisms and tools are freely distributed with Java Standard Edition [23]. As for other languages (like C++) it is usually needed to choose and install additional ORB libraries.
- Java is very similar to the C and C++ programming languages and the programs should be understandable even for those who do not know Java but are familiar with C-like languages.
- Java is the most frequently chosen programming language for Parlay/OSA-based implementations. In consequence, the author had access to many interesting examples and procedures which, partly, have been used here.
- Ericsson's Parlay/OSA Simulator [4], which is employed as a gateway for the application presented in this work, provides many useful sub-procedures and features which are written in Java (in source code or in `.jar` files).

Chapter 3

The Architecture of Parlay/OSA APIs

This chapter discusses the architecture of the Parlay/OSA-based system, that is: where it is located, and how it is built, how the objects are spread across multiple CORBA hosts and how they cooperate.

3.1 API-based Protocol

For a telecommunications engineer, Parlay/OSA APIs should be treated rather as a telecommunications protocol than as what we used to call an "API". Using interfaces, objects and the distributed platform (CORBA) has greatly increased the protocol's potentials – applications can be deployed with very little effort (no binary compatibility issues, no message encoding/decoding, ...) and the whole system is easily scalable. But still – this API is a telecommunications protocol, which above all gives VASPs an opportunity to communicate with telecom operators.

Parlay/OSA APIs must be used by both the application and gateway to communicate. The APIs (or a protocol in form of a programming interface) may be deeply integrated into application/gateway programs and mixed with other APIs.

Fig. 3.1 shows the exemplary internal structure of the application and the gateway. Note

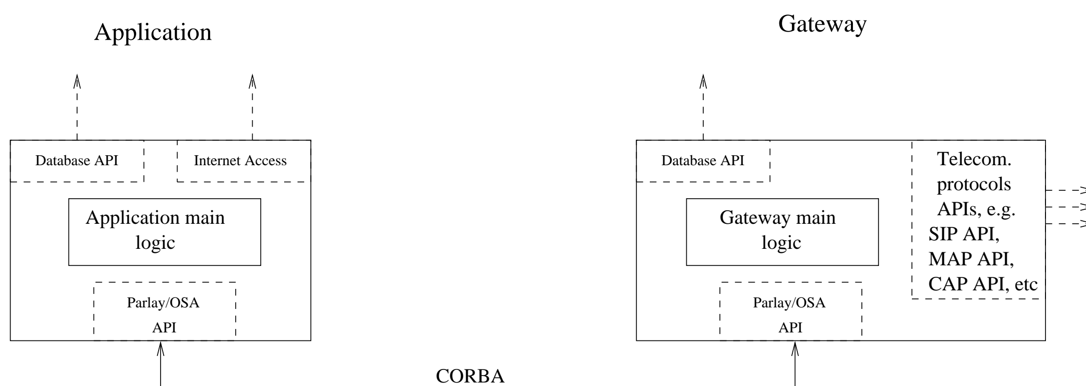

Fig. 3.1: Application and gateway: how Parlay/OSA APIs are used

that all other APIs shown in the Fig. – database API, protocol APIs – are not elements of the Parlay/OSA APIs nor they are defined nor required for the system to work correctly.

The application and the gateway must both support the API (have some objects and procedures implemented), and, until now, it must be the same version of the APIs, from the same author[1]. They communicate over the CORBA system, which allows message exchange (in form of operations called on CORBA objects). The CORBA system is located over a TCP/IP medium, which may be, obviously, the Internet. But to ensure greater security, a dedicated transmission medium should be used [10].

3.2 Building Blocks

The real API architecture is more complicated than the application–gateway model presented above. In the first place, the gateway is divided into one or more so-called *service capability servers* (SCSs). An SCS is a kind of gateway, which provides certain network functionality or, in other words, basic building blocks for telecommunications services' creation. The blocks are called *service capability features* (SCFs) (refer Fig. 3.2). Different SCFs, as defined by

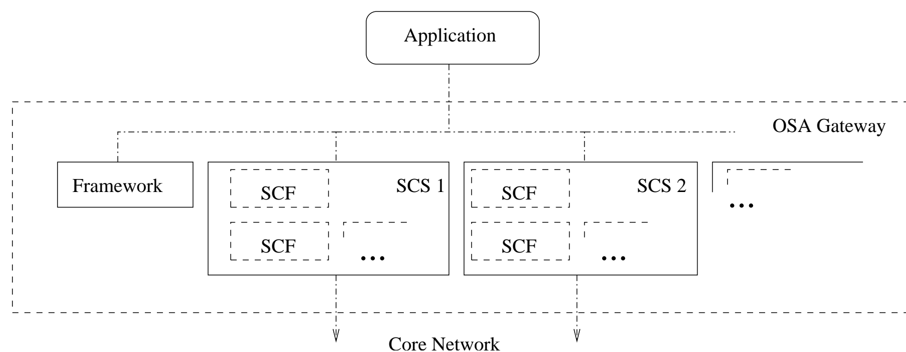

Fig. 3.2: How the gateway is built: the framework and SCFs

Parlay/OSA APIs, have their functionalities separated. For example, one SCF provides user location in mobile networks, while another – the call control feature, and so on. On the contrary, the SCF's functionalities may be combined by the application, e.g. it can localize the user and supervise his/her calls at the same time. The list of all SCFs will be introduced in the next section (Table 4.1).

As seen in the Fig. 3.2, one of the SCSes is called *framework*. A framework is the Parlay/OSA architectural element, which is always present in any gateway. Theoretically, in a minimalistic case, there could be no other SCSs apart from the framework in a gateway. However, the framework provides some very common functionality (like SCF access and authentication capabilities) but does not offer network capabilities itself. Consequently, a framework-only architecture would have no practical sense, because no real service capability features would be

[1]For example, former versions of 3GPP OSA, ETSI OSA and Parlay API version, even if almost identical, will not work together.

offered to the application. In other words, there would be nothing to add value to for a value-added service provider. On the other hand, the architecture would be perfectly correct from the APIs' point of view, which do not specify the number of SCFs in the gateway (it may vary from zero to hundreds of items). What is more, during the application's lifetime, the number of accessible SCFs may change. For instance, in case of a sport event, a new conference SCF may appear to allow transmission of competitions and tele-conference supporting. This example reveals an important issue related to the Parlay/OSA APIs – they define a dynamic system, where SCSes and SCFs may be added and subtracted, and where no strong relation between different SCFs is present. Moreover, the architecture was designed in such manner, that it is possible for various vendors to offer functionalities through a common framework.

3.3 Structure of a typical SCF

Elements of the call model (playing roles as calls, call legs, and other items) are usually represented by two objects: at gateway's and the application's side. The application-side objects are called *callback* objects, because they are often employed as callbacks (from the application's point of view), i.e. their methods are called by the gateway to inform it about a certain events in the network. The application interfaces' names start with the `IpApp` prefix (in contrast to simply `Ip` for gateway objects). Callback objects *have* to be implemented at the application's side. Since their bodies form a part of the application logic, they must be customized procedures, designed and written by application developers. A simple scenario is presented in Fig. 3.3.

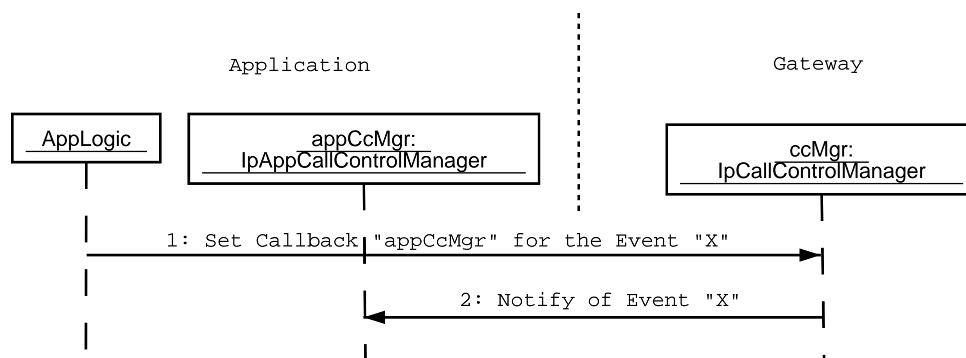

Fig. 3.3: Communication between objects in Parlay/OSA APIs

The `AppLogic` object calls the "Set Callback" operation on the remote `ccMgr` object. It specifies the type of the event (`X`), which is going to be monitored, and the callback object reference (`appCcMgr`). After a certain period of time, the event `X` appears in the network. Then, the gateway uses the previously received object `appCcMgr` to call the "Notify of Event" operation on this object.

This example shows one more essential thing about the APIs. The CORBA object references are exchanged between the application and the gateway by means of method's parameters and returned values. Here, in the first operation, the callback object is passed to the gateway. Similarly, the gateway objects may be (and are) passed to the application by the use of methods.

But in such an architecture there must be one initial gateway's object, to which the application has a reference (or *vice versa* – initial application's object with a reference at the gateway) to be used to start this chain of object exchanging. Parlay/OSA APIs define such an object, and the object is located at the gateway's side. The interface, which is implemented by this object, is called `IpInitial`. The means of how the application can acquire this initial object reference is not defined by the specification. It could be requested from a well-known CORBA nameserver, or transformed from a "stringified" object (compare 2.3), which may be placed on the FTP, LDAP or any other site.

Once the application possesses the object reference, it does not know whether it points a correct object or even whether it is the `IpInitial` one. If so, it does not know whether the `IpInitial` is not a "fake object". To ensure that the reference is the desired one, the application must initiate authentication procedures (actually, it does not have any other choice – the `IpInitial` interface defines the one and only method, `initiateAuthentication()`). After successful mutual authentication (both the gateway and the application may authenticate the peer), the application is given the access to the rest of the framework functionality.

Access to SCFs is given by means of so-called *service manager objects* which are first contact objects with every SCF (Fig. 3.4). Object references to them are passed to the framework by SCF providers, and then applications are given the reference if they request.

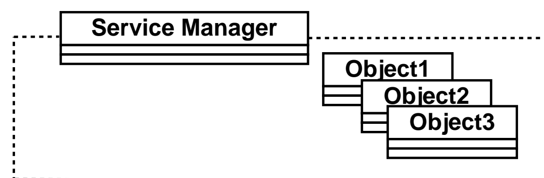

Fig. 3.4: A typical SCF: service manager and other objects

3.4 Fault tolerance and Scalability

The callback technique (augmented by middleware technology) gives the opportunity to create very reliable, efficient and scalable solutions. Initially, Parlay/OSA APIs allow setting more than one callback objects for the same monitoring event in the network. It does not mean that all of callbacks are informed when the event appears, but the second, third, and other callbacks are treated as supplementary callbacks, i.e. they are used only if the primary callback fails. Let us consider an example illustrated in Fig. 3.5.

The application calls the "Set Callback" operation twice, but every time it uses a different callback object as the operation's parameter[2]. When an event appears, the primary callback (`appCcMgrPrimary`) is used to call the "Notify of Event" operation. However, it fails (due to the application crash, overload, etc). The gateway detects the failure (using CORBA mecha-

[2]Note, that despite it is not shown in the figure, these callback objects should belong at least to different processes

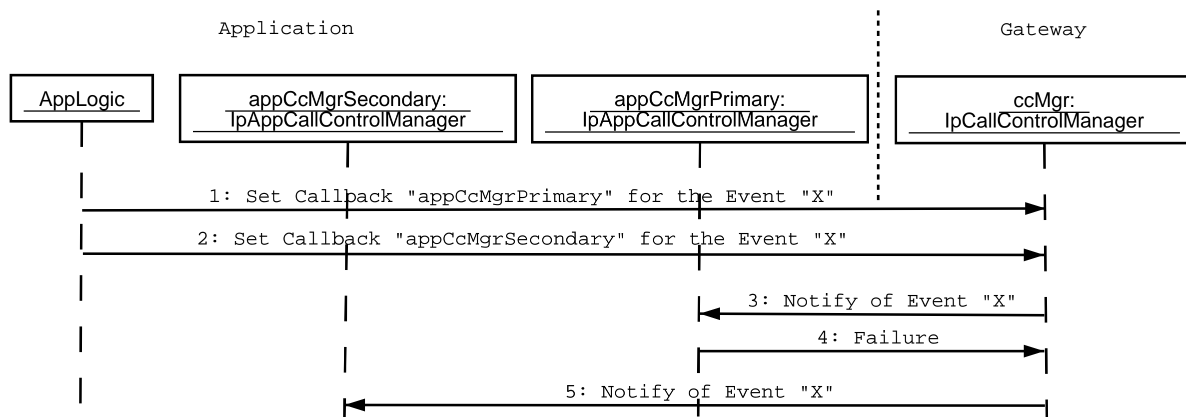

Fig. 3.5: Setting and using supplementary callbacks

nisms) and calls the operation on the second, supplementary callback (`appCcMgrSecondary`). Fortunately, this one succeeds.

Naturally, the two callbacks need not to be the part of the same process (in meaning of operating system). Owing to middleware, they could be (or rather should be) located in separate execution environments or, to improve reliability in greater extent, on separate servers (in physical meaning). This separation can “distribute” the application even more than the gateway is distributed.

This mechanism may be useful when upgrading the system, in software as well as in hardware meaning. For instance, it could be a “hot upgrade”, i.e. without shutdowning the whole system to do that. The primary server can be turned off, in that time all the events are served by supplementary callbacks. In the meantime, the server is being upgraded (e.g. more memory is being added), and finally, the server is turned on and the application properly booted. Similarly, the application can be upgraded by setting new (upgraded) callback objects and unsetting the older ones.

The application may be also divided because the efficiency and scalability reasons. In case of many various events being monitored by the application, especially if they require high computational power, it may be necessary to place different callback objects on separate servers. This could be also applied to the same event but different address ranges, e.g. events concerning numbers starting with `12345` (user "A", among others) are going to be served by `callback_1`, while numbers starting with `12346` (user "B") – by `callback_2`. This case is shown in Fig. 3.6.

Here, the application is built of 2 servers, each implementing one callback object. The gateway detects that the `X` event appeared twice in the network, for user `A` and user `B`. But, in spite of the same event, the users are served by different callbacks. The separation of callbacks does not need to be caused by efficiency issues only. It could be a part of an application logic to apply those users to various callbacks, e.g. depending to what kind of SCFs they have subscribed.

The next important feature of such an architecture is scalability. In any moment the system can be enlarged by one on more servers and the appropriate callbacks transferred or added to them.

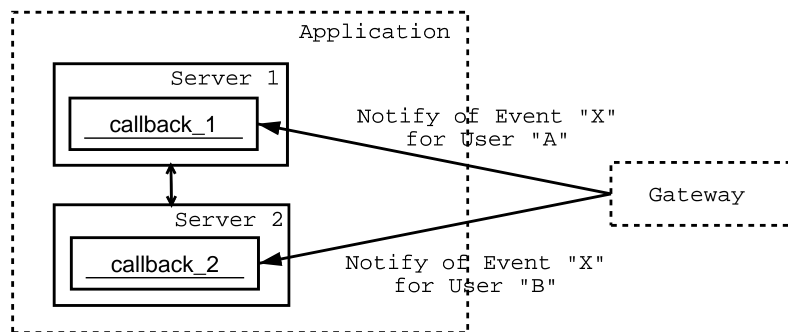

Fig. 3.6: Division of the application

Chapter 4

APIs' functionality, SCF by SCF

Parlay/OSA APIs are defined by a set of documents that describe all the programming interfaces and related issues, including finite state machine's description data structures and relevant information models. Since the Unified Modeling Language has been used to define the APIs, in addition to programming language–independence, the documents are more easily-readable than in case of ordinary API or protocol definitions. However, the APIs are designed among others to system developers and they have to cover any aspect of the APIs. And this definition's complexity motivated the author to write this, descriptive chapter.

The chapter describes all the OSA service capability features (SCFs), as defined in ETSI OSA version 1 (to be more specific: final draft v1.1.1, dated 2001-12) [24]. The SCFs are some pieces of functionality of the telecommunications network, usually very basic ones, which are provided by SCF providers to application providers. Then, the application uses those pieces to create more powerful and more complex services (to put it in another way – the application provider adds value to those basic features supplied with SCFs).

