The provision of a Generic Application (GApp) Layer for the Parlay/OSA Architecture

Opeyemi Oni

A project report submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, South Africa, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, June 2006

Declaration

I declare that this project report is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg, South Africa. It has not been submitted before for any degree or examination at any other university.

Signed on this the ____ day of ____________ 20__

____

Opeyemi Oni.

Abstract

The OSA/Parlay architecture supports the development of applications that control network connections through an open API. This research presents a proposal on improving the rate at which applications are developed and deployed using the Parlay/OSA architecture. The work seeks to facilitate software reuse by providing logical groupings in the application layer of the Parlay/OSA architecture.

This research presents a new layer to provide a higher level of abstraction for application developer using Parlay to provide telecommunication services. The layer introduced is referred to as the Generic Application Programming (GApp) interface. This document details the design and implementation of this interface.

Acknowledgements

The following research was performed under the auspices of the Center for Telecommunications Access and Services (CeTAS) at the University of the Witwatersrand, Johannesburg, South Africa. This center is funded by Telkom SA Limited and Siemens Telecommunications. This financial support was much appreciated.

I would like to extend thanks to my supervisors, Prof. Hu Hanrahan for his guidance and assistance throughout the duration of this research project. In addition, I would like to thank Thabo Machethe, Mike Powell, Rolan Christian, David Vannuci, Brain Leke, Dany Kabongo, Andile Shiba and my other colleagues at CeTAS for their valuable inputs during the research. Finally, I would like to thank my fiance (Prudence Ramabulana), my brothers (Bola, Banji and Bunmi oni) and parents (Stephen and Eunice Oni) for their love, support and patience throughout my tertiary studies. Without them, I would have not been able to achieve my goals and aspirations.

Contents

Declaration i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures viii

List of Tables xii

Acronyms xiii

1 Introduction 1

1.1 Next Generation Network (NGN) 1

1.1.1 The Existing Network 2

1.1.2 General NGN Characteristics 3

1.2 Comparison of the NGN with its key Predecessors 4

1.3 NGN Architecture 5

1.4 Business Model for NGN 7
1.5 NGN Strategies 8
1.6 Research Justification and Aim 8
1.7 Outline of Report 10

2 Background 12

2.1 Parlay 12
2.1.1 Objectives of the Parlay Group 13
2.1.2 Architectural view point 14
2.1.3 Framework Interface 15
2.1.4 Parlay APIs 17
2.2 Parlay X 23
2.3 Telecommunication Information Networking Architecture (TINA) . 26
2.3.1 TINA service features 27
2.4 Chapter Summary 30

3 Architecture and Design Principles 32

3.1 Research Focus 32
3.2 Architectural Overview 33
3.2.1 Structural 33
3.2.2 Service Session 35
3.2.3 Overview of GApp use case analysis 37