OSA version 1 defines nine SCFs. Parlay v3.0, which includes all the SCFs defined by OSA, adds two more SCFs (which, on the contrary, will not be thoroughly discussed in this work). All the SCFs described here (with respective section numbers) are shown in Table 4.1. The common part of the API, framework, consisting of 4 separate API sets, is described in Section 4.1.

This chapter is designed in the following manner: first goes an overall description of a given SCF (or the framework); next, more specific information are discussed about features, or so-called *primitives*, which are supported by the SCF. The concept of primitives does not come from the specifications. It is introduced here for exposition purposes. In primitive we mean behaviour, which can be requested via the API, e.g. "connect a user with a specified address" or "terminate the call". The primitives are written in English sentences and this is more descriptive than interface operations' names used by the specifications[1]. In this description, some of primitives' names are perfectly equal their corresponding methods, while some primitives are simplified versions (e.g. when a procedure is distributed or if it requires more than one operation call). Some less important (or auxiliary) operations are not presented to keep the description

[1] Usually the features of SCFs are explained in the form of the interfaces' operations. Nevertheless, this approach seems to be too hard for an unexperienced user, especially due to complex data types.

Tab. 4.1: Service capability features (SCFs), as defined in Parlay/OSA

Name	Functionality	Section
Call Control	Generic voice calls, multi-party calls, multimedia calls and conferences	4.2
User Interaction	Interaction with users: sending and receiving textual messages (e.g. SMSes), playing announcements on terminals	4.3
Mobility	User localization and user status (functionality not limited to mobile users)	4.4
Terminal Capabilities	Querying user terminal capabilities	4.5
Data Session Control	Data transmission session management (e.g. GPRS)	4.6
Generic Messaging	Mail, mailfolder and Mailbox management.	4.7
Connectivity Manager	VPN and QoS management	4.8
Account Manager	Monitoring of users' billings	4.9
Charging	Charging user for using the application and other resources	4.10

clearer.

After description of primitives, their typical order is presented. Primitives, which are called by the application are marked with *(App)*, by SCF provider – *(Svc)*, enterprise operator – *(EntOp)*, framework – *(Fw)*. If the described feature may be used by multiple entities it is marked as *(App)* (as the most common case).

Finally, at the end of each description, the interface's names (and a short description) of described API are given.

4.1 Framework

Framework [25] is a part of Parlay/OSA APIs, which is used by all other Parlay/OSA entities. It provides a number of common interfaces (e.g. gateway access) and some auxiliary functionality.

Fig. 4.1 shows how framework is linked to other entities of Parlay/OSA architecture.

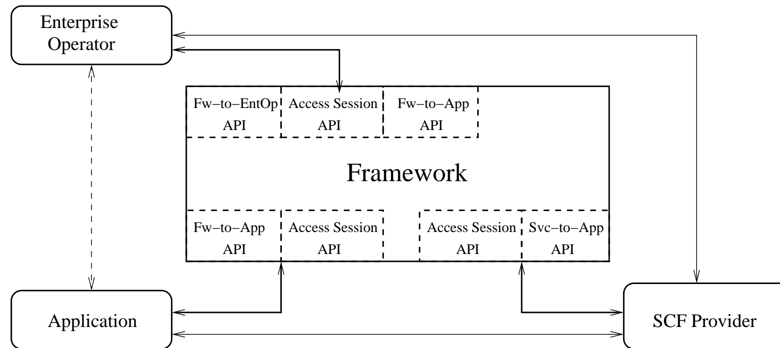

Fig. 4.1: Central role of the framework in Parlay/OSA architecture.

From all three types of Parlay/OSA entities' point of view, the framework is a kind of broker. First, it collects information from SCF providers (any such provider has to register its SCFs into the framework). Then, it stores the information, and, using the stored data, it can grant a direct SCF access to applications. It also maintains the register of which applications are allowed to use which SCFs (as the enterprise operator requests). Finally, it allows SCF providers to set some load policy, i.e. how many applications can use their features.

For applications, it provides many useful features, like application authorization, discovery of existing SCFs and choosing the desirable ones, fault management, etc.

The framework's typical procedure chain is shown in Fig. 4.2. Besides the framework, there

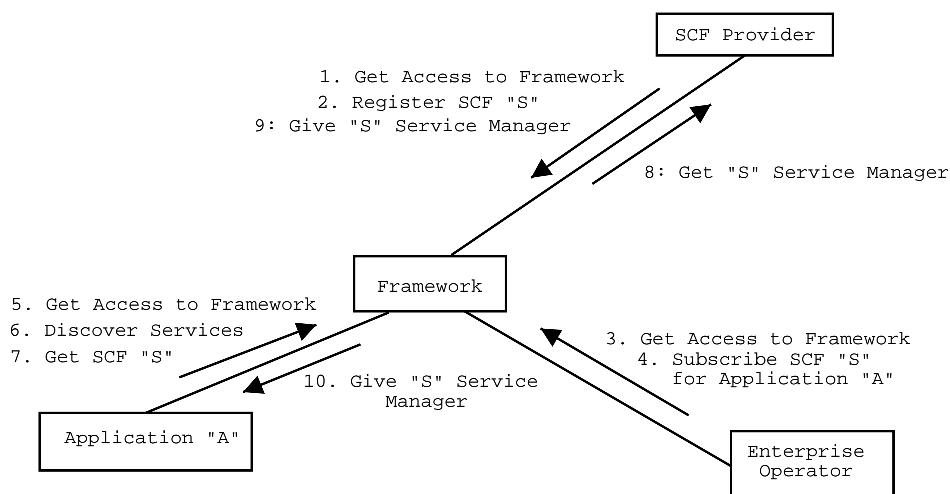

Fig. 4.2: Event chain linked to framework.

are three other Parlay/OSA entities illustrated in Fig. 4.2: an enterprise operator, a SCF provider and an application.

First, every single entity has to gain access to the framework (steps 1, 3 and 5). The SCF provider registers its "S" SCF to the framework (step 2) and since then it is available for other entities. The enterprise operator first subscribes the "S" SCF for the "A" application (step 4). In some cases the framework might follow its own policy for enabling registered SCFs to applications, but this is the typical Parlay/OSA business model's scenario. In step 6, the application discovers SCFs that have been already registered to the framework. It finds "S" in this manner. Next, the application requests access to the "S" SCF, expecting acquiring the service manager object reference (step 7). But before the framework will be able to do this, it has to acquire the reference itself from the SCF provider's object factory (step 8). Finally, the desired object reference is passed to the application.

In the Parlay/OSA model, one framework can "serve" many various SCF providers. Consequently, the application can pick then the most appropriate, for instance the cheapest, solution. Similarly, one application may use many instances of the same SCF, to allow users of many mobile carriers use localization services.

4.1.1 Framework Access Session API

The Framework Access Session API is used by all other types of Parlay/OSA entities: applications, enterprise operators and SCF providers. This is the "first contact" with the framework, and there is no other way to get access to the framework. The application primitives delivered by the API are described in Table 4.2.

Tab. 4.2: Framework Access Session API application's primitives

Primitive	**Description**
Start Authentication	The first operation called on the framework. The authentication, even in case of a trusted party (it could be simplified then), is a mandatory part of framework procedures.
Request Framework Access	Upon successful authentication, it enables the access to the framework and its features.
End Framework Access	Ends the framework session. The entity can no longer access framework interfaces.
List Available Framework Features	This gives a list of framework features which are available at time. Possible features include: SCF discovery, event notification feature, load management, etc. Note, that some of them may be not available (e.g. not implemented).
Request the *X* Framework Feature	Allows choosing and using a specific framework feature *X*.

Typical order of primitives:

1. Start Authentication (App)
2. Request Framework Access (App)
3. List Available Framework Features (App)
4. Request the *SCF Discovery* Framework Feature (App)

Description of interfaces:

- `IpInitial` – this is the framework's initial contact. It is used by applications as well as enterprise operators and SCF providers willing to communicate with the framework. This interface's object reference must be accessible by applications and may be acquired in many ways (e.g. from CORBA name services).

- `IpAPILevelAuthentication` and `IpClientAPILevelAuthentication` – the pair of interfaces used to authenticate the client (an application, enterprise operator or an SCF provider) into the framework and *vice versa*. The additional interface, `IpAuthentication` (the super-interface to the `IpAPILevelAuthentication` interface), defines the means of requesting access the framework.

- `IpAccess` and `IpClientAccess` – the pair representing an access to the framework. `IpAccess` is acquired upon successful authentication.

4.1.2 Framework-to-Service API

This API is located between the SCF provider and the framework. First, it allows SCF providers register the SCFs they offer, so that the SCFs become present in the framework and applications can discover them. The second function is to provide the service manager object reference on any framework's request, giving an opportunity the functionality can be used by many applications (if needed).

Notice that you do not have to be familiar with this API when you want to design and implement only client applications. Nonetheless, it may be useful to know the API to fully understand the system behaviour. Tables 4.3 and 4.4 show the primitives supported by the API.

Tab. 4.3: Framework-to-service API SCF provider's primitives

Primitive	**Description**
Register SCF	This is used by an SCF provider to register a new SCF to the framework. The SCF's name and its properties are supplied.
Unregister SCF	Unregisters the (previously registered) SCF.
Announce / Unannounce SCF Availability	These two primitives are used to tell the framework that the SCF is available (fully functional and ready to be used by applications) or unavailable in the moment. Note that calling the "Register SCF" primitive is not enough to provide the SCF's functionality. SCF availability must be signalled to the framework beforehand.
List SCF Types	Lists all SCF types supported by the framework, e.g. Generic Call Control or Mobility .
Describe SCF Type	This gives the SCF provider properties of a given registered SCF.
Discover SCFs	Get a list of registered SCF instances that have given properties. It is up to the framework whether it reveals only this SCF provider's registered SCFs or any others.
Are you Operational?	This question can be asked to the framework to ensure that the peer is workable.
Set the *X* SCF Unavailable	This primitive turns the *X* SCF off. Now, the framework can no longer offer it to applications.
Check Fault Statistics	This primitive checks the fault statistics for the framework.
Enable Heartbeating	This primitive is used to make the framework give a special signal (a "pulse") to the SCF provider. The pulses are repeated periodically until disabled.
Disable Heartbeating	Turns off the periodical heartbeating.
Check Load Statistics	This checks the load statistics i.e. whether it is in a normal state, or it is overloaded (e.g. cannot cope with forthcoming calls) for applications that use the SCF.
Turn on/off Load Reporting	These two primitives are used to enable the periodic load reporting of the application load.
Report Load Conditions Change	This primitive is called when the load conditions change (e.g. an application no longer overloaded).
What is the Time?	This primitive allows the framework and the SCF provider synchronize their clocks.
Notify of an Event	This primitive is called on the framework by the SCF provider when the SCF has changed its state, e.g. has become unavailable.

Tab. 4.4: Framework-to-service API framework's primitives

Primitive	**Description**
Create Service Manager	This primitive is used by the framework to request the SCF provider to create a new service manager object and return its reference to the framework. It is further passed to the SCF-requesting application.
Destroy Service Manager	Framework tells that a given SCF is no longer used by applications. Now it can be released now.
Report Framework Fault	Informs the SCF provider about a serious fault in the framework. The provider cannot use the framework until it recovers.
Report Framework Recovery	Tells the SCF provider that the previously reported fault is no longer valid. The application can use the framework.
Are you Operational?	This question can be asked to the SCF provider to ensure that the peer is workable.
The *X* SCF is no longer used	This primitive tells the SCF provider that the application stopped using the X SCF (e.g. as the result of a failure). The SCF provider should reestablish normal network event handling.
Check Fault Statistics	This primitive checks the fault statistics for the SCF provider .
Enable Heartbeating	This primitive is used to make the SCF provider give a special signal (a "pulse") to the framework. The pulses are repeated periodically until disabled.
Disable Heartbeating	Turns off the periodical heartbeating.
Check Load Statistics	This checks the load statistics of a given SCF, i.e. whether it is in a normal state, or it is overloaded (e.g. cannot cope with forthcoming calls).
Turn on/off Load Reporting	These two primitives are used to enable the periodic load reporting of the SCF's load.
Report Load Conditions Change	This primitive is called when the load conditions change (e.g. an application or the framework is no longer overloaded).
What is the Time?	This primitive allows the framework and the SCF provider synchronize their clocks.
Turn on/off Event Notification	This primitive enables/disables the notification of SCFs' conditions (is it available, or not).

Typical order of primitives:

1. Register SCF (Svc)
2. Announce SCF Availability (Svc)
3. Create Service Manager (Fw)
4. Unannounce SCF Availability (Svc)
5. Unregister SCF (Svc)

Description of interfaces:

- `IpFwServiceRegistration` – used to register and unregister SCFs into the framework.
- `IpServiceInstanceLifecycleManager` – this interface is provided to the framework by the SCF and is a "factory" of proper service manager objects. When an application requests a service manager (by signing a service agreement), the framework first uses this interface to acquire an object reference of the appropriate service manager and then pass it further to the application.
- `IpFwServiceDiscovery` – discover SCFs which are already registered into framework.
- `IpFwFaultManager` and `IpSvcFaultManager` – these interfaces allow exchanging information about serious problems encountered in the framework and in the application (similar to fw-to-app API).
- `IpFwHeartBeatMgmt` and `IpSvcHeartBeatMgmt`, and specified interfaces: `IpFwHeartBeat` and `IpSvcHeartBeat` – a periodically repeated "pulse", which means: "I am still alive" (similar to fw-to-app API).
- `IpFwLoadManager` and `IpSvcLoadManager` – allows congestion level monitoring of the framework and applications.
- `IpFwOAM` and `IpSvcOAM` – interfaces used to synchronize time and date between the framework and an SCF provider (similar to fw-to-app API).

4.1.3 Framework-to-Enterprise API

This API is used by the enterprise operator only. Enterprise operator is in charge of SCF subscription (for its applications) from the framework. The model supplied by this API is presented in Fig. 4.3.

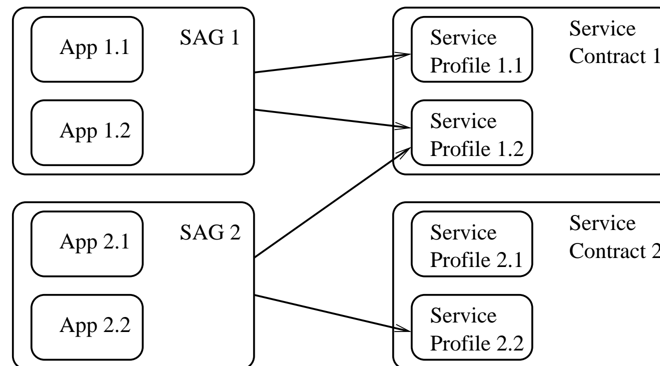

Fig. 4.3: Service profiles - subscription assignment groups (SAGs) relation

Before applications will be able to use SCFs, any single application must become a member of so-called *subscription assignment group* (SAG). A SAG is simply a means of applying the same access privileges to multiple applications.

SCFs, on the other hand, are represented by *service profiles* (the list of properties: the SCF type, start and expiry date, exact values of possible resources, e.g. the number of simultaneous calls), which are collected into *service contracts*. To allow applications from a given SAG access SCFs, the SAG has to be assigned to appropriate service profile (arrows in the Figure). All primitives of this API are shown in Table 4.5. Note, that the "gateway" primitives are not present in this API.