3.3 The GApp Design Principle 37

3.4 Chapter Summary 40

4 GApp Components Design **41**

4.1 GApp Frame Work 41

4.1.1 Service Discovery 44

4.1.2 Service Selection 45

4.2 GApp API Manager Classes 46

4.2.1 GApp GCC Manager 52

4.2.2 GApp MPCC Manager 54

4.2.3 GApp MMCC Manager 60

4.2.4 GApp CCC Manager 61

4.2.5 GApp UI Manager 64

4.3 Chapter Summary 65

5 Implementation of GApp layer **66**

5.1 Overview 66

5.2 Client (ASP) and Server (SCS) 67

5.3 Usage Scenario 70

5.4 Chapter Summary 71

6 Conclusion **72**

References 74

Appendix 77

A GAppGCCLogic 78

B GAppMPCCLogic 81

C GAppUILogic 85

List of Figures

1.1 NGN Abstracted Architecture 6

1.2 Business Model for NGN 7

1.3 Current Parlay Implementation 9

1.4 Target Parlay Implementation 10

2.1 Parlay Architecture 14

2.2 Frame work services setup 17

2.3 MPCC API class diagram 21

2.4 Click-to-Dial implementation on Parlay 22

2.5 Parlay Architecture 24

2.6 Click-to-Dial using Parlay X API 25

2.7 Click-to-Dial implementation on proposed GApp interface 29

3.1 Architectural View 34

3.2 Service Session Graph [1] 36

3.3 Class diagram for GApp manager inheritance 38

3.4 Sequence of events used to accomplish *opname*1() in figure 3.3 . . 38

3.5 Class diagram for a GApp child Manager with a new functional block 39

3.6 Sequence diagrams to accomplish parent and child class methods of figure 3.5 on Parlay SCF 39

3.7 Class diagram for a GApp child Manager with a changed functional block 39

3.8 Sequence diagrams to accomplish parent and child class methods of figure 3.7 on Parlay SCF 39

4.1 GApp Framework class 42

4.2 Application Initiates Service 42

4.3 Obtaining Service Information 44

4.4 Selecting Service 45

4.5 GApp Call manager inheritance Structure 47

4.6 GApp service relationships 47

4.7 Application Gains control of the Call through notification 48

4.8 Application creates a call object 49

4.9 Application releases Call 50

4.10 Application ends service 50

4.11 Asynchronous Methods 51

4.12 Synchronous methods on the SCF interface 51

4.13 Synchronous methods on the Application call back interface 52

4.14 Application releases call 53

4.15 Application adds party 54

4.16 Application starts existing session members 55

4.17 Application adds new members to call object 56

4.18 Application releases call 57

4.19 Application obtain member's information 58

4.20 Application obtain session information 58

4.21 Application suspend or resumes call 59

4.22 Media Session Initiation 60

4.23 Initiate Conference Service 62

4.24 Application creates a user interaction call 64

5.1 Experimental Setup 67

5.2 Linked List Data Type 68

5.3 Welcome Menu 70

5.4 Main Menu 71

5.5 Client output for create call 71

5.6 Server output for create call 71

A.1 Application Sets up Call 78

A.2 Application ends Service 78

A.3 Asynchronous Methods 79

A.4 Synchronous Method on SCF Interface 79

A.5 Synchronous Method on Call Back Interface 80

B.1 Application Sets up Call 81

B.2 Application ends Service 82

B.3 Asynchronous Methods 82

B.4 Synchronous Method on SCF Interface 83

B.5 Synchronous Method on Call Back Interface 84

C.1 Application ends Service 85

C.2 Asynchronous Methods 86

C.3 Synchronous Method on SCF Interface 86

C.4 Synchronous Method on Call Back Interface 87

List of Tables

1.1 Comparison of the NGN and its key Predecessors [2] 5

2.1 TINA Feature sets and allowed operation 28

Acronyms

API	Application Programming Interface
ASP	Application Service Provider
CORBA	Common Object Request Broker Architecture
CSM	Communications Session Manager
ASP	Application Service Provider
IDL	Interface Definition Language
JAIN	Java API's for Integrated Networks
NGN	Next Generation Network
OMG	Object Management Group
OO	Object Oriented
ORB	Object Request Broker
OSA	Open Services Access
PSTN	Public Switched Telephone Network
NO	Network Operator
SCF	Service Capability Feature
FW	Framework
GCC	Generic Call Control
MPCC	Multi-Party Call Control
MMCC	Multi-Media Call Control
CCC	Conferenec Call Control
UI	User Interaction
SSM	Service Session Manager
FS	Feature Set
APL	Application Logic
TINA	Telecommunications Information Networking Architecture
UML	Unified Modeling Language
USM	User Service Session Manager
GApp	Generic Application
GAppFW	Generic Application's Framework
GAppGCC	Generic Application's Generic Call Control
GAppMPCC	Generic Application's Multi-Party Call Control
GAppMMCC	Generic Application's Multi-Media Call Control
GAppCCC	Generic Application's Conference Call Control
GAppUI	User Interaction
M	Manager
CB	Call Back

Chapter 1

Introduction

Over the past two decades there has been a convergence of fixed, mobile and packet based networks. Convergence allows non-telecommunication programmers the use of telecomm infrastructure. Thus, we have the enhancement of IT application through telecommunication infrastructure and the enhancement of telecommunication services through external service logic, leading to the concept of the next generation network (NGN).

One of the main objectives of the NGN is to enable an Application Service Provider (ASP) to easily create services through the exposed service logic provided by a Network Operator (NO). Therefore, telcos who provide a number of services will have many ASPs using their infrastructure. Opening up the network in this way allows competition amongst telcos and ASPs, since telcos need to offer services to attract ASPs and ASPs need to offer their customers applications that are transparent (service independent of network) to the user.

1.1 Next Generation Network (NGN)

To achieve communication and access to information anywhere, anytime, and in any form, the NGN is employed by the major role-players in the telecomm industry. The NGN is a multi-service bearer packet network that physically and logically separates packet switching from the service/call control intelligence. A NGN supports multiparty, multimedia, real-time and information services with service differentiation and mobility. The emergence of NGN is attributed to three factors

[3, 4]:

- Environmental Drivers: Deregulation and privatization of the telecommunication industry over the past two decades has lead to a competitive industry operating in an open market.
- Service and Market Drivers: There has been a continuing increase in the set of services that users demand (e.g. user and service mobility).
- Technological Drivers: There has been an emergence of distributed computing and communication.

Elaborating on the first point we observe that liberalization is leading to increased business competition and new business opportunities. Liberalization enables growth in the industry by allowing competition between existing and newly created companies. This competitive atmosphere is leveraged by the NGN as it provides a platform for new businesses to easily create and deploy new features faster and cheaper [5, 6]. Therefore, telcos that provide new innovative services at cheaper prices will emerge as the survivors in this new environment.

On analysis of the second point we observe that service and market drivers are focused on satisfying the user's personal and professional needs. This was defined in [7] as being *User Centric*. User centricity can further be divided into *user satisfaction* and *user mobility*. The ability to customize user profiles is *user satisfaction* while *user mobility* provides transparency (the same service, look and feel) across different networks and terminals.

The third point is analyzed in the next section where we examine how the NGN came about from existing networks.

1.1.1 The Existing Network

Today the two main networks that define our understanding of service provisioning through connectivity are the packet based networks (e.g. Internet) and circuit based networks (e.g. PSTN).

Due to the difference in the networks, alternate approaches were taken in providing each network. The Public Switched Telephone Network (PSTN) provided service to subscribers by placing intelligence in the network. This placing of intelligence sees a small group of developers develop services they think the market needs. They therefore try to develop an application that will be a “killer application” (an application that is successful in terms of sales) [8]. This method of development and deployment of services closes the market, since third-party service developers cannot use the Telco’s network to provide services they developed. The Internet brought a new architecture in which services are developed and deployed by service providers. This new architecture is due to the end-to-end principle whereby all intelligence is placed in the end systems. As a result of this, the Internet growth rate is tremendous as different service providers provide services to customers connected to the Internet. To provide an open network similar to that provided by the Internet, telcos proposed the idea of the Open Service Market [9]. The Open Service Market key attributes include [9, 5]:

- Allowing ASPs access to a telco’s network to deploy both voice and data value-added services.
- Allowing ASPs with little or no knowledge of telecommunication network technologies to develop and deploy value added services. This is achieved by providing ASPs with an abstract view of the underlying network.
- Removing all value-added service and associated service logic from the underlying network.

1.1.2 General NGN Characteristics

This section provides the characteristics of a NGN. The characteristics are provided by detailing the attributes of the NGN [2, 10, 11, 9, 12]:

- The transport network is packet based, supporting a variety of Quality of Service (QoS) levels for voice, video and data.
- The NGN provides broadband and multimedia services.
- The NGN supports user mobility.

- The NGN service architecture must be open (have standardized interfaces) to allow third-party service providers access to services provided by the network operators.
- The NGN separates service and associated service logic from the underlining network.
- The NGN terminals can hold intelligence. This helps support the services provided in the network.
- The NGN inter-works with legacy networks via signaling and media gateways.
- The NGN supports a variety of services, applications and mechanisms based on service building blocks (Application Programming Interface (API), including real time/ streaming/ non-real time services and multi-media) allowing a number of logical groupings in provisioning specific Application Logic (APL).

1.2 Comparison of the NGN with its key Predecessors

Table 1.1 shows the similarities and differences between the NGN and some of its key predecessors. As the attributes in table 1.1 show, the NGN inherits from both the PSTN and Internet, but it inherits more from the Internet than it does the PSTN. The major difference between the Internet and the NGN is that the Internet transports packets on a best effort basis (i.e., tries to deliver packets with no guarantee of delivery) whilst the NGN provides different levels of quality of service that ensures packet delivery. An important difference is that the Internet service providers do not federate, and that intelligence in the Internet is kept at the end system with the network strictly for providing transport [2].

Table 1.1: Comparison of the NGN and its key Predecessors [2]

Attribute	PSTN/IN	Internet	NGN
Multimedia services	No	Yes	Yes
QoS-enabled	Yes (voice)	No	Yes
Network intelligence	Yes	No	Yes
Intelligent Terminals	No (except ISDN)	Yes	Yes
Underlying transport network	TDM	Packet	Packet
Service Architecture	Semi structured	Ad hoc	Structured
Service Reliability	High	Low	High
Service Creation	Complex	Ad-hoc	Systematic
Ease of use of application services	Medium	High	High
Ease of creation of application services	Low	Medium	High
Evolvability or Modularity	Low	Medium	High
Time to market of service	Long	Short	Shorter
Architectural openness	Low	Medium	High

1.3 NGN Architecture

Nowadays voice communication has become a common commodity, the quest now is to provide value-added services that have the greatest ability to generate revenue. As the next money making application is unknown, it is not useful to spend enormous amount of money on infrastructure to enable a single application, as too many factors influence its commercial success. It is preferable to provide an environment that enables the creation of applications by providing services that are commonly needed for application development: and such an environment is referred to in [8] as the killer environment. To achieve this environment a new architecture with open access to, and programmability of network resources is required. This leads to the use of the Application Programming Interface (API) which has been used with much success in the IT world. An API provides application developers with programmability of software resources by defining these resources in terms of objects, methods, data types and parameters that operate on those objects [13]. In other words, a killer environment is an environment in which a number of ASPs provide services to their consumers using Open APIs provided by telcos. Hence a killer environment will allow ASPs to easily and quickly create and deploy any application without disrupting on going network services [8].