Tab. 4.5: Framework-to-enterprise API enterprise operator's primitives

Primitive	**Description**
Create/Delete Application	This creates (or deletes) a new (framework representation of) application related to a given enterprise operator.
Create/Delete SAG	This creates/deletes a (new) SAG.
Add/Remove Application to/from given SAG	This can be used to apply an application to (or remove from) a given SAG.
Show all Applications of given SAG	This is used to get information about what application have already been applied to a given SAG.
Describe SAG	Returns the description of the SAG.
List all SAGs	Returns a list of all SAGs that have been created.
List all Applications	Returns a list of all applications no matter which SAG they are applied to.
Create a Service Profile	Creates a new service profile (including all related informations).
Assign SAG to Service Profile	Assigns a SAG to a given service profile. Note that no raw application can be applied – even if there are a single application applied to a profile it must be contained by a SAG.
Deassign SAG from Service Profile	Deassign (remove) a SAG from a given profile.
List all Service Profiles	Returns all defined service profiles.
Describe Service Profile	Returns a description of a service profile.
Show all SAGs assigned to Service Profile	Returns all SAGs that have been assigned to a given service profile
Create/Modify/Delete Service Contract	This is used to perform operations on structures called service contracts, consisting of: service requester, billing contact, start/end dates, service name and ID, and service properties. After creating service contract, the proper SCFs are ready to be requested by applications.
Describe Service Contract	Returns all details about a given service contract.
List Service Contracts	List all created service contracts.
Modify/Delete Enterprise Operator Account	This account identifies an enterprise operator within the framework domain. It is created off-line by a framework operator. These two operations allow enterprise operator delete or modify its account only.
Describe Enterprise Operator	Returns some details about the enterprise operator.

Typical order of primitives:

1. Create Application (EntOp)
2. Create SAG (EntOp)
3. Create Service Profile (EntOp)
4. Add Application to SAG (EntOp)
5. Assign SAG to Service Profile (EntOp)
6. Create Service Contract (EntOp)

Description of interfaces:

- `IpClientAppManagement` – this interface allows creating, modifying and deleting applications associated with the enterprise operator. Additionally, it allows whole groups of applications (SAGs) being created and filled with applications. The next interface, `IpClientAppInfoQuery`, allows checking which applications and SAGs are already present in the framework and which applications belong to certain SAGs.
- `IpServiceProfileManagement` and `IpServiceProfileInfoQuery` – these are used to control (create/destroy) service profiles and assign SAGs to them.
- `IpServiceContractManagement` and `IpServiceContractInfoQuery` – these two interfaces are used to manage service contracts (create, modify and delete them).
- `IpEntOpAccountManagement` and `IpEntOpAccountInfoQuery` – these two interfaces are used to make make some operations (data modifications) on Enterprise Operator Account (enterprise operator's representation within the framework). There are no way in the OSA API to create such accounts; they must be created manually or off-line by the framework operator before any enterprise operator can access the framework.

4.1.4 Framework-to-Application API

This API is used by applications and enterprise operators. It defines essential features for those entities like SCF discovery, SCF selection and others. Every application that want to get access to networks functionality (by acquiring an appropriate SCF reference) has to go through at least a few procedures described here. The API's primitives are shown in Tables 4.6 and 4.7.

Tab. 4.6: Framework-to-application API application's primitives

Primitive	**Description**
List SCF Types	Lists all available types of SCFs, e.g. Generic Call Control, User Interaction and others.
Describe SCF Type	SCF type may be not enough. This gives user detailed information about a given SCF.
Discover SCFs	Get a list of registered SCF instances that have certain properties. Note that there could be more than one instance of an SCF returned, e.g. there could be several Generic Call Control providers, everyone in charge of its own number area.
List Subscribed SCFs	Lists all of SCFs subscribed by the enterprise operator until now.
Select SCF	Selects the desired SCF. As the result, a special "token" is returned, which may be later used to sign the service agreement and get access to the SCF.
Sign Service Agreement	Digitally signs a service agreement document. If the procedures are successful, the proper SCF's object reference is returned.
Terminate Service Agreement	Digitally signs a service termination document.
Are you Operational?	This question allows to ensure that the framework is workable.
Enable Heartbeating	This primitive is used to make the framework give a special signal (a "pulse") to the application. The pulses are repeated periodically until disabled.
Disable Heartbeating	Turns off the periodical heartbeating.
Check Load Statistics	This checks the load statistics of a given SCF, i.e. whether it is in a normal state, or it is overloaded (e.g. cannot cope with forthcoming calls).
Turn on/off Load Reporting	These two primitives are used to enable/disable periodic load reporting from the framework (or specified SCF).
What is the Time?	This primitive allows the framework and the application synchronize their clocks.
Turn on/off Event Notification	This primitive enables/disables the notification of SCFs' conditions (is it available, or not).

Tab. 4.7: Framework-to-application API gateway's primitives

Primitive	**Description**
Report Framework Fault	Informs the application about a serious fault in the framework. The application cannot use the framework until it recovers.
Report Framework Recovery	Tells the application that the previously reported fault is no longer valid. The application can use the framework again.
Are you Operational?	This question can be asked to the application by the framework to ensure that the peer is workable.
Set the *X* SCF Unavailable	This primitive turns the *X* SCF off. Now, the application can no longer use it .
Enable Heartbeating	This primitive is used to make the application give a special signal (a "pulse") to the framework. The pulses are repeated periodically until disabled.
Disable Heartbeating	Turns off the periodical heartbeating.
Check Load Statistics	This primitive allows framework to get informed whether the application is overloaded.
Turn on/off Load Reporting	These two primitives are used to enable/disable periodic load reporting from the application.
What is the Time?	This primitive allows the framework and the application synchronize their clocks.
Notify of an Event	This primitive is called by the framework when the SCF has changed its state, i.e. became (un)available.

Typical order of primitives:

1. List SCF Types (App)
2. Describe SCF Type (may be used for all possible types) (App)
3. Discover SCFs (App)
4. Select SCF (App)
5. Sign Service Agreement (App)

Description of interfaces:

- `IpServiceDiscovery` – discover which SCFs are registered into the framework. Acquire the first contact to them.
- `IpServiceAgreementManagement` and `IpAppServiceAgreementManagement` – these interfaces are used to mutually sign the proper service agreement between an application and the picked SCF provider. It is one step before acquiring an SCF's service manager interface.
- `IpEventNotification` and `IpAppEventNotification` – information about which SCFs are available or unavailable in the moment.
- `IpFaultManager` and `IpAppFaultManager` – these interfaces allow exchanging information about serious problems encountered in the framework and in the application. They are usable to ensure network integration.
- `IpHeartBeatManagement` and `IpAppHeartBeatManagement`, – these interfaces allow to create heartbeat session, e.g. a continuously repeated message: "I am still alive". The message is sent from the application to the framework and from the framework to the application.
- `IpHeartBeat` and `IpAppHeartBeat` – represent a heartbeat session. These interfaces define the `pulse()` method, which is called periodically by a peer.
- `IpLoadManager` and `IpAppLoadManager` – gives the opportunity to set load management policies and to control the congestion level (e.g. when the application cannot deal with new, incoming calls).
- `IpOAM` and `IpAppOAM` – interfaces used to synchronize time and date between framework and application.

4.2 Call Control SCF

Call Control (CC) SCF [26] is the hugest part of Parlay/OSA APIs. And the most significant so far. It defines call control interfaces, which allow creating, destroying and managing calls in any form (e.g. voice-carrying). The SCF is divided into four modules, starting from voice-only, two-party Generic Call Control Service (GCCS) to Multiparty Call Control Service (MPCCS) to Multimedia Call Control Service (MMCCS) and, finally, to the most complex Conference Call Control (CCCS).

The inheritance diagram for "call" and the "call control manager" (service manager) interfaces is presented in Fig. 4.4. Although in former Parlay API versions (Phase 1 and 2) the base

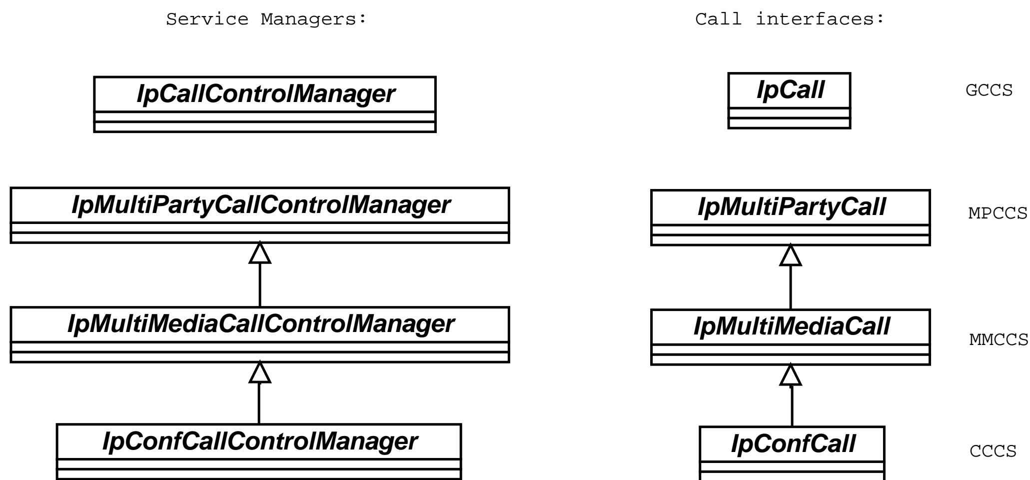

Fig. 4.4: Dependence among the call and the call service manager interfaces.

interfaces for all the CC APIs were GCCS interfaces, in Phase 3 interfaces the dependency was broken. Excluding GCCS, all CC "call" and "call control manager" base interfaces are MPCCS ones. However, all the more complex SCFs support the GCCS' call model. Consequently, on the functional layer, GCCS *is* the base for other modules. Because in this chapter functional features only are described, the GCCS is treated as if any other module inherited its functionality. Any single primitive which can be found in GCCS' description can be also applied to MPCCS, MMCCS and CCCS.

The sections that follow describe every call control module.

4.2.1 GCCS

Generic Call Control Service (GCCS) is a versatile set of interfaces, which could be applied to various networks. They support two party-calls only and give no direct control over parties in a call. The call model in Generic CCS is very simple. As shown in Fig. 4.5, it consists of two objects: of type `IpCall` (on the application's side) and of type `IpAppCall` (on the gateway's side). The parties (up to two) are not explicitly represented in this model.

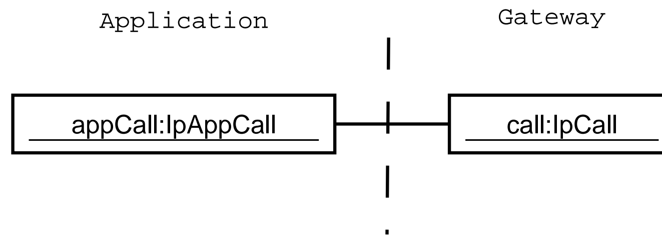

Fig. 4.5: The GCCS' call model

New calls may be created on the users' request as well as by the application (in such case the call is usually referred as third party call). Calls may be also destroyed by the application. The application can turn on monitoring of specified users and events in the network. Finally, the application may specify the charge plan (how and whom to charge for the service)[2]. Tables 4.8 and 4.9 present the list of possible operations.

[2]The Call Control SCF includes this very simple charging mechanisms, but the the Charging SCF (Section 4.10) defines much more complex and powerful API.

Tab. 4.8: Generic Call Control SCF's API primitives – called by application

Primitive	**Description**
Register Event	This primitive allows the application to specify the event that will be monitored by the network. An event may be for instance: "offhook event", „busy", "answer from called party", etc. When the event appears, the gateway notifies the application (by the "Notify of Event" primitive – Table 4.9).
Unregister Event	This primitive is used to unregister an event (which was previously registered with the "Register Event" primitive).
Connect Call with Specified Address	This primitive connects the call to the specified user. It can be used after being informed (from the gateway) about a call attempt in the network ("Notify of Event") or it can be a third party (application-initiated) call as well.
Deassign from Call	The application will be no longer assigned to the call. The call will be handled by the network in ordinary way.
Create Call	This primitive allows creating a call by the application (a third party call).
End call	This primitive ends the specified call. The entire call is ended, all parties are disconnected by the network.
Set Charge Plan	This primitive specifies the user who will be charged for the call (originating, destination or, possibly, another user). Additionally, the primitive sets a so-called *charge plan* for a call, i.e. the way the user is charged. Charge plans are operator-specific.
Set Load Control	This primitive is used to set the maximum number of calls that the gateway may pass to the application per certain time interval, e.g. allows 1 call per 10 milliseconds (100 calls secondly).

Tab. 4.9: Generic Call Control SCF's API primitives - called by gateway

Primitive	**Description**
Notify of Event	Once the event criteria (specified by the "Register Event" primitive) have been met, this primitive is passed by the gateway to the application.
Connect Response	This primitive is passed to the application after a specified progress with connecting the call (e.g. the destination user answers a call, ringing timeout is reached, etc.).

Typical order of primitives for user-generated calls:

1. Register Event "address collected" (App)
2. Notify of Event (Svc)
3. Connect Call with Specified Address (App)
4. Deassign from Call (App)
5. Unregister Event (App)

Typical order of primitives for third party calls:

1. Create Call (App)
2. Connect Call with Specified Address (user A) (App)
3. Connect Response (from user A) (Svc)
4. Connect Call with Specified Address (user B) (App)
5. Deassign from Call (App)

Description of interfaces:

Service Manager: `IpCallControlManager`

- `IpCallControlManager` and `IpAppCallControlManager` – interfaces used to create, supervise and destroy calls; allow enabling and disabling event notification.
- `IpCall` and `IpAppCall` – represent a call (at the gateway's and application's side, respectively); allow routing calls and setting charging polices.

4.2.2 MPCCS

Multiparty Call Control Service (MPCCS) is a set of interfaces that provide GCCS's functionality, and additionally, it defines the multi-point conference functionality. Since this functionality is not supported in some networks (like in fixed POTS networks), this set probably will not be available everywhere.

On the API level, MPCCS differs from the GCCS's interfaces and the inheritance relations between respective interfaces have been broken.

In this model, every single call party is represented by a so-called *call leg*. It is possible to add/subtract call legs to/from the call object during a call. Fig. 4.6 shows objects in a typical 2-party call based on MPCCS.

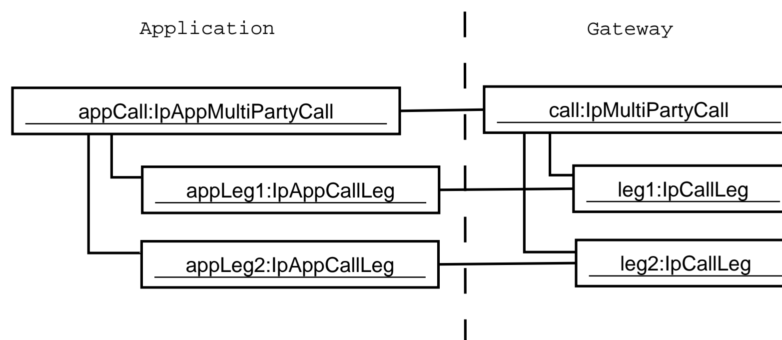

Fig. 4.6: The MPCCS' call model

The new features of the MPCCS SCF (comparing to GCCS) are presented in Table 4.10. Notice that MPCCS supports all the primitives presented in Section 4.2.1, in Tables 4.8 (for applications) and 4.9 (for gateway). They are not showed here to not double the information. Notice also that the "gateway table" is absent here since this module does not provide new functionality.

Tab. 4.10: New MPCCS SCF's API application's primitives over GCCS primitives

Primitive	**Description**
Create Call Leg	Creates and returns a new call leg (related to a specified call). At this moment the newly created call leg is an object at the gateway's side not related to any real user/address in the network.
Show Call Legs	This primitive is used to get all the call legs (which are identified by addresses) related to the call specified in a parameter.
Route Call Leg	This primitive links a user (specified by an address) to the call leg. Once linked (connected), the user may be able or not to send and receive any call-related media streams (it depends to primitive's parameters). If not – the application should call the "Attach Media to Call Leg" primitive (below).
Create and Route Call Leg	This primitive creates a new call leg and immediately starts routing procedures to the specified address. Note that in the result of this primitive two call legs are created – the first one directly, and the second indirectly if routing is successful.
Attach Media to Call Leg	After calling this primitive the call leg will be able to send and receive all the call-related media streams. In other words, in will be able to speak and hear. Note that before calling this primitive, the call leg is "dump".
Detach Media from Call Leg	Detaches the call leg from all call media streams. The call leg becomes "dump" again.
Release Call Leg	This primitive releases the specified call leg, i.e. closes all media streams and removes the user from the call.