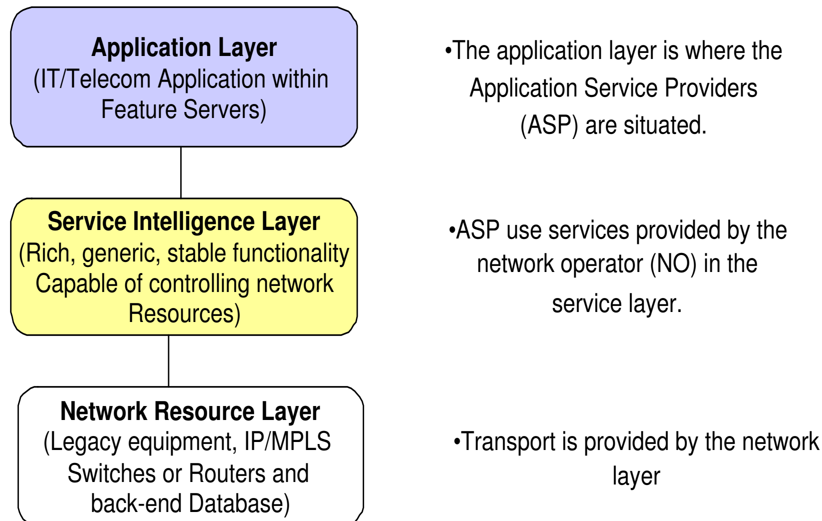

Figure 1.1: NGN Abstracted Architecture

To create a killer environment, the NGN architecture requires a clean separation of functions and domains with the maximum degree of reuse built into the architecture and its components [2]. As a result, the architecture should cater for middleware (in the sense of generic service control) that can hide the complexity of the transport layer from the ASP. This architectural requirement gives rise to the layers detailed in figure 1.1

- The Application Layer: contains the logic specific to the application i.e., contains the ASP applications.
- The Service Layer: A *service* is a software application that provides a tangible functionality for the Application layer. Therefore, the *service layer* contains data and logic required to perform services required by the application i.e it contains open APIs.
- The Network Layer: This layer provides the connectivity required by the service layer, according to QOS and policy constraints.

Two or more NGNs can be federated to provide more services to a user, implying that a user in one NGN domain can use applications or services provided in another NGN domain [7]. The architecture ensures the "killer enabler" (openness and programmability of network service capability) which allows a wider base of developers for the introduction of value added services and future applications [8].

1.4 Business Model for NGN

Figure 1.2 shows the model adapted for this research. This model is a modification of the TINA Consortium (TINA-C) business model. The modification was proposed in [9], where a movement was made from a telco centric model to an ASP centric model.

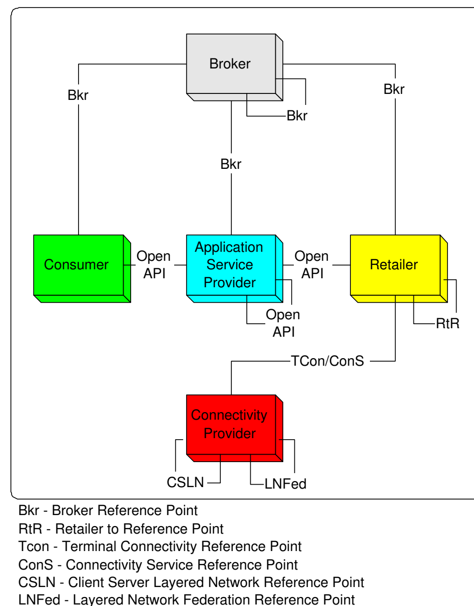

Figure 1.2: Business Model for NGN

The components specified in this diagram include:

- The **Broker**: it allows for service location between domains.
- The **Application Service Provider**: it develops applications for consumer.
- The **Consumer**: it represents the end user.
- The **Retailer/Network Operator**: it is responsible for the provisioning of reusable APIs that will enable the ASP to manipulate physical network resources during service provision.
- The **Connectivity Provider**: it provides physical network resources and infrastructure.

This approach allows for the use of open APIs. It is ASP centric, and the interface between the ASP, retailer and consumer is defined using a set of standardized and

open APIs. The federation of domains is accomplished through the broker and the transport is provided by the connectivity provider.

1.5 NGN Strategies

Different strategies that contribute towards NGN are suggested by different industry groups, these include Java APIs for Integrated Networks (JAIN), Telecommunications Information Networking Architecture (TINA) and Open Services Access (OSA)/Parlay [9].

Parlay was conceived by major telecommunication and Information Technology companies. The purpose is to produce APIs that enable third-party application service providers to develop value-added services that use the capabilities and resources existing in telecommunications networks. These APIs are of two types, known as the *Framework* and the *Service Capability Feature* (SCF). The Framework provides ASPs with controlled access to SCFs whilst the SCFs provide ASPs with access to network resources. Parlay is the NGN contributor this research focuses on. The Parlay specification is currently on version five at the time of writing, but the research was conducted using version four. The issues addressed in this research are based on version four but still apply in version five, as the specification has not changed with respect to the APIs detailed in this research.

1.6 Research Justification and Aim

The complexity faced by a Parlay service developers has increased due to the introduction of new functionality in the Parlay call control interface in version four of the specification. For example developers need knowledge on manipulation of telecomm related objects such as call-legs. This implies that Parlay developers using the Parlay gateway for development need telecommunication knowledge. To avoid this requirement a modification to the Parlay gateway is required. This modification is usually achieved by providing an abstract view of the Parlay gateway.

The current method of attaining abstraction usually reduces the programmer control over the gateway. For example, the Parlay X gateway is an abstraction of the

Parlay gateway. The Parlay X gateway provides an interface for Web developers to create and deploy services simply, by using simple synchronous method calls which reduce the functionality of the API in comparison to the Parlay gateway. For example, a request issued in Parlay X has to be satisfied before any further request is issued. As Parlay X focuses on providing abstraction for web programmers, we require a scheme that provides abstraction satisfying the needs of all programmers. Such a scheme must maintain the same level of functionality as the Parlay gateway and hide details that do not add functionality when providing a service with the Parlay gateway.

Therefore this research aims to provide a simple interface to enable application developers to easily create services without requiring telecommunication knowledge, whilst maintaining the same level of functionality currently provided by the Parlay gateway. In other words, the research **aims to provide an interface with a higher level of abstraction and the same level of functionality as the current Parlay gateway**.

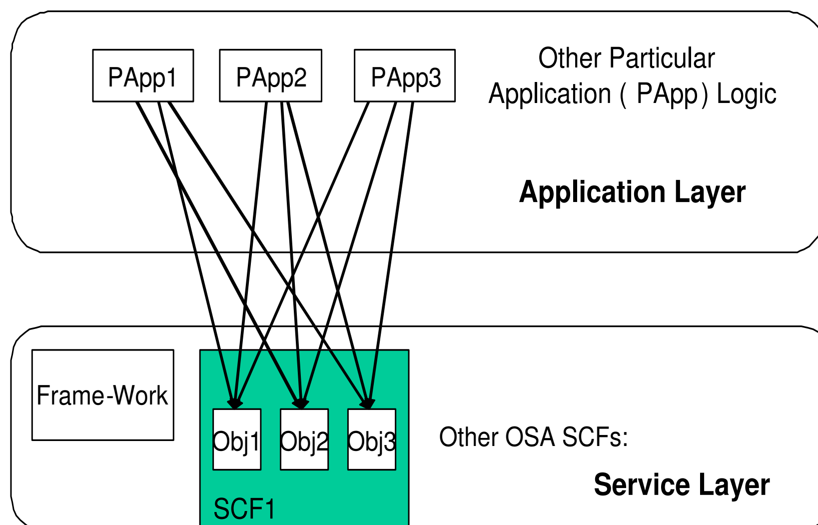

Figure 1.3: Current Parlay Implementation

Figure 1.3 (obj represent object in the figure) shows a simplified model of the current implementation of services on the Parlay gateway. Each application needs to coordinate the provision of its service in the gateway. As a result it needs to keep information on:

- identity of the objects in the SCS (i.e reference and object ID)

- methods available on the objects
- logical grouping of method calls on objects
- object life cycle management

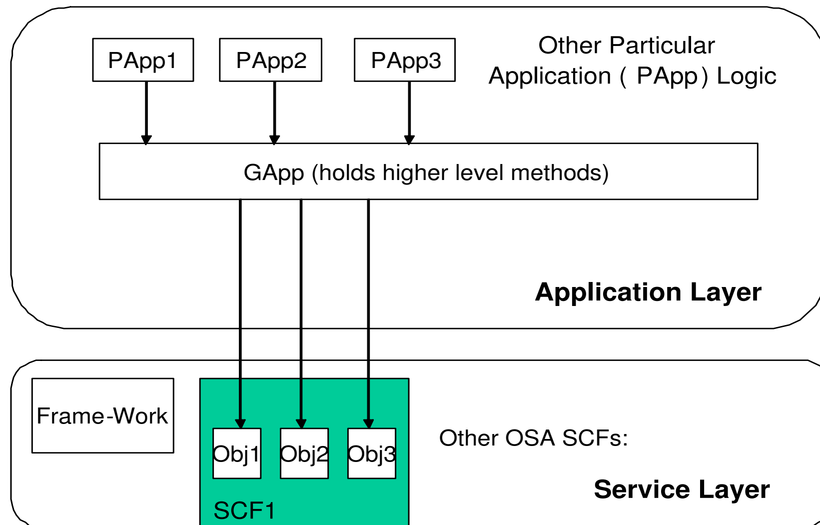

Figure 1.4: Target Parlay Implementation

This research aims to provide abstraction for the ASPs by providing a new layer that provides a higher level of abstraction for the Parlay gateway. This new layer is indicated in figure 1.4 as the Generic Application Programming (GApp) layer. The Particular Application (PApp) Logic contains application specific logic that use GApp functional blocks, therefore the application programmer provides the PApp logic through the functional blocks provided in the GApp. The GApp layer abstraction is attained by providing:

- a simple interface for the ASP to interact with.
- higher level methods on the GApp interface thereby simplifying the calls to the SCF.
- life cycle management of objects created in the SCF during the execution of a service instance.