Typical order of primitives:

1. Create Call (no users now) (App)
2. Create and Route Call Leg (two users in the call) (App)
3. Create Call Leg (App)
4. Route Call Leg (three users) (App)
5. Attach media (App)
6. Deassign from Call (App)

Description of interfaces:

Service Manager: `IpMultiPartyCallControlManager`

- `IpMultiPartyCallControlManager` and `IpAppMultiPartyCallControlManager` – interfaces used to create and supervise multi-party calls, and enabling/disabling call notification.
- `IpMultiPartyCall` and `IpAppMultiPartyCall` – represent the call on the SCF's and application's side, respectively. They Allow call routing, acquiring call details, call destruction, and charging. Direct call leg control is possible.
- `IpCallLeg` and `IpAppCallLeg` – these interfaces represent a party in the call. They may be created, destroyed and managed.

4.2.3 MMCCS

Multimedia Call Control Service (MMCCS) defines some mechanisms which may be used when multimedia calls are involved (like in case of H.323 terminals). In addition to multimedia versions of call and call leg interfaces, the special interface `IpMultiMediaStream` was defined, which represents, following the H.323-based naming convention, a pair of uni-directional channels of the same media type but opposite direction (for audio and video) or a bi-directional logical channel (for data). Fig. 4.7 shows a 2-party, 2-stream (audio+video) multimedia call.

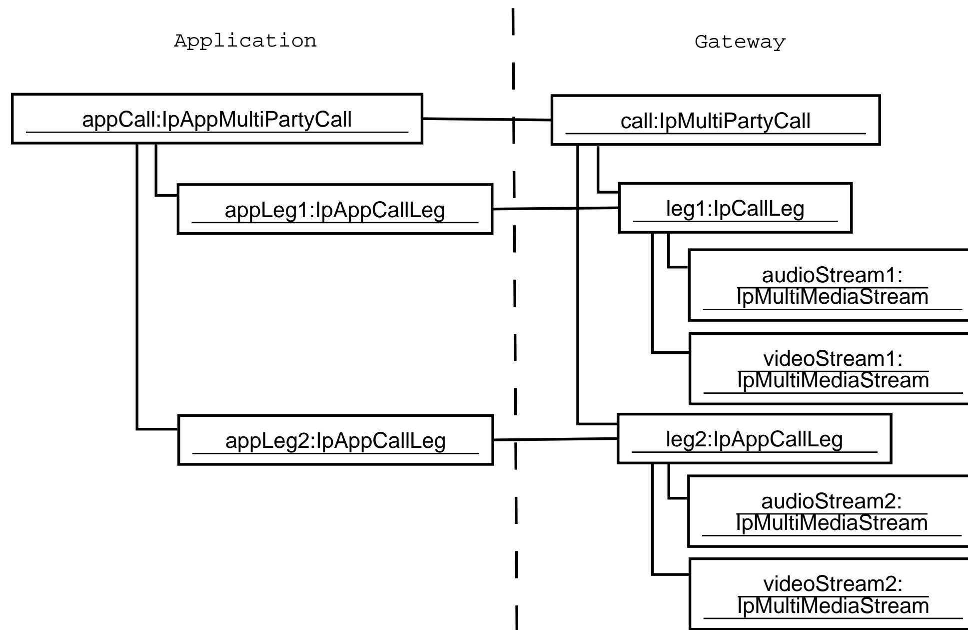

Fig. 4.7: The MMCCS' call model

In the figure, two call legs are controlled by the application. Physically, there are two streams in the call. Every call leg "owns" two media stream objects but they are separate objects, even if they physically point the same streams.

Tables 4.11 and 4.12 describe the new features comparing to MPCCS that appear in MMCCS. Notice that the MMCCS is inherited from MPCCS and it supports all the MPCCS's operations: Create Call (Leg), Route Call Leg, Release Call (Leg) and others.

Tab. 4.11: New MMCCS SCF's API application's primitives over MPCCS primitives

Primitive	**Description**
Enable Media Notification	This primitive enables multimedia stream detection. When specified conditions are met, i.e. a certain user opens a stream of a specified type (e.g. voice) and properties (e.g. the G.732.1 audio codec), the application will be notified.
Supervise Volume	This primitive is used to set amount of granted data for the call. If the amount is exceeded, the application will be notified and the call treated as the application decides (e.g. it will be released).
Attach Stream to Party	This primitive attaches a given multimedia stream to a party. From this moment, the party (represented by a call leg) will be able to send/receive data through the stream.
Enable Stream Notification	The primitive turns on/off stream event notification, i.e. whether the stream is added/subtracted to/from the call.
Subtract Stream from Call	This primitive, used by the application, removes the stream from the call.

Tab. 4.12: New MMCCS SCF's API gateway's primitives over MPCCS primitives

Primitive	**Description**
Notify of Media Event	This primitive is used by the gateway when certain criteria are met in network. The application will be informed about the event, all parties involved in the call and all attached media streams.
Notify of Stream Event	This primitive is used by the gateway to tell the application that the monitored stream-related event appeared.

Typical order of primitives:

1. Enable Media Notification (App)
2. Notify of Media Event (Svc)
3. Enable Stream Notification (App)
4. Notify of Stream Event (Svc)
5. Attach Stream to Party (App)
6. Subtract Stream from Call (App)
7. Stop supervising call (App)

Description of interfaces:

Service Manager: `IpMultiMediaCallControlManager`

- `IpMultiMediaCallControlManager` and `IpAppMultiMediaCallControlManager` – the pair of interfaces responsible for creating new multimedia calls and managing call event notification.
- `IpMultiMediaCall` and `IpAppMultiMediaCall` – these interfaces represent a multimedia call; have the same functionality as Multi-Party Call interfaces (part of MPCCS), but these are enhanced by volume supervision.
- `IpMultiMediaCallLeg` and `IpAppMultiMediaCallLeg` – represent a party (identified by an address) in a multimedia call; linked to zero or more multimedia streams.
- `IpMultiMediaStream` – represents a multimedia stream, i.e. audio/video/data bi-directional data stream.

4.2.4 CCCS

This module (Conference Call Control Service) provides the conference features. In this model, presented in Fig. 4.8, a conference is a certain abstraction which includes one (or more) so-called sub-conferences. The conference interfaces (of types `IpConfCall` at the gateway's and `IpAppConfCall` at the application's side) and sub-conference ones (`IpSubConfCall` and `IpAppSubConfCall`) objects are inherited from the multimedia call interfaces (`IpMultiMediaCall` and `IpAppMultiMediaCall`). In consequence, any conference is a call and any sub-conference is also a call. And, as every call, the conferences and sub-conferences are the collections of call legs (representing parties in the call).

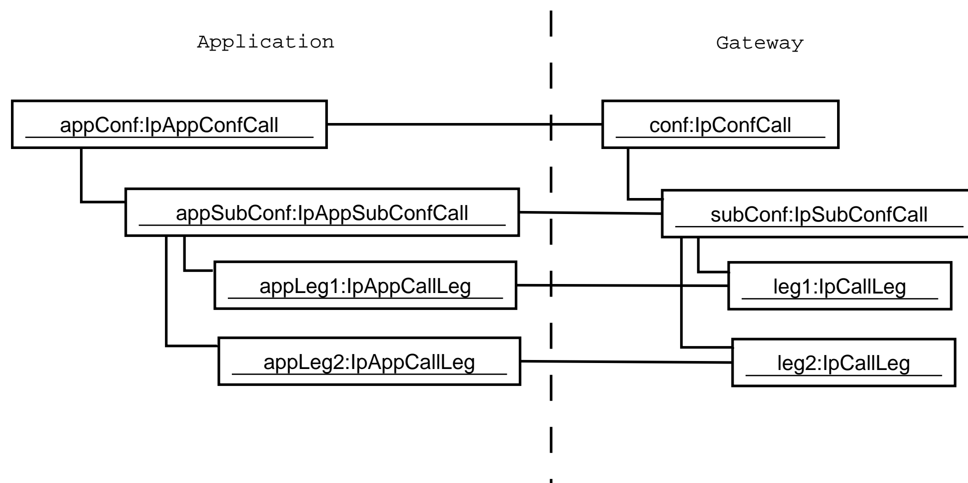

Fig. 4.8: The CCCS' call model

The difference between conference and sub-conference is that the former have the opportunity to create sub-conferences, but the latter (sub-conference) have quite a wide spectrum of more specific operations. The operations include: splitting and merging sub-conference, moving parties from one to another conference, choosing the chair and the likes. Every conference has also defined a conference policy, i.e. whether the conference can be joined, what kind of media is allowed and if the conference is chaired. All the CCCS's operations are presented in Tables 4.13 and 4.14.

Tab. 4.13: New CCCS SCF's API application's primitives over MMCCS primitives

Primitive	Description
Create New Conference	This primitive (called by the application) is used to create a new conference object. The number of participants and sub-conferences are specified, but the number may be excessed in future (although the resources are not guaranteed for those "extra" participants).
Reserve Resources	This primitive is used to reserve certain amount of bandwidth resources in the network for a specified number of conference participants at the specified time for a given period of time. At the requested time, if resources can be reserved, the gateway informs the application with the "Conference Created" primitive.
Free Resources	This primitive releases the previously reserved resources.
Create New Sub-conference	This primitive creates a new sub-conference.
Split Sub-conference	This allows splitting a sub-conference into two ones (the new sub-conference appears). The parties which have to be transferred to the new sub-conference are specified with this primitive.
Merge Sub-conference	This merges a sub-conference into other one. The merged sub-conference is released, all its call legs now start to belong to the target conference.
Move Call Leg	This primitive moves the specified call leg from one sub-conference into another.
Inspect Video	This specifies which video stream will be sent to the chair of the conference.
Appoint Speaker	This primitive is used to give a user the permission to talk (in multi-user conferences not all of participants may be allowed to speak simultaneously).
Select Chair	This selects the chair of the conference from the group of call legs.
Set Conference Policy	This primitive allows setting policy during a call (in an ad-hoc manner).

Tab. 4.14: New CCCS SCF's API gateway's primitives over MMCCS primitives

Primitive	**Description**
Conference Created	The gateway uses this primitive to tell the application that the (previously requested with the Reserve Resources primitive) conference has been created and the resources reserved.
Party Joined Conference	This primitive is passed from the gateway to the application when a party has joined the conference.
Party Left Conference	This primitive is passed from the gateway to the application when a party has left the conference.

Typical order of primitives:

1. Create New Conference (App)
2. Create New Sub-conference (App)
3. Party Joined Conference(Svc)
4. Party Joined Conference (Svc)
5. Split Sub-conference (App)
6. Release Call (in this case "Call" means "Conference") (App)

Description of interfaces:

Service Manager: `IpConfCallControlManager`

- `IpConfCallControlManage` and `IpAppConfCallControlManager` – the purpose of these interfaces is to create conferences and reserve some resources for a given period of time.
- `IpConfCall` and `IpAppConfCall` – represent a conference; since they are inherited from `IpMultiMedia(App)Call` they support all the multi-party and multi-media call features.
- `IpSubConfCall` and `IpAppSubConfCall` – represent a sub-conference; allow splitting and merging sub-conferences, moving call legs, chair selecting, giving a speaker the floor, and choosing the video stream for the chair; inherited from `IpMultiMedia(App)Call`.

4.3 User Interaction SCF

This SCF [27] is used to allow the application communicate interactively with users. The application can send textual messages to the user (like "Enter PIN", "Type password" and others); additionally, it may order the network to play specified voice announcement to the user. As for the application, it can collect messages typed by users on their terminals (PINs, SMSes) to further process them.

There are two sets of the User Interaction (UI) interfaces. The first set, represented by interfaces `IpUI` and `IpAppUI`, is a standalone set – the application may interact (e.g. send messages) with users separately from any call. The other set, represented by interfaces `IpUICall` and `IpAppUICall`, is related to an existing call (as described in section 4.2). The `UICall` interfaces are inherited from the `UI` interfaces. In addition, they define possibility to record voice messages (e.g. to replay them later). Both sets are shown in tables 4.15 and 4.16.

Tab. 4.15: User Interaction API application's primitives

Primitive	**Description**
Enable UI Notification	This primitive allows to enable notification about user's UI activity (which is usually a typing of a textual message, but the operator may widen the spectrum of possible activities). As a parameter, the application specifies the user's address.
Disable UI Notification	Turns off the notifications previously turned on by the "Enable UI Notification" primitive.
Send Information to User	The application uses this primitive to send information (e.g. textual) to the user or play an announcement on the user's terminal.
Collect Information from User	The primitive requests that the gateway should collect some data (e.g. textual) from a specified user. The application specifies the criterion of that type of data should be collected.
Record Message	(`UICall` only) Requests to record (save) all what is being told during the call.
Delete Message	(`UICall` only) Deletes the message (which had been previously recorded with the "Record Message" primitive).

Tab. 4.16: User Interaction API gateway's primitives

Primitive	**Description**
Notify of UI Event	This primitive is used by the gateway to signal to the application the activity of monitored users. The activity is usually a short textual message.

Typical order of primitives:

1. Enable UI Notification (App)
2. Notify of UI Event (Svc)
3. Send Information to User (App)
4. Collect Information from User (App)
5. Disable UI Notification (App)

Description of interfaces:

Service Manager: `IpUIManager`

- `IpUIManager` and `IpAppUIManager` – these two interfaces are used to create, destroy and control user interaction session. `IpUIManager` provides both the `IpUI` and `IpUICall` object factory.
- `IpUI` and `IpAppUI` – represent a user interaction session (on the gateway and on the application side, respectively).
- `IpUICall` and `IpAppUICall` – represent a user interaction session (this one is always linked to a call).

4.4 Mobility SCF

This SCF [28] has two main functions. First, it has the user location features. They may be used to localize the position of mobile, fixed or IP-telephony terminal. The other function of this SCF is called *User Status* (US) feature, i.e. user's reachability in the specified moment.

The location features have been divided into three modules:

1. User Location (UL) Feature – this module is used to acquire the geographical position of a given user. The returned values are latitude, longitude and uncertainty shape.

2. User Location Camel (ULC) Feature – the information that may be acquired with this module are rather network-based that geographical. The geographical coordinates can be also returned if requested (and supported).

3. User Location Emergency (ULE) Feature – this module is used to automatically localize the user in case of an emergency call (e.g. police, fire, ambulance services).

The features of the User Location module and the User Status module primitives are presented in Tables 4.17 and 4.18. Since UCL and ULE modules may be viewed as "special cases" of UL, they are not shown here.

Tab. 4.17: Mobility SCF's API application's primitives

Primitive	**Description**
Localize User	This primitive allows to localize a given user (specified by the address).
Enable Localization Reporting	This primitive is used to turn on periodic user localization. The message about actual user position is send to the application every specified period of time.
Disable Localization Reporting	This primitive turns off the periodic user localization (enabled with the "Enable Localization Reporting" primitive).
Enable Triggered Localization	This primitive allows the application to set certain triggers on a user. When the user meets given conditions (i.e. enters or leaves the specified area) the application will be informed about the fact.
Disable Triggered Localization	This primitive turns off the triggered user localization (enabled with the "Enable Triggered Localization" primitive).
Determine User Status	This operation determines the user's state. The state can be `reachable`, `unreachable` or `busy`.
Enable Triggered State Reporting	This primitive is used to turn on user state monitoring. The application will be informed only if the user's state changes (e.g. he/she just becomes reachable).
Disable Triggered State Reporting	This primitive turns off the specified state reporting (enabled with the "Enable Triggered State Reporting" primitive).