1.7 Outline of Report

The rest of this report is organized as follows:

In Chapter 2, an overview of Parlay is given by providing the objectives of the Parlay API and detailing the architectural view in terms of the computational division. The TINA, and its service features are also examined together with other approaches that have been used to try to introduce a higher level of abstraction.

In Chapter 3, the architectural design is detailed for the extended Parlay model. This is done by looking at the computational interaction in the structural and service session models.

In Chapter 4, details of the design of the GApp components (GAppFW, GAppGCC, GAppCCC, GAppMPCC, GAppMMCC, GAppUI) are provided.

In Chapter 5, we provide an overview of the implementation of the GApp interface. This is done by detailing the process used to implement a GApp MPCC interface. Finally, in Chapter 6, conclusions are presented and recommendations for further work are suggested.

Chapter 2

Background

The chapter provides the reader with an understanding of the levels of abstraction currently provided for in Parlay and the level sought after for the Generic Application Programming (GApp) components. We therefore give an overview of Parlay by providing its objectives and detailing the architectural view in term of the computational division. We then detail Parlay X as a method of providing a higher level of abstraction for the Parlay gateway and finally we examine the Telecommunication Information Networking Architecture (TINA) and its service features as a possible contributor to an API of greater abstraction.

2.1 Parlay

The Parlay Group was initiated in 1998 by a community of operators, IT vendors, network equipment providers, and application developers. It was initiated to meet the need for innovative applications that combine network features with Internet services and critical enterprise data, thereby producing Application Programming Interface (API) specifications that would combine the best of the telecomm and IT worlds [14]. Parlay is a non-profit consortium of over 65 companies representing the telecommunication and IT worlds including Alcatel, British Telecom, Ericsson, Fujitsu, HP, IBM, Incomit, Lucent, NTT, Siemens, Sun, Telcordia Technologies, Telecom Italia and Teltier [15, 16].

The Parlay architecture provides an open platform for service creation and delivery

in future multi-service networks. Chapter 1 showed that NGN is achieved by placing a service platform between the application and network layer. Parlay provides this service platform through APIs. An API is thus a mechanism by which network capabilities are accessed. As a result, application programming is abstracted from protocol details within the network, which enables applications to evolve independently from the network.

The specification of Parlay focuses on a number of APIs including call control, user interaction and mobility management. The Parlay group specifies interaction between its components through UML definitions and ensures backward compatibility with previous versions.

2.1.1 Objectives of the Parlay Group

One of the main objectives of Parlay APIs is to encourage third-party application developers to develop applications on telco's infrastructure. These APIs are thus designed to be [9, 15]:

1. Open and secure. Despite the openness, security is a key issue since ASPs and telcos need to be assured of the security of APIs to assure the security of their domain.
2. Network and technology independent. Since the advent of Parlay APIs, there is the possibility of writing applications that can work on a variety of protocols. Therefore the APIs should address a broad range of network functionality. Thus allows for the combination of different network capabilities that is provided using different technologies [9, 17].
3. Simple to use and easily extendable through UML. This feature is to attract ASPs to develop new application using the easy-to-use APIs. This objective was not totally meet as the addition of new functionality to the APIs increased its complexity in the later versions of the standards.

2.1.2 Architectural view point

The service architecture in Parlay is divided into two parts, the application layer (that may be located in a third-party service provider domain) and the service layer (that contains the Service Capability Server (SCS) and is located in the network operator's domain). The application layer contains objects that implement the call-back interface and the application logic, whilst the service layer contains objects that provide two categories of interfaces:

- The Service Capability Feature (SCF): is a logical grouping of associated interfaces e.g., call control interfaces are grouped into the call control SCF.
- Framework: is a component whose main function is to cryptographically authenticate the application, and return object references to the application for those Parlay/OSA functions (or service capability features) it has been allowed to use by the service provider [15] i.e., it provides interfaces that support capabilities necessary for SCFs to be secure, resilient and manageable [9].

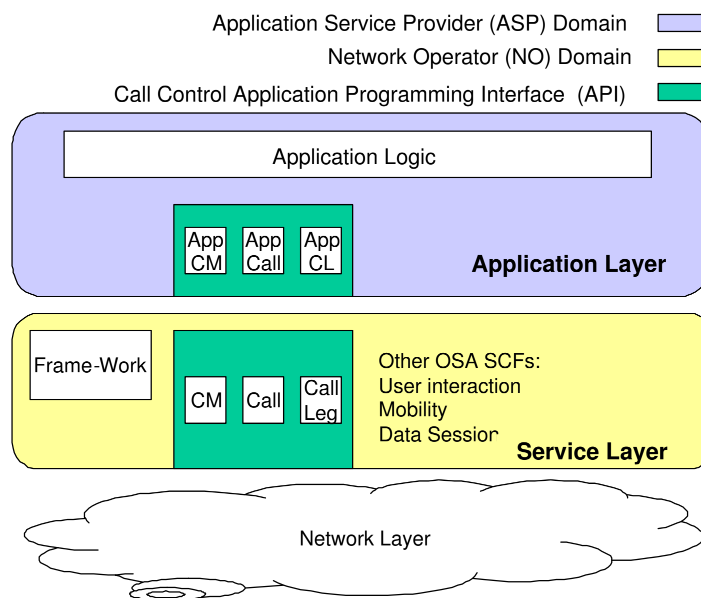

Figure 2.1: Parlay Architecture

Parlay uses the concept of APIs to describe a set of standardized programming interfaces. APIs are implemented on the client side (application layer) as callback interfaces and on the server side (service layer) as SCFs in SCSs, this is illustrated

in figure 2.1. As shown in figure 2.1, the client side API is implemented in the application domain and the SCFs are implemented in the server/Network Operator (NO) domain. Communication between client and server objects is achieved through Distributed Object Computing (DOC) (e.g., CORBA). The number of SCFs and richness of methods of each SCF makes Parlay API powerful.

The ASP accesses the services provided in the service layer through SCF interfaces. The SCFs are initially accessed through the Framework which stores information about the applications and creates the required computational object (manager) for each SCF that the application uses. Parlay thus separates application logic from generic service logic in the SCFs, implementing the APIs (for example call control, messaging and event notification) across the client and server sides.

2.1.3 Framework Interface

Framework interfaces provide the necessary capabilities for the SCFs to perform their functions while maintaining the system integrity. In this section, the interfaces enabling communication between an application and the framework is indicated by *FW_App* whilst the interfaces enabling communication between the framework and a SCS is indicated by *FW_SCS* . The framework offers the following interfaces [18]:

- Trust and Security Management (*FW_App*): serves as a contact point between the application and the framework interface, therefore it authenticates domains.
- Service Access (*FW_App*): ensures that the application is allowed to use specific SCFs and interfaces.
- Service Discovery (*FW_App*): enables the location of interfaces supported by the framework and SCFs Service.
- Registration (*FW_SCS*): enables the registration of interfaces at the framework to make a repository of interfaces available.
- Discovery and Service Registration (*FW_App*): enables an application to discover what interfaces are available and provides information about each interface.

- Event notification: reports the generic events that occur in the network.
- Integrity management (*FW_SCS*): is achieved through the Load manager, the Heartbeat Manager and Operation, Administration and Maintenance (OAM) interface.
  - Load manager: achieves load balancing according to a load management policy.
  - Heartbeat manager: allows the supervision of calls by keeping track of message invocation and states of different object instances [9].
- Service Life Cycle Manager (*FW_SCS*): creates a new instance of an API implementation.
- Contract management: manages the contract between different domains (e.g., between an application provider and a network operator).

The functionalities of the framework are explained by means of an example where a new SCS is installed and an application starts using it. These functionalities include:

1. Registration (adding a new SCS): This includes a SCS requesting a registration interface from the framework. After the framework provides this interface, the SCS uses it to publish its type and capabilities and provides the framework with an object reference allocated by the Service Life Cycle manager.
2. Setup of Service Agreement: The information supplied by the SCS to the framework and the condition under which an application is allowed to use this SCS is captured in a service agreement in the management system.
3. Run-time communication establishment: The sequence in figure 2.2 illustrates the events that occur when an application requires the capability offered at the SCS. The sequence in figure 2.2 is:
   - The application logic uses the service discovery interface to find the services that are registered at the Framework (1). The framework then returns the set of services the application is allowed to use (2). The application logic then selects a service (3); the framework then uses the SCS to create a service manager (indicated by M in a box) object (4) and the reference is returned to the application logic (steps (5)&(6)).

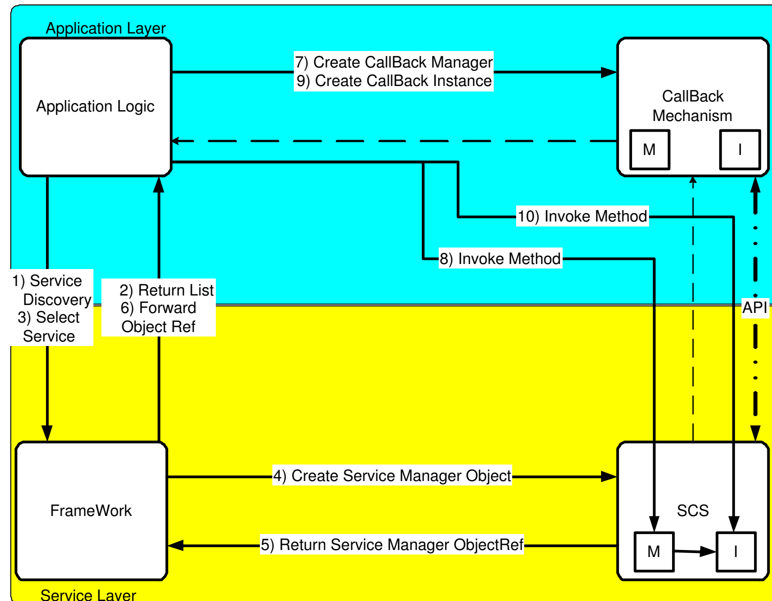

Figure 2.2: Frame work services setup

- The application logic then decides it needs to invoke a method on the SCF. It creates the callback interface for the object manager (7); the callback interface returns the reference of this object to the application logic (not shown). The application logic then uses the callback reference to invoke a method (for example to create a call) on the service manager object in the service layer (8), if the method invoked requires object instance creation an object instant is created (indicated by I in a box). The application logic can now create a callback object and invoke methods on the object instant in steps (9) & (10).