Tab. 4.18: Mobility SCF's API gateway's primitives

Primitive	**Description**
Report Localization	This primitive is a localization report sent to the application from the gateway periodically after the "Enable Localization Reporting" operation has been called.
User Enters/Leaves Area	This primitive is send from the gateway to the application when a monitored user enters/leaves the specified area. Such primitives are set only if there was a former "Enable Triggered Localization" call.
User Status Changed	This primitive is send from the gateway to the application if the monitored user's status has changed. This notifying takes place only if it was formerly enabled by the "Enable Triggered State Reporting" primitive.

Typical order of primitives:

1. Enable Localization Reporting (App)
2. User Enters Area (Svc)
3. Disable Localization Reporting (App)

4.4.1 User Location Interfaces

Service Manager: `IpUserLocation`

- `IpUserLocation` and `IpAppUserLocation` – interfaces that allow enabling/disabling user location features (described in Tables 4.17 and 4.18).
- `IpTriggeredUserLocation` and `IpAppTriggeredUserLocation` – these two interfaces are inherited from `IpUserLocation` and `IpAppUserLocation`, respectively. The "triggered" interfaces enhance the functionality by adding trigger functionality (the application is informed when the user enters of leaves certain region/area).

4.4.2 User Location Camel Interfaces

Service Manager: `IpUserLocationCamel`

- `IpUserLocationCamel` and `IpAppUserLocationCamel` – these interfaces allow using the "Camel" of location features (network localization instead of the geographic one). These two include the trigger functionality.

4.4.3 User Location Emergency Interfaces

Service Manager: `IpUserLocationEmergency`

- `IpUserLocationEmergency` and `IpAppUserLocationEmergency` – the "Emergency" versions of the above interfaces.

4.4.4 User Status Interfaces

Service Manager: `IpUserStatus`

- `IpUserStatus` and `IpAppUserStatus` – these two interfaces contain all the User Status (US) functionality. They include trigger functionality.

4.5 Terminal Capabilities SCF

This SCF [29], the smallest one, is used to acquire some information about user terminal which is specified by address. The information returned is not fully defined by Parlay/OSA APIs. It is expected to include URLs, terminal attributes and other values[3]. Table 4.19 shows supported operations.

[3] The reader may be surprised that some APIs' elements may be not specified in those strictly-defined APIs. However, to improve flexibility, many data elements in Parlay/OSA APIs are defined as "strings", and those string values are not specified by the standard. Thus the API is both coherent and flexible.

Tab. 4.19: Terminal Capabilities SCF's API application's primitives

Primitive	**Description**
Get Terminal Capabilities	This primitive returns a data set about a given user terminal (e.g. mobile phone)

Description of interfaces:

Service Manager: `IpTerminalCapabilities`

- `IpTerminalCapabilities` interface is the only interface that has one method, which is called `getTerminalCapabilities()`.

4.6 Data Session Control SCF

This SCF [30] is used to manage data sessions (DS), like the GPRS ones. It has a simple call model, which is in some aspects similar to the GCCS model. The main difference is that here all calls are initiated by terminals, i.e. there is no possibility to create third party calls. Tables 4.20 and 4.21 present the primitives for this SCF.

Tab. 4.20: Data Session Control SCF's API application's primitives

Primitive	**Description**
Enable DS Notification	This primitive turns on the specified data session events monitoring ("DS setup", "DS established", "DS QoS Changed").
Disable DS Notification	This primitive turns off event monitoring enabled with the "Enable DS Notification".
Connect Call with Specified Address	This primitive routes the call to the specified user. It is used by the application after having acquired an event from the gateway.
Release Session	This primitive releases an existing data session and all the related resources (at the gateway side and in the network).
Set Charging Plan for Session	This primitive requests how the data session must be charged (currency, amount of money, time interval per a "tick").
Supervise Session	This primitive enables session supervision, i.e. allows the application setting how much bytes may be sent during a single session and how to treat the session if the limits have been reached (e.g. to release it).

Tab. 4.21: Data Session Control SCF's API gateway's primitives

Primitive	**Description**
Report the *X* Event	This primitive tells the application that the monitored event has appeared. The event information (addresses, event name, QoS class) is passed with this primitive.
Supervision Data Session Report	This primitive is sent by the gateway to tell the application that the volume limits have been reached; the gateway behaves in the way specified in the "Supervise Session" primitive parameters.

Typical order of primitives:

1. Enable DS Notification (App)
2. Report Event (Svc)
3. Set Charging Plan for Session (App)
4. Connect Call with Specified Address (App)
5. Supervision Data Session Report (Svc)
6. Release Session (App)

Description of interfaces:

Service Manager: `IpAppDataSessionControlManager`

- `IpAppDataSessionControlManager` and `IpDataSessionControlManager` – the pair of interfaces used to create sessions and manage event notifications.
- `IpAppDataSession` and `IpDataSession` – represent a session (which is the equivalent of a call in Call Control SCF); allow connecting, setting charging plans, and session supervision.

4.7 Generic Messaging SCF

This SCF [31] delivers mechanisms which are used to manage (send, acquire) messages. In this model (presented in the Fig. 4.9), a message resembles an e-mail: it is described by a sender, an addressee, a specified format, and other properties. Messages are collected in folders and folders are attached to mailboxes. A single messaging manager (the service manager for this SCF) may open several mailboxes at the same time.

Notice that the application have no access to the messages' bodies (text, attachments) nor cannot generate the content. All it can do is getting and sending messages, and performing of operations on messages' properties (addresses, dates, carbon copies, ...).

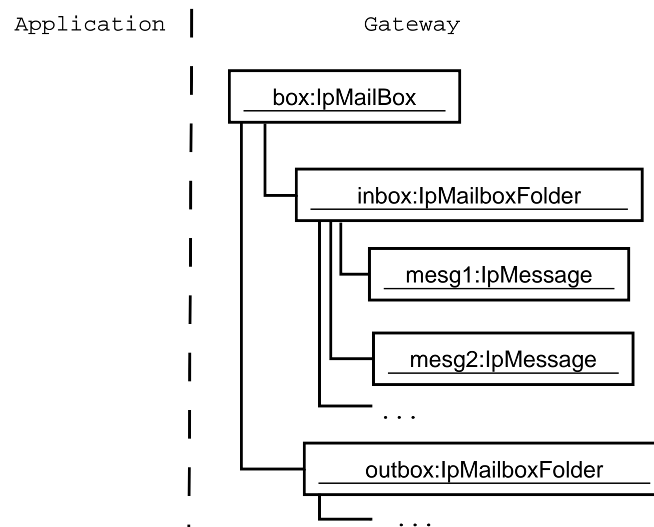

Fig. 4.9: The Generic Messaging call model

As shown in the Fig. 4.9, all mailboxes, folders and messages are located at the gateway's side – there are no `App` versions of those interfaces. The figure presents a mailbox with two two folders. The mailbox, like every mailbox in this model has an owner. The folders are called `inbox` and `outbox`, and these two are always present in every Generic Messaging (GM) SCF instance. Putting a message to the `outbox` folder means sending it. Naturally, the GM SCF supports the event notification features (that a new message has arrived), but this is achieved by means of the manager and app-manager interfaces (not shown in the figure). All the SCF's primitives are presented in Tables 4.22 and 4.23.

Tab. 4.22: Generic Messaging SCF's API application's primitives

Primitive	**Description**
Enable Message Notification	This primitive is used to enable notification of arriving messages.
Disable Message Notification	This primitive disables the specified message notification (enabled with the "Enable Message Notification" primitive).
Lock Mailbox	This primitive locks the mailbox. If locked, no other applications/threads can access the mailbox.
Unlock Mailbox	This primitive unlocks the mailbox (formerly locked with the "Lock Mailbox" primitive).
Modify Mailbox Properties	This primitive is used to modify mailbox properties, i.e. mailbox identifier, mailbox owner, folders and dates.
Remove Mailbox	This primitive removes the mailbox and all included folders/messages.
Create Folder	This primitive creates a new folder.
Open Folder	This primitive opens a (previously created) folder; once opened, it can be accessed.
Get Message	This primitive is used to get messages from a specified folder; once a message is "got" it can be accessed.
Put Message	This primitive is used to put the message to a folder; if the message is "put" to the `outbox` folder, it will be sent to the addressee.
Modify Folder Properties	This primitive is used to modify folder properties.
Close Folder	This primitive closes the folder; no further access until it is opened again.
Remove Folder	This primitive removes the folder and all messages inside.
Get Message Properties	This primitive is used to acquire message's properties, i.e. sender, subject, dates (when sent, when received, ...), size, format, and so on.
Modify Message Properties	This primitive allows to modify message's properties (compare "Get Message Properties", above).
Remove Message	This primitive removes the message from a specified folder.

Tab. 4.23: Generic Messaging SCF's API gateway's primitives

Primitive	**Description**
Message Arrived	This primitive is used by the gateway to notify the application that a new message has just arrived.

Typical order of primitives:

1. Enable Message Notification (App)
2. Message Arrived (Svc)
3. Lock Mailbox (App)
4. Open Folder `inbox` (App)
5. Get Message (App)
6. Get Message Properties (App)
7. Modify Message Properties (App)
8. Put Message into `outbox` (App)
9. Close Folder `inbox` (App)
10. Unlock Mailbox (App)

Description of interfaces:

Service Manager: `IpMessagingManager`

- `IpMessagingManager` and `IpAppMessagingManager` – the pair of interfaces that allow message arrival management, opening mailboxes and detecting faults in mailboxes.
- `IpMailbox` – represents a mailbox; gives access to folders.
- `IpMailboxFolder` – represents a folder; allows putting and getting (sending and acquiring) messages; gives access to them.
- `IpMessage` – represents a message; its properties may be modified, and the message may be taken and put from/to various folders.

4.8 Connectivity Manager SCF

This SCF [32] is used by the enterprise operator (note: not the application). Its purpose is to manage virtual private networks (VPNs), i.e. which sites they include and how they are connected by means of virtual provisioned pipes (VPrPs). VPrP is a link between 2 sites, a time slot for instance. A collection of VPrPs is called a virtual provisioned network (VPrN). Sites and networks cannot be created or modified using this API. They must be defined using off-line methods which are not defined Parlay/OSA.

Virtual provisioned pipes are created inside virtual provisioned networks at the request of the enterprise operator. The enterprise operator uses so-called quality of service (QoS) templates to create them. In short, every VPrP is associated with a QoS template. A template is a kind of a pattern defining every detail about QoS. First, it has a type (e.g. "Gold", "Silver" or "Bronze"). It has also a set of properties, e.g. call policy, bandwidth, delay, jitter, packet loss and others. Templates cannot be created by the enterprise operator. Many properties may be specified only by the SCF provider and thus they are read-only for the enterprise operator. Other properties can be changed. It is the SCF provider who sets which properties are read-only and which are not for the enterprise operator (e.g. the provider may allow enterprise operator setting bandwidth, but not delay). The primitives supported by this SCF are shown in Table 4.24. They are all generated by the enterprise operator.

Tab. 4.24: Connectivity Manager SCF's API primitives

Primitive	**Description**
Get Site List	This primitive returns the full list of sites which form the VPN (or enterprise network, as it is called here)
Get Site Details	The primitive is used to acquire some specific information about a given site: the list of service access points (SAPs), site location, description, and IP subnet details (in case of an IP network).
Get Template List	This primitive returns the list of possible QoS templates.
Get Template Details	This primitive acquires information about the template: its type, description, associated SLA, QoS details and validity (when the template can be used, e.g. only in weekends).
Get VPrP List	This primitive acquires the list of all virtual provisioned pipes of the virtual provisioned network.
Create VPrP	This primitive creates a new virtual provisioned pipe.
Delete VPrP	This primitive deletes an existing virtual provisioned pipe.
Get VPrP Details	This primitive acquires VPrP details, i.e. SLA, QoS details, validity and status (whether it is active, pending or disallowed).

Typical order of primitives:

1. Get Template List (EntOp)
2. Get Template Details (EntOp)
3. Create VPrP (EntOp)
4. Get VPrP Details (EntOp)
5. Delete VPrP (EntOp)

Description of interfaces:

Service Manager: `IpConnectivityManager`

- `IpConnectivityManager` – the service manager; gives access to QoS menus and enterprise network objects.
- `IpEnterpriseNetwork` – this represents the enterprise network; gives access to network sites and the related VPrN.
- `IpEnterpriseNetworkSite` – this represents any network site in the enterprise network.
- `IpQoSMenu` – this interfaces gives access to QoS templates.
- `IpQoSTemplate` – this represents a QoS template.
- `IpVPrN` – represents a virtual provisioned network (a collection of VPrPs).
- `IpVPrP` – represents a virtual provisioned pipe.

4.9 Account Management SCF

This SCF [33] is used to retrieve information related to user's accounts maintained by the operator and the money they are being (and have been) charged by the operator. Note, that the application cannot change the state of the account; only inquire about it. Tables 4.25 and 4.26 show the most important primitives.

Tab. 4.25: Account Management SCF's API application's primitives

Primitive	**Description**
Enable Account Notification	This primitive enables the notification about any charging activity for a specified users. The activity can be specified in a parameter and it may be: “charging event”, “recharging”, “account is low”, “account is zero” and “account is disabled”.
Disable Account Notification	This primitive disables the notification (enabled with the “Enable Account Notification” primitive).
Query Balance	This is used to acquire the specific information about a given account including detailed information about amount of money charged.
Get Transaction History	This retrieves the history of transactions for a given user within a given time period.

Tab. 4.26: Account Management SCF's API gateway's primitives

Primitive	**Description**
Report the *X* Event	This primitive (called by the gateway on the application) tells that the monitored event has appeared.

Typical order of primitives:

1. Enable Account Notification for "charging event" (App)
2. Report the charging event for User A (Svc)
3. Report the charging event for User B (Svc)
4. Disable Account Notification (App)
5. Get Transaction History for User C (App)

Description of interfaces:

Service Manager: `IpAccountManager`

- `IpAccountManager` and `IpAppAccountManager` – a pair of interfaces supporting the functionality described above.

4.10 Charging SCF

This SCF [34] is was designed to charge users, i.e. to measure and reserve money [4] on user's account (maintained by the operator). This SCF many be used, for example, to build a billing application. First, some amount of money can be reserved by the applications in the framework. Next, during a call, the amount may be added and subtracted from the reserved amount, depending on the price for the application's usage. Table 4.27 shows the most important features of this SCF's API. All presented primitives are application's primitives.

[4]this SCF may be also used to operate on specified units (e-mails, bytes of data, etc.), not only money. However, to make the description clear, we do not discuss this case here.

Tab. 4.27: Charging SCF's API application's primitives

Primitive	**Description**
Reserve *X* Amount of Money	This primitive is used to reserve a given amount of money.
Release Reservation	This primitive is used to release a reservation made with the "Reserve Amount of Money" primitive.
Debit Amount of Money	This primitive subtracts some amount of money from the reserved amount. The debit depends on the price of the service usage.
Credit Amount of Money	Add some amount of money to the previously reserved amount. This may happen, for instance, in case of bonus rechargements.
Get Amount of Money Left	The primitive returns the amount of money that has left from the reserved amount.
Rate the *A* User	This primitive is used to present the pricing information to the specified end user A.

Typical order of primitives:

1. Reserve 10 EUR (App)
2. Debit 1 EUR (App)
3. Get Amount of Money Left (App)
4. Debit 1.5 EUR (App)
5. Release Reservation (App)

Description of interfaces:

Service Manager: `IpChargingManager`

- `IpChargingManager` and `IpAppChargingManager` – this pair is used to create charging sessions (every single session is linked to a specified user).
- `IpChargingSession` and `IpAppChargingSession` – these interfaces represent a charging session. All the operations (described in the Table 4.27) may be undertaken on the `IpChargingSession` interface, while the latter one is used to acquire responses.

Chapter 5

Service Design in Parlay/OSA

5.1 Introduction

This chapter describes how to develop an application using Parlay/OSA. It describes in details which objects must be implemented, and how they should behave (in form of UML diagrams). Some author's solutions are presented here (like inter-object communication, synchronization) and source code fragments in Java are provided. The full application's sources are also available (Appendix C).