2.1.4 Parlay APIs

An API provides the capabilities for application to control physical network resources. Different APIs include [17, 15, 16]:

- Call Control: includes a range of APIs that provide different types of calls ranging from a basic call to a multimedia conference call.
- User Interaction: enables an application to obtain and provide information to the end user. Interaction is achieved through announcement, short messages etc.

- Mobility: enables an application to obtain location and request notifications on location update of a terminal.
- Data Session Control: enables application management of data sessions initiated from terminals. It is mainly used for GPRS applications.
- Generic Messaging: enables an application to access mailboxes, send and receive messages and interact with messaging systems, such as voice, FAX or email.
- Connectivity Manager: enables an application to manage the quality of service between the end systems in a Virtual Private Network.
- Terminal Capabilities: enables an application to determine the capability of end-user terminals.
- Presence and Availability Management: enables an application to set and obtain information about a user's presence and availability.
- Account Management: enables an application to query accounts and obtain transaction history.
- Charging: enables an application to request a particular payment method for service usage. (e.g "content-based charging" or reserve payment for future services "Pre-Paid Charging")
- Policy Management: enables an application to setup and register for policy related events.

Parlay Services

Parlay call service interfaces can be categorized according to the service management work they perform. This categorization sees the more common service management interfaces grouped together in the service manager objects. Elaborating on this, the interfaces include [19]:

- The Service manager: This object is created for each application that communicates with the SCF. Its purpose is to create new objects as needed by the application in the SCF. The combination of the service manager and the objects it creates are referred to as a *service instance*.

- Call: represents a relationship between parties i.e., it relates all parties that have an association. Therefore it facilitates communication between parties. Different applications can have access to the same call object (since different applications might have similar trigger points resulting in them being passed the same call object reference) but share different views, with one application at the originating side and the other at the terminating side. As a result, the applications are unaware of each other's presence. This implies that the entire call is released from the applications perspective if an application releases the call object.

- Call-Leg: The *CallLeg* object associates a Call object with an address. As a result, attaching a call-leg implies routing the call to the target address and enabling the media or bearer channel to allow communication with the other parties connected to the call.

Call Control API

This section details each Call control API catered for, by giving a brief explanation of each.

The Generic Call Control (GCC) API provides the basic call control services. These services are generally for two-party calls as the GCC API does not give explicit access to legs and media channels. The GCC allows for call-setup by an application or through the network. It provides access to network related services through its interfaces *IpCallControlManager* and *IpCall*. The SCF communicates with the application through the GCC call back API with interfaces *IpAppCallControlManager* and *IpAppCall*.

The Multi-Party call control (MPCC) API extends the GCC by providing leg management functionality. It allows for establishment of multi-party calls, with the total number of legs varying according to service requirements. Access to network related services is provided through interfaces *IpMultiPartyCallControlManager*, *IpMultiPartyCall* and *IpCallLeg*. The MPCC SCF communicates with the application through the MPCC call back interface *IpAppMultiPartyCallControlManager*, *IpAppMultiPartyCall* and *IpAppCallLeg*.

The Multi-Media call control (MMCC) API extends MPCC by providing multi-media capabilities. It allows for the association of media streams with call-legs. Access to network related services is provided through *IpMultiMediaCallControlManager*, *IpMultiMediaCall*, *IpMultiMediaCallLeg* and *IpMultiMediaStream*. The MMCC SCF communicates with the application through the MMCC call back interfaces *IpAppMultiMediaCallControlManager*, *IpAppMultiMediaCall*, *IpAppMMultiMediaCallLeg* and *IpAppMultiMediaStream*.

The Conference Call Control (CCC) API extends the MMCC by providing the ability to manipulate sub-conferences within a conference. A *subconference* describes the connection of call-legs within a conference. Access to the network resources is provided through *IpConfCallControlManager*, *IpConfCall* and *IpSubConfCall*. The CCC SCF communicates with the application through the CCC call back interfaces *IpAppConfCallControlManager*, *IpAppConfCall* and *IpAppSubConfCall*.

There is increased complexity in the new Parlay specification (Version four) due to the increased number of interfaces therefore making it difficult to use. The increased complexity is brought about since the MPCC, MMCC and CCC provide interfaces for detailed manipulation of resources