During the research, the author found several references to Parlay/OSA test applications. The very first ones were developed by big companies, especially the Parlay Group members, like Lucent [35] or Ericsson [4]. Other implementations were academic projects: G. Weitoft and P. von Dolwitz in their MSc thesis [36] described a Parlay non-middleware application and "gateway". They (manually) generated the Java APIs from IDL sources and integrated this with the INAP protocol. P. Ebben in his MSc thesis [37] discusses the integrity aspects of using Parlay/OSA and tests the implementation with the SPIN program. One of the earliest implementation is the one presented as a part of Eurescom P909 project [7]. This implementation is based on Parlay v1.2.

The author hopes that by now the application described here and the examples would be useful for someone who tries to create an application in Parlay/OSA, and especially for all those who want to get familiar with the potentials of the APIs.

5.2 Testing platform

Very fortunately, during the research on this work, Ericsson published its free platform called "Ericsson OSA/Parlay Simulator". It saved the author much effort with designing a Parlay/OSA gateway.

The simulator, written entirely in Java, is built of a network part and a Parlay/OSA gateway part. The network part allows to built a simple network consisting of a set of terminals (mobile phones) and to assign addresses to them. Once the network is built, calls may be started from terminals (by typing telephone numbers). They are routed to specified addresses.

The Parlay/OSA gateway part is strictly linked to the network part. Naturally, the network can work without any Parlay/OSA application, but the whole default behaviour may be changed and altered by applications.

Here, apart from the framework module, the gateway is built of two SCFs: Call Control and User Interaction. They both are not fully implemented (not all methods are workable). Besides, this implementation is based on the 3GPP version (release 4) of the APIs [38] (based on Parlay phase 2, compare Section 1.4), which is much smaller than original Parlay APIs. Especially, is does not support multimedia and conference modules. However, the spectrum of supported features in the Ericsson gateway is broad enough to allow creating powerful applications.

Two applications are provided with the simulator as examples: "Call Barring", and "Web Dial". They are very useful examples of how to use the Parlay/OSA APIs. What is more, every application includes some common classes provided by Ericsson (packages, placed in `com.ericsson` are named: `datastructures`, `configuration`, `parlay`, `tracebug`), which have some framework mechanisms implemented (e.g. authorization, getting access). They all are put to Java archive file `classes.jar` and are used by the application presented below. This helped to separate the author's code from the Ericsson's. Additionally, the Ericsson's applications and the gateway use a special library, `security.jar`, which implements security procedures, i.e. encrypting, digital signing and supporting classes. All 3GPP OSA interfaces were provided in the `parlay3gpp.jar` file, which were generated from appropriate IDL sources, compiled with Java and archived with Java archiver (jar).

5.3 Application

The application presented here is simply a number translation application. In other words, the application is informed any time when a specified user types a specified number. Then, the application's logic changes the destination number, and, in consequence, the calling user is routed to the new (translated) number.

There can exist many variations of number translation. First, some numbers may be translated for *all* subscribers in the same manner in a given area (that is how IN's 0800 works). Next, some users may be treated independently and their destination numbers translated differently.

In the case being described here, only one user is monitored. The triggered telephone number is "1". If the user types a number of equal or greater value than 10, it is rerouted to number value modulo 10. Otherwise, it will be routed the number requested. This example will show how simple would be writing fairly exotic applications.

To achieve this goal, the application uses the GCCS module of the Call Control SCF. The pseudocode for the application is shown in Fig. 5.1. The pseudocode uses primitives, which are described in Tables 4.8 and 4.9 (Section 4.2.1). The "Create New Destination Address" procedure returns a new destination address.

Note that the application is working in its own thread. The gateway has also its (separate) thread. If any of them call a method on the peer's object, it is done in the thread of method's callee.

When using the GCCS module, the two following interfaces have to be implemented by the application: `IpAppCallControlManager` and `IpAppCall`. Their purpose is to work as callback

```
1 Register Event ("Address Analyzed Event");
2 do {
3   wait for "Notify of Event" from gateway;
4   newAddress = Create New Destination Address (dialedAddress);
5   Connect User with Specified Address (newAddress);
6   Deassign from Call;
7 } while (true);
```

Fig. 5.1: Application logic – pseudocode

objects: the former is an event collector and the latter – application's representation of a call. Since those two interfaces cannot be instantiated theirselves (interfaces are always abstract), the following classes implement them: the `AppCallControlManager` class (which implements `IpAppCallControlManager`) and `AppCall` (which implements `IpAppCall`). They are placed in files `AppCallControlManager.java` and `AppCall.java`, respectively.

Additionally, some specific classes have been developed as parts of the application. They all are prefixed with `MyApp` to show that they are implementation-specific (not defined by Parlay/OSA APIs):

- `MyAppInit` – the object of this class is responsible for initializing CORBA system, acquiring the `IpInitial` object from the CORBA name service, authenticating and getting access to the framework and discovering available SCFs (all this by using Ericsson-provided procedures). Then, it acquires the GCCS' call control service manager object (`IpCallControlManager`). Finally, it creates the application logic object (`MyAppLogic`, below) passing to it the service manager's reference. File: `MyAppInit.java`.
- `MyAppLogic` is the "main logic" of the application. First, it enables call notification by calling appropriate method on the `IpCallControlManager`-class object. Next, it starts waiting. When an event appears, it is processed here: the destination address is computed and it is passed to the gateway. After that, the application deassigns from the call, and starts waiting for next events. File: `MyAppLogic.java`
- `MyAppEvent` – this class represents an event (and all related data) that appears at the gateway and is passed to the application. File: `MyAppEvent.java`
- `MyAppEventQueue` – this is a queue of events. This FIFO queue is used by the callback object (`AppCallControlManager`) to put all the events it gets from the gateway. Then the events (`MyAppEvent`) are taken by the application logic (`MyAppLogic`).

  The queue implements two methods: `get()` (used by the application logic) and `put()` (used by the callback objects). The methods are synchronized (i.e. thread-safe). If there are no events while calling `get()`, then the thread is blocked until a new event appears. File: `MyAppEventQueue.java`

Furthermore, the following jar files (described in Section 5.2) are included to the program: `parlay3gpp.jar`, `security.jar`, `classes.jar`.

Since the framework procedures were not developed by the author, they will be not described here at all. Usually those procedures follow the similar pattern but they may include very specific procedures (e.g. authorization methods). However, the initial procedures' development should not influence the application logic's development. In the application presented here, the initial procedures (`MyAppInit`) are separated from the application logic (`MyAppLogic`).

The description of the application starts in the moment when it already possesses an object reference to the call control manager object. What happens next is presented in Fig. 5.2.

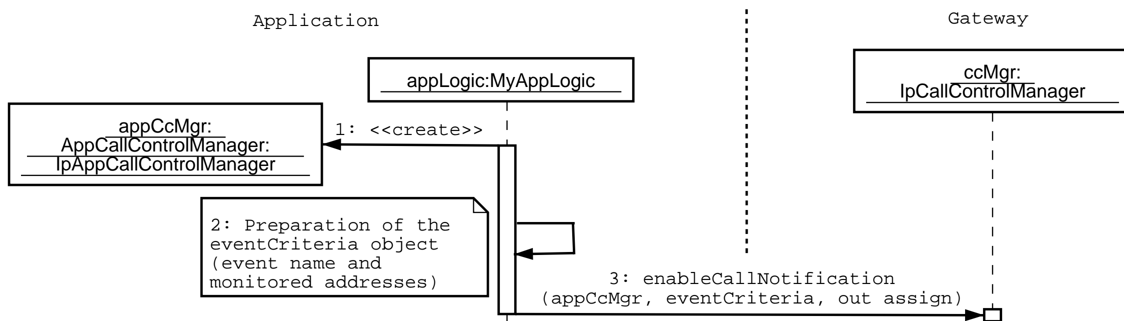

Fig. 5.2: Enabling call notification

First, a new `AppCallControlManager` object (derived from the `IpAppCallControlManager` interface) is created at the application's side.

Next, the event details and the appropriate user's address must be prepared. In Parlay/OSA there is a special structure, which contains all data needed. The `TpCallEventCriteria` structure is defined in IDL as shown in Fig. 5.3. The two first fields are addresses ranges: destination

```
1 struct TpCallEventCriteria {
2     TpAddressRange DestinationAddress;
3     TpAddressRange OriginatingAddress;
4     TpCallEventName CallEventName;
5     TpCallNotificationType CallNotificationType;
6     TpCallMonitorMode MonitorMode;
7 };
```

Fig. 5.3: The IDL definition of the TpCallEventCriteria structure

address and originating address. Addresses in Parlay/OSA may have various forms: telephone E.164 numbers, IPv4 and IPv6 addresses (multicast as well as unicast), URL, SMTP, X.400, and a few others. That is why the special wrapper class, `TpAddress`, was introduced.

The `TpAddressRange` class, which is used as the address type in Fig. 5.3, has broader meaning: the address string can cover many possible addresses due to using wildcards (in `TpAddress` they are forbidden). Parlay/OSA defines two wildcards: "*" (meaning zero or more characters) and "?", meaning only one character. The examples of E.164 addresses are: "12345*" – all numbers starting with "12345"; "12345?" – all six-digit numbers starting with "12345"; "*" – any address.

In the program presented here, the originating address is “1” but the destination can be any address. Then “*” is set in the `DestinationAddress` field (filling of the structure is shown in Fig. 5.4).

The next field, `TpCallEventName`, specifies the name of event. The possible events are:

- **Offhook event** the user has just started calling procedures, the line is hung, but no numbers have been collected yet.
- **Address collected event** the user has typed a number but it has not been checked yet (at the network’s side). The address may be incorrect or not full.
- **Address analyzed event** in this moment the address has been checked – it has correct form.
- **Called party busy** the destination user’s terminal is busy
- **Called party unreachable** the destination user is unreachable
- **No answer from called party** the destination user has not answered the call during a specified time period.
- **Route select failure** the call could not be routed to the destination address (e.g. due to network overload).
- **Answer from called party** the destination user has answered the call

The event type that is set here is **Address analyzed event**. This means that the application will be informed once the originating user has typed a correct telephone number.

The next field (line 5 in Fig. 5.3), of the type `TpCallNotificationType`, tells what should happen to the call in the network once the event occurs. The usual manner is to interrupt the call until the application has done its job. But it is also possible continue processing of the call in the network (in such case the application would not have the possibility of routing the call). Here, all the calls will be interrupted.

The last field of the Fig. 5.3’s structure is `MonitorMode`. It specifies the party the monitored event refers to. It could be either an originating or destination party. Here, we are interested in the originating party event (**Address analyzed event**). The source code of the procedure filling the `TpCallEventCriteria`’s fields as requested is shown in Fig. 5.4.

The procedure, called `createOrigEventCriteria()` (“Orig” – because it always sets the criteria for originating user), takes three parameters: two addresses (originating and destination) and the event name (`P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value`). While the event name is defined as an integer number (`int`) in Java, it could be also set in more user-friendly manner by specifying static member event of a corresponding class. This method is called here as follows:

```
createOrigEventCriteria(new String("1"),
        P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value);
```

```
1  TpCallEventCriteria createOrigEventCriteria(
2      String originating,
3      int event_name) {
4          TpCallEventCriteria ec = new TpCallEventCriteria();
5          ec.DestinationAddress = nonOsaCreateE164Address("*");
6          ec.OriginatingAddress = nonOsaCreateE164Address(originating);
7          ec.CallEventName = event_name;
8          ec.CallNotificationType = TpCallNotificationType.P_ORIGINATING;
9          ec.MonitorMode = TpCallMonitorMode.P_CALL_MONITOR_MODE_INTERRUPT;
10         return ec;
11 }
```

Fig. 5.4: Java source code for the MyAppLogic.createOrigEventCriteria() method

The program uses a special procedure, which creates an E.164 address based on given string. The procedure is called `nonOsaCreateE164Address()`. This procedure and other parts of the application's source code are presented in Appendix C.

Once the `AppCallControlManager` and `TpCallEventCriteria` objects are created, the event has to be registered. The method, which is the API equivalent for the "Register Event" primitive, is named `enableCallNotification()`. The respective source code is shown in Fig. 5.5. The Parlay/OSA'a method call is here included in the `monitorNumbers()` `MyAppLogic`'s method.

```
1  int monitorNumbers(IpCallControlManager mgr,
2          IpAppCallControlManager appMgr,
3          String originat_address) {
4
5      TpCallEventCriteria ec = createOrigEventCriteria(originat_address,
6          new String("*"),
7          P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value);
8      IntHolder assignment = new IntHolder();
9
10     try {
11         mgr.enableCallNotification(appMgr, ec, assignment);
12     } catch (Exception e) {
13         /* Exception handling */
14     }
15     return assignment.value;
16 }
```

Fig. 5.5: Java source code for MyAppLogic.monitorNumbers() – registering events

As seen in Fig. 5.5, the `enableCallNotification()` method takes three parameters. The first two are our newly created objects. The last parameter is a returned value. The value, called *assignment ID*, is a handle assigned to each `enableCallNotification()` call. This handle will be further used as a parameter in any "Notify of Event" operation; it can be also used to disable this call notification. Note, that the assignment ID value is not a plain Java's `int` but an

IntHolder object. Holder objects are created by IDL compilers as implementation of out-type parameters because there is no direct way to pass a value from inside a method to the method's caller.

After having called the enableCallNotification() method, the application starts waiting for events. The event appearance and the application's behaviour are illustrated in Fig. 5.6.

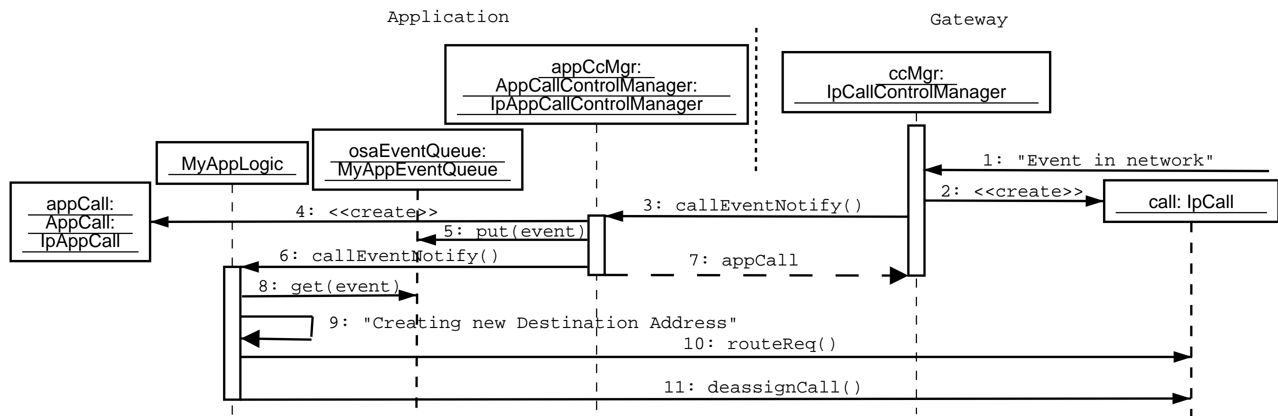

Fig. 5.6: Event notification diagram

In the first step, the network informs the gateway that a monitored event has appeared. Since the network–gateway protocol is not covered by Parlay/OSA APIs, this may have various forms. Here, an operation "Event in Network" is called on the ccMgr object (of type IpCallControlManager). Then, the ccMgr creates a new IpCall object (step 2). The Parlay/OSA method, callEventNotify(), is called in step 3 (its body is presented in Fig. 5.7). The method has four parameters. The first parameter is a structure called TpCallIdentifier, which contains a reference to the IpCall object and the call ID (integer). The second parameter is the event's description: event name and addresses: originating and the destination addresses[1]. The third parameter is assignment ID, which must be of the same value as the assignment ID returned from respective enableCallNotification() call. The fourth and last parameter is out-type. The application uses this one to pass an IpAppCall object reference to the gateway. But before the callEventNotify() method returns, it has to create a new AppCall object (step 4) and put the information about the event to the queue (object osaEventQueue of type MyAppEventQueue) in step 5. Next, the MyAppLogic object is informed about the event with the callEventNotify() method call (notice, that this is the same name as the IpAppCall's, but since the MyAppLogic class is non-standard, the MyAppLogic's name might be different). Once awaken, the MyAppLogic' simply gets the event object from the queue. It should be underlined, that until the IpAppCall's callEventNotify() method returns (steps 3-7), it blocks the gateway's thread. Therefore it should last as shortly as possible. No application's logic elements should be present there. Moreover, it is forbidden to call Parlay/OSA methods from callback object's methods (e.g. routeReq()). That is why all relevant information is passed to the MyAppLogic and then the method finishes, returning the expected object reference (step 7).

[1] Speaking truthfully, it also contains two more addresses – original destination address and redirecting address (both may be useful if the address has been rerouted more than once)

The `AppCall.callEventNotify()` is presented in Fig. 5.7. The `logic` object, shown in the figure, if a `MyAppLogic` object which must be passed to each created object to allow them to communicate with the application's logic.

```
1 public void callEventNotify(
2         TpCallIdentifier callReference,
3         TpCallEventInfo eventInfo,
4         int assignmentID,
5         IpAppCallHolder appCallRef) {
6     appCallRef.value = new AppCall(logic);
7     logic.callEventNotify(callReference,
8             eventInfo, assignmentID); /* push it further */
9 }
```

Fig. 5.7: Java source code for the AppCall.callEventNotify() method.

The application has been asleep since then. Now, it is being awaken. Initially, the application tries to determine the new destination address, which is the application's purpose (step 9 in Fig. 5.6). Next, once the new address is ready, the application calls two methods on the gateway's `call` object. In step 10, the `routeReq()` method is called, which requests routing the call to the specified destination address. In step 11, the application calls the `deassignCall()` method, which deassigns the application from the call. Obviously, the call is not released now. By calling the method the application tells the gateway that it is no longer interested in the call. The application has done what it was expected to do.

In this implementation the application's logic works in a loop. The whole loop is presented in Fig. 5.8.

```
1  do {
2      MyAppEvent anEvent = osaEventQueue.get();
3
4      if (anEvent.eventInfo.CallEventName ==
5                      P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value) {
6          String addrString  = translateModulo10(anEvent.
7              eventInfo.DestinationAddress.AddrString);
8          doRouteReq(anEvent, addrString);
9          doDeassignCall(anEvent);
10     } else {
11         /* This section shouldn't be reached */
12     }
13 } while (true);
```

Fig. 5.8: Java source code for the application's logic main loop (in MyAppLogic)

At the beginning of the loop, line 2, the application tries to acquire the first event (`anEvent`) from the queue (`osaEventQueue`) by calling the `get()` method. If the queue is empty, the thread is blocked. It will be unblocked when an event appears. Otherwise the event is acquired immediately.

Once acquired, the event is stored in the `anEvent` variable. If the event is the **Address analyzed event** (it must be – no other events have been registered), the application calls the `translateModulo10()` method. This method is the "real" telecommunications service. As the argument it takes the original destination address and the new destination address is returned.

The `translateModulo10()` method's body is shown in Figure 5.9 but it could be any other method of the similar API (getting a string and returning another string).

```
1  String translateModulo10(String addressToTranslate) {
2      int addrInt = Integer.parseInt(addressToTranslate);
3      String addressTranslated;
4      if(addrInt > 10) {
5          addrInt = (addrInt % 10);
6          addressTranslated = Integer.toString(addrInt);
7      } else {
8          addressTranslated = addressToTranslate;
9      }
10     return addressTranslated;
11 }
```

Fig. 5.9: Java source code for the MyAppLogic.translateModulo10() method

The address is converted into integer, the modulo operation is undertaken and the result is converted back to string. Finally, this result is returned.

Coming back to Fig. 5.8, in the next step the new string is used to route the call to the new destination address. It is done by calling the `doRouteReq()` method (line 8). This method is a private wrapper to `routeReq()` and is presented in Fig. 5.10.

```
1  private void doRouteReq(MyAppEvent event, String newDestination) {
2      IntHolder callLegSessionID = new IntHolder();
3      try {
4          event.callId.CallReference.routeReq(
5              event.callId.CallSessionID,
6              new TpCallReportRequest[0],
7              nonOsaCreateE164Address(newDestination),
8              event.eventInfo.OriginatingAddress,
9              event.eventInfo.OriginalDestinationAddress,
10             event.eventInfo.DestinationAddress,
11             event.eventInfo.CallAppInfo,
12             callLegSessionID);
13     } catch (TpGeneralException e) {
14         /* Exception handling */
15         } catch (Exception exc) {
16         /* Exception handling */
17     }
18 }
```

Fig. 5.10: Java source code for the MyAppLogic.doRouteReq() method – call routing

The `routeReq()` method is used to request the call to be routed to the destination address. The method takes 8 parameters. The most important parameter is the third one, which is the destination address.

After successful `doRouteReq()` call, the application deassigns from the call by calling the `doDeassignCall()` method. The method is presented in Fig. 5.11.

```
1 void doDeassignCall(MyAppEvent event) {
2     try {
3             event.callId.CallReference.deassignCall(
4             event.callId.CallSessionID);
5         } catch (Exception exc) {
6         /* Exception handling */
7     }
8 }
```

Fig. 5.11: Java source code for MyAppLogic.doDessignCall() – deassigning from the call

Upon successful calling of this method, the application comes back to the beginning of the loop taking the next event.

The application shown here is a very simplified (just-workable) Parlay/OSA client application. The main goal was to show that it works, and that after some initial effort, service creation can be very simple. The application presented here may be now upgraded (empty methods properly filled, exception handling included and, especially, the application logic altered).

Chapter 6

Conclusions

The main goal of this work was to describe Parlay/OSA APIs in simple form. Since the work was initially planned to be purely descriptive, the author is glad to be able to write and run a Parlay/OSA application (mostly due to Ericsson Parlay/OSA Simulator).

However, probably in future, Parlay/OSA applications will not be developed in the manner presented here (using raw APIs). Applicationis will be generated by means of Parlay/OSA SDKs, which will simplify the whole process and make the future applications more error-proof. It is also possible, that in future, most applications will be developed in a higher abstraction layer (using Parlay X). The raw Parlay/OSA APIs will be then employed in so-called Parlay X gateways.

This work does not discuss the professional techniques of service creation. Conformance testing, deadlock, livelock detection and other issues are not analysed here since it is quite complex area of knowledge not to be covered here. Refer to G. Holtzmann book [39] or other works on protocol/service development if you are interested in those issues.

Moreover, the code presented here is hand-written, no code-generators have been used. In case of more complex services, techniques like UML or SDL modelling may be very useful. The discussion of using such techniquies with Parlay/OSA APIs may be found in Koltsidas et al. work [40].

Appendix A

Acronyms

3GPP Third-Generation Partnership Project

API Application Programming Interface

CAMEL Customised Applications for Mobile Network Enhanced Logic

CAP CAMEL Applicaton Part/Protocol

CCCS Conference Call Control Service

CGI Cell Global Identity

CORBA Common Object Request Broker Architecture

ETSI European Telecommunications Standards Institute

GCCS Generic Call Control Service

GSM Global System for Mobile Communication

IDL Interface Description Language

IN Intelligent Network

IN Internet Protocol

LAI Location Area Identity

MAP Mobile Applicaton Part

MCC Mobile Country Code

MMCCS MultiMedia Call Control Service

MNC Mobile Network Code

MPCCS MultiParty Call Control Service

OO Object-Oriented

OSA Open System Architecture

PSTN Pubic Switched Telephone Network

QoS Quality of Service

SAG Subscription Application Group

SCF Service Capability Feature

SCP Service Control Point

SCS Service Capability Server

SDK Software Development Kit

SIP Session Initiation Protocol

SLA Service Level Agreement

SMS Short Message System

TINA Telecommunications Information Networking Architecture

UL User Location

ULC User Location Camel

ULE User Location Emergency

UML Unified Modeling Language

UMTS Universal Mobile Telecommunications System

USSD Unstructured Supplementary Services Data

VLR Visitor Location Register

VASP Value Added Service Provider

VPN Virtual Private Network

VPrN Virtual Provisioned Network

VPrP Virtual Provisioned Pipe

WAP Wireless Application Protocol

XML eXtensible Markup Language

Appendix B

Glossary

The following definitions are taken from Parlay 3.0 specification [41]:

- **(Client) Applications** Services, which are designed using SCFs
- **Service Capabilities** Bearers defined by parameters, and/or mechanisms needed to realize services (they are located within networks and under network control).
- **Service Capability Feature** Functionality offered by service capabilities that are accessible via the standardized OSA interface.
- **Service Capability Server** Functional entity providing OSA interfaces toward an application.
- **Value Added Service Provider** Provides services other that basic telecommunications service for which additional charges may be incurred.

The following definitions, even though not specified directly in the specification, have the meaning as follows:

- **Application provider** An entity, which provides applications and controls them. Usually, terms “application provider” and “value added service provider” may be used interchangeably.
- **Enterprise Operator** The entity that subscribes SCFs for certain groups of client applications (note – it does not use those SCFs). In many cases enterprise operator does not have to be present in Parlay/OSA-based system. There is not always need for SCF subscribing
- **(Parlay/OSA) Entity** Each of the four functional items present in Parlay/OSA: application, enterprise operator, framework, or SCF provider.
- **Gateway** In the very first approach it is a collection of all SCSes (including the framework); in closer look it can also be a single SCS.

- **Network** The network operator's resources. The exact meaning depends on the SCF context: a "network" may be a whole PSTN system, IP network or even a single PC computer.
- **(Network) Operator** This term is widely used in Parlay/OSA documents. It means an entity that has control over certain network resources. Here, "operator" may be treated as a synonym for "SCF Provider".
- **SCF Provider** The entity which provides the network functionality to applications in form of Service Capability Feature (SCF). It must register into the framework before its capabilities may be used.
- **User** A network operator's subscriber, which uses its telecommunications services. This term appears in following contexts: to localize the *user*, to connect the *user* with, to give *users* an opportunity to, *user* interaction.

Appendix C

Source code

C.1 MyAppEvent.java

```
1  import org.open_service_access.cc.gccs.TpCallIdentifier;
2  import org.open_service_access.cc.gccs.TpCallEventInfo;
3
4  /**
5   * MyAppEvent is a class representing an event. It has 3 public
6   * attributes: call ID (call object reference + its descriptor),
7   * event details, and so-called assignment ID (integer).
8   */
9  public class MyAppEvent {
10     public TpCallIdentifier callId;
11     public TpCallEventInfo eventInfo;
12     public int assignmentID;
13
14     MyAppEvent(TpCallIdentifier cid,
15         TpCallEventInfo ei,
16         int a) {
17     callId = cid;
18     eventInfo = ei;
19     assignmentID = a;
20     }
21 }
```

C.2 MyAppEventQueue.java

```
1  import java.util.*;
2
3  /**
4   * MyAppEventQueue is a collection of events. Callback objects
5   * use this list to put any event that appear, and the appplication
6   * logic object uses this list to get events.
7   */
8  public class MyAppEventQueue {
9      List list;
10
11     public MyAppEventQueue() {
```

```
12          list = new ArrayList();
13      }
14
15      public synchronized void put(MyAppEvent event) {
16          list.add(event);
17          notifyAll();
18      }
19
20      public synchronized MyAppEvent get(){
21          if (size() == 0) do {
22              try {
23                wait();
24              } catch (Exception e) {
25                System.out.println("(MyAppEventQueue.get()): " + e);
26                System.exit(1);
27              }
28          } while (size() == 0);
29
30          return (MyAppEvent) list.remove(0);
31      }
32
33      public synchronized int size(){
34          return list.size();
35      }
36  }
```

C.3 MyAppInit.java

```
1  /* Ericsson's stuff, used to simpify the program */
2  import com.ericsson.parlay.application.serviceprovision.*;
3  import com.ericsson.tracedebug.*;
4
5  /* CORBA stuff */
6  import org.omg.CORBA.*;
7  /* CORBA naming services */
8  import org.omg.CosNaming.*;
9
10 /* Framework (not used in other modules) */
11 import org.open_service_access.fw_client.discovery.IpServiceDiscovery;
12 import org.open_service_access.TpService;
13 import org.open_service_access.IpInterface;
14 import org.open_service_access.TpGeneralException;
15
16 /* Call Control service manager */
17 import org.open_service_access.cc.gccs.IpCallControlManager;
18
19 /* Java Stuff */
20 import java.io.*;
21 import java.util.*;
22
23 /**
24  * The MyAppInit class is responsible for the application's initial sequence.
25  * If all succeeds, the MyAppInit object creats an MyAppLogic instance.
```

```
26    */
27  public class MyAppInit
28      implements ServiceStatusListener { // it must be implemented when using
29                                  // Ericsson's framework procedures
30      private ORB theORB;
31      /* These variables are used by Ericsson's Fw procedures */
32      public static String SIGNING_ALGORITHM = "P_NONE";
33      public static String CF_APP_ID = "test_application1";
34      public static String CF_APP_KEY = "3A47B193823F7D88";
35      private ServiceProvider theServiceProvider; // as defined by Ericsson
36
37      private IpCallControlManager theCCM;
38
39      public static void main(String args[]) {
40          try {
41              System.out.println(
42                  "(main): Starting the MyAppInit object");
43              new MyAppInit(args);
44          } catch (Exception e) {
45              System.out.println("(MyAppInit): Cannot proceed, exitting");
46          }
47      }
48
49      public MyAppInit(String args[]) throws Exception {
50          /* The following variables are for Ericsson simulator purposes */
51          ServiceProvisionConstants.TRACE_ACTIVE = false;
52          ServiceProvisionConstants.COMPILE_TARGET_IS_JSCS = false;
53
54          /* Let's get the ORB, connect to name service and
55             register to the framework */
56          theORB = ORB.init(args, null);
57          System.out.println("(MyAppInit): Connecting to OSA Gateway");
58          try {
59              theServiceProvider = new ServiceProvider(theORB,
60                  CF_APP_ID, CF_APP_KEY, this);
61          } catch (Exception e) {
62              System.out.println("(MyAppInit): Couldn't connect");
63              throw e;
64          }
65
66          /* Get the Call Control Service Manager */
67          System.out.println("(MyAppInit): Getting GCCS Manager");
68          theCCM = CCMProvider.get(theServiceProvider, SIGNING_ALGORITHM);
69
70          /* And start the main application logic (AppLogic object) */
71          System.out.println("(MyAppInit): Starting AppLogic");
72          MyAppLogic appLogic = new MyAppLogic(theCCM);
73      }
74
75      public void serviceTerminated(IpInterface iface) {}
76  }
```

C.4 MyAppLogic.java

```
import org.open_service_access.*;
import org.open_service_access.cc.gccs.TpCallIdentifier;
import org.open_service_access.cc.gccs.TpCallEventInfo;
import org.open_service_access.cc.gccs.TpCallReport;
import org.open_service_access.cc.gccs.TpCallEventCriteria;
import org.open_service_access.cc.gccs.TpCallReportRequest;
import org.open_service_access.cc.gccs.TpCallNotificationType;
import org.open_service_access.cc.gccs.IpCallControlManager;
import org.open_service_access.cc.gccs.IpAppCallControlManager;
import org.open_service_access.cc.gccs.P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT;
import org.open_service_access.cc.TpGCCSException;
import org.open_service_access.cc.TpCallMonitorMode;
import org.open_service_access.ui.*;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.util.*;

/**
 * MyAppLogic implements the application's logic. It could be modified in
 * a very simple way. This version uses the GCCS SCF. The service is
 * address translation.
 */
public class MyAppLogic {
    MyAppEventQueue osaEventQueue;
    IpCallControlManager ccMgr;
    String number;

    MyAppLogic(IpCallControlManager ccMgr_param) {
        osaEventQueue = new MyAppEventQueue();
            ccMgr = ccMgr_param;

         start_logic();
    }

    /* The main procedure of the service */
    void start_logic(){
        System.out.println("(MyAppLogic): Starting " +
                      "monitoring for number " + number);

        /* Register number(s) at the gateway */
            monitorOrigNumbers(ccMgr,
            new AppCallControlManager(this), number);

                System.out.println("(MyAppLogic): Entering loop");
        do {
            /* Wait for network events */
                MyAppEvent anEvent = osaEventQueue.get();

            System.out.println("(MyAppLogic): " +
                "Got event. Event ID: " +
```

```
52                  anEvent.eventInfo.CallEventName +
53                  ", from address: " + anEvent.eventInfo.
54                  OriginatingAddress.AddrString);
55
56              /* Check the event (of what type is it) */
57              if (anEvent.eventInfo.CallEventName ==
58                          P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value) {
59                  /* Translate the address */
60                  String addrString  = translateModulo10(
61                  anEvent.eventInfo.DestinationAddress.AddrString);
62                  /* Route to new address */
63                  doRouteReq(anEvent, addrString);
64                  /* Deassign from call */
65                  doDeassignCall(anEvent.callId);
66                  } else {
67                          System.out.println("(MyAppLogic): unknown event");
68                      }
69              } while (true);
70      }
71
72      /* This method is called from AppCallControlManage.callEventNotify().
73      It puts an event to the queue */
74      public void callEventNotify (
75          TpCallIdentifier callReference,
76          TpCallEventInfo eventInfo,
77          int assignmentID) {
78              osaEventQueue.put(new MyAppEvent(callReference,
79              eventInfo, assignmentID));
80      }
81
82      public void routeRes (int callSessionID,
83          TpCallReport eventReport,
84          int callLegSessionID)
85      {}
86
87      /* This method routes the party (identified by event.callId) to the
88      new address (specified in the newDestination parameter) */
89      private void doRouteReq(MyAppEvent event, String newDestination) {
90              IntHolder callLegSessionID = new IntHolder();
91          try {
92              event.callId.CallReference.routeReq(
93                  event.callId.CallSessionID,
94                  new TpCallReportRequest[0],
95                          myAppCreateE164Address(newDestination),
96                  event.eventInfo.OriginatingAddress,
97                  event.eventInfo.OriginalDestinationAddress,
98                  event.eventInfo.DestinationAddress,
99                  event.eventInfo.CallAppInfo,
100                         callLegSessionID);
101         } catch (TpGeneralException e) {
102                     System.out.println("(MyAppLogic): " +
103             "Caught OSA exception, num: " + e.exceptionType);
104             } catch (Exception exc) {
```

```
105             System.out.println("(MyAppLogic): " +
106                 "doRouteReq() exception: " + exc);
107         }
108     }
109 
110     /* Deasigns the application from the call */
111         void doDeassignCall(TpCallIdentifier callId) {
112         try {
113                 callId.CallReference.deassignCall(
114                 callId.CallSessionID);
115             } catch (Exception exc) {
116                     System.out.println("(MyAppLogic): " +
117                 "deassignCall() exception: " + exc);
118         }
119     }
120 
121     /* Registers event "address analysed event" at the gateway.
122     The originating party is specified (by the originating_address
123     parameter) */
124     int monitorOrigNumbers(IpCallControlManager mgr,
125                     IpAppCallControlManager appMgr,
126                     String originating_address) {
127 
128             TpCallEventCriteria ec = createOrigEventCriteria(
129                 originating_address, new String("*"),
130                 P_EVENT_GCCS_ADDRESS_ANALYSED_EVENT.value);
131             IntHolder assignment = new IntHolder();
132 
133             System.out.println("(MyAppLogic): " +
134             "calling enableCallNotification()");
135             try {
136         /* This is the Parlay/OSA event registration operation */
137         mgr.enableCallNotification(appMgr,
138             ec, assignment);
139             } catch (TpGeneralException e) {
140                     System.out.println("(MyAppLogic): " +
141             "Caught OSA general exception in" +
142             " enableCallNotification(), num: " + e.exceptionType);
143             } catch (TpGCCSException e) {
144                         System.out.println("(MyAppLogic):" +
145                         "Caught GCCS exception in " +
146             "enableCallNotification(), num:" + e.exceptionType);
147             }
148     return assignment.value;
149     }
150 
151     /* This method creates an appropriate TpCallEventCriteria structure,
152     containing the event description and monitored addresses */
153     TpCallEventCriteria createOrigEventCriteria(String originating,
154                 String destination, int event_num) {
155         TpCallEventCriteria ec = new TpCallEventCriteria();
156         ec.DestinationAddress = myAppCreateE164Address(destination);
157         ec.OriginatingAddress = myAppCreateE164Address(originating);
```

```
158         ec.CallEventName = event_num;
159         ec.CallNotificationType = TpCallNotificationType.P_ORIGINATING;
160         ec.MonitorMode = TpCallMonitorMode.P_CALL_MONITOR_MODE_INTERRUPT;
161         return ec;
162         }
163 
164     /* This method returns an E.164 address structure (based on
165     the address parameter) */
166     TpAddress myAppCreateE164Address(String address) {
167         TpAddress addr = new TpAddress();
168         addr.Plan = TpAddressPlan.P_ADDRESS_PLAN_E164;
169             addr.Presentation =
170             TpAddressPresentation.P_ADDRESS_PRESENTATION_ALLOWED;
171             addr.Screening = TpAddressScreening.
172             P_ADDRESS_SCREENING_USER_VERIFIED_PASSED;
173         addr.AddrString = new String(address); // Address string
174             addr.Name = new String("");
175             addr.SubAddressString = new String("");
176         return addr;
177     }
178 
179     /* This method translates an address string into a new (destination)
180     address */
181     String translateModulo10(String addressToTranslate) {
182             int addrInt = Integer.parseInt(addressToTranslate);
183         String addressTranslated;
184         if(addrInt > 10) {
185             addrInt = (addrInt % 10);
186             addressTranslated = Integer.toString(addrInt);
187         } else {
188             addressTranslated = addressToTranslate;
189         }
190         return addressTranslated;
191     }
192 }
```

C.5 AppCall.java

```
1  import org.open_service_access.cc.TpCallFault;
2  import org.open_service_access.cc.TpCallError;
3  import org.open_service_access.cc.gccs._IpAppCallImplBase;
4  import org.open_service_access.cc.gccs.TpCallReport;
5  import org.open_service_access.cc.gccs.TpCallEndedReport;
6  import org.open_service_access.cc.gccs.TpCallInfoReport;
7 
8  /**
9   * AppCall is a class implementing IpAppCall interface.
10  * This is a collection of callback methods. Most of them
11  * are still empty now.
12  */
13 public class AppCall extends _IpAppCallImplBase {
14     MyAppLogic logic;  // reference to appLogic
15                        // all events are passed there
```

```
16
17      public void routeRes(int callSessionID,
18          TpCallReport eventReport,
19          int callLegSessionID) {
20      logic.routeRes(callSessionID, eventReport, callLegSessionID);
21      }
22
23      public void routeErr(int callSessionID,
24              TpCallError errorIndication,
25          int callLegSessionID) {
26      System.out.println("(AppCall): routeErr() called");
27      }
28
29      public void getCallInfoRes(int callSessionID,
30          TpCallInfoReport callInfoReport) {
31      System.out.println("(AppCall):  getCallInfoRes() called");
32      }
33
34      public void getCallInfoErr(int callSessionID,
35          TpCallError errorIndication) {
36      System.out.println("(AppCall): getCallInfoErr() called");
37      }
38
39      public void superviseCallRes(int callSessionID,
40              int report, int usedTime) {
41      System.out.println("(AppCall):  superviseCallRes() called");
42      }
43
44      public void superviseCallErr(int callSessionID,
45          TpCallError errorIndication) {
46      System.out.println("(AppCall):  superviseCallErr() called");
47      }
48
49      public void callFaultDetected(int callSessionID,
50          TpCallFault fault) {
51      System.out.println("(AppCall):  callFaultDetected() called");
52      }
53
54      public void getMoreDialledDigitsRes(int callSessionID,
55          String digits) {
56      System.out.println("(AppCall):  getMoreDialledDigitsRes() called");
57      }
58
59      public void getMoreDialledDigitsErr(int callSessionID,
60          TpCallError errorIndication) {
61      System.out.println("(AppCall): getMoreDialledDigitsErr() called");
62      }
63
64      public void callEnded(int callSessionID,
65          TpCallEndedReport report) {
66          System.out.println("(AppCall): callEnded() called");
67          }
68
```

```
69      /** Constructor */
70      AppCall(MyAppLogic logicReference) {
71          logic = logicReference;
72      }
73  }
```

C.6 AppCallControlManager.java

```
1   import org.open_service_access.cc.gccs._IpAppCallControlManagerImplBase;
2   import org.open_service_access.cc.gccs.TpCallIdentifier;
3   import org.open_service_access.cc.gccs.TpCallEventInfo;
4   import org.open_service_access.cc.gccs.IpAppCallHolder;
5
6   /**
7    * AppCallControlManager is a class implementing IpAppCallControlManager
8    * interface. The most important callback method is callEventNotify().
9    * The rest are empty.
10   */
11  public class AppCallControlManager extends _IpAppCallControlManagerImplBase {
12      MyAppLogic logic;
13
14      public void callEventNotify(
15          TpCallIdentifier callReference,
16          TpCallEventInfo eventInfo,
17          int assignmentID,
18              IpAppCallHolder appCallRef) {
19
20      System.out.println("(AppCallControlManager): callEventNotify() called");
21
22      appCallRef.value = new AppCall(logic);
23      logic.callEventNotify(callReference,
24          eventInfo, assignmentID); /* push it further */
25      }
26
27      public void callAborted(int callReference) {
28      System.out.println("(AppCallControlManager): callAborted() called");
29      }
30
31      public void callNotificationInterrupted () {
32      System.out.println("(AppCallControlManager):" +
33          " callNotificationInterrupted() called");
34      }
35
36      public void callNotificationContinued () {
37          System.out.println("(AppCallControlManager): " +
38          "callNotificationContinued() called");
39      }
40
41      public void callOverloadEncountered (int assignmentID) {
42          System.out.println("(AppCallControlManager): " +
43          "callOverloadEncountered() called");
44      }
45
```

```
46      public void callOverloadCeased (int assignmentID) {
47          System.out.println("(AppCallControlManager):" +
48          "callOverloadCeased() called");
49      }
50
51      /** Constructor */
52      AppCallControlManager(MyAppLogic logicReference) {
53          logic = logicReference;
54      }
55  }
```

Bibliography

[1] J.L. Bakker et al. Rapid Development and Delievery of Converged Services Using APIs. *Bell Labs Technical Journal*, 7-9 2000.

[2] Appium. GBox Application platform. <http://www.appium.com>.

[3] Ericsson. JAMBALA Platform. <http://learning.ericsson.net/in/jambala.html>.

[4] Ericsson. OSA/Parlay Simulator, 2002. Available from: <http://www.ericsson.com/mobilityworld>, registration required.

[5] Steve Davis Simon Beddus, Gary Bruce. Opening Up Networks with JAIN Parlay. *IEEE Communications Magazine*, pages 136–143, April 2000.

[6] TINA-Consortium. TINA Service Architecture Version 5.0, 1997. Available from: <http://www.tinac.com>.

[7] EURESCOM. Eurescom project p909. Deliverables and sources available from: <http://www.eurescom.de>.

[8] Martin Cookson Steve Davis. Parlay Septemberfest: View of the Future, September 2001. Available from: <http://www.parlay.org>.

[9] Zygmunt Lozinski. Parlay. In *Eurescom P1110 WorkShop*, Heidelberg, Germany, February 2002.

[10] Ard-Jan Moerdijk and Lucas Klostermann. Opening the Networks with Parlay/OSA APIs: Standards and Aspects behind the APIs. Draft version to be resubmitted to IEEE Communications Magazine.

[11] Michał Rój and Jarosław Domaszewicz. Tworzenie usługi telekomunikacyjnej wykorzystującej Parlay/OSA API. In *Krajowe Sympozjum Telekomunikacyjne*, September 2002.

[12] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling And Design*. Prentice-Hall, 1991.

[13] Martin Fowler, Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley.

[14] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language: User Guide*. WNT, Warszawa, 2001. Polish translation: "UML: podręcznik użytkownika".

[15] Bernd Oestereich. *Developing software with UML*. Addison-Wesley and Longman, 1999.

[16] Object Management Group. OMG Unified Modeling Language Specification, version 1.3, 1999. Available from: <http://www.rational.com/uml>.

[17] Object Management Group. CORBA Homepage. <http://www.omg.org/corba>.

[18] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, 1996.

[19] Bruce Eckel. *Thinking in Java*. 2000. Available from: <http://www.BruceEckel.com>.

[20] Sun Microsystems. Java(TM) 2 SDK, Standard Edition Documentation. Technical report, 2000. Available from: <http://java.sun.com>.

[21] Sun Microsystems. The Java(TM) Tutorial: A practical guide for programmers. Technical report. Available from: <http://java.sun.com/docs/books/tutorial>.

[22] Object Management Group. OMG IDL Syntax and Semantics, 2000. Part of CORBA v2.4 documentation, Available from: <http://www.omg.org>.

[23] Sun Microsystems. The Java(TM) Standard Edition (J2SE). Available from: <http://java.sun.com/j2se>.

[24] Open Service Access; API. ETSI Standard ES 201 915, ETSI/The Parlay Group, 2001.

[25] OSA API, Part 3: Framework. ETSI Standard ES 201 915-3, ETSI/The Parlay Group, 2001.

[26] OSA API, Part 4: Call Control SCF. ETSI Standard ES 201 915-4, ETSI/The Parlay Group, 2001.

[27] OSA API, Part 5: User Interaction SCF. ETSI Standard ES 201 915-5, ETSI/The Parlay Group, 2001.

[28] OSA API, Part 6: Mobility SCF. ETSI Standard ES 201 915-6, ETSI/The Parlay Group, 2001.

[29] OSA API, Part 7: Terminal Capabilities SCF. ETSI Standard ES 201 915-4, ETSI/The Parlay Group, 2001.

[30] OSA API, Part 8: Data Session Control SCF. ETSI Standard ES 201 915-8, ETSI/The Parlay Group, 2001.

[31] OSA API, Part 9: Generic Messaging SCF. ETSI Standard ES 201 915-9, ETSI/The Parlay Group, 2001.

[32] OSA API, Part 10: Connectivity Manager SCF. ETSI Standard ES 201 915-10, ETSI/The Parlay Group, 2001.

[33] OSA API, Part 11: Account Management SCF. ETSI Standard ES 201 915-11, ETSI/The Parlay Group, 2001.

[34] OSA API, Part 12: Charging SCF. ETSI Standard ES 201 915-12, ETSI/The Parlay Group, 2001.

[35] M.Wegdam J.W. Hellenthal, F.J.M Panken. Validation of the Parlay API through prototyping. In *Proceedings of IEEE Intelligent Networks Workshop*, 6-8 May 2001.

[36] Gabriel Weitoft, Petter von Dolwitz. Integrating the Intelligent Network with Next Generation Service Architectures. Master's thesis, Department of Communication Systems; Lund Institute of Technology, 2000.

[37] P. Ebben. On Integrity of Telecommunications Networks when Controlled via the Parlay API. Master's thesis, Katolieke Universiteit Nijmegen, 2001.

[38] OSA APIs Specification. 3GPP Standard TS 29.198 (R4), Third Generation Partnership Project, 2001. Available from: <http://www.3gpp.org/ftp/Specs/latest>.

[39] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[40] Markos Koltsidas, Ognjen Ornjat, Lionel Sacks. Development of Parlay-based Services Using UML and SDL. In *Third IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS)*, September 2000. Available from: <http://www.ee.ucl.ac.uk/ oprnjat/Prnj00c.pdf>.

[41] OSA API, Part 1: Overview. ETSI Standard ES 201 915-1, ETSI/The Parlay Group, 2001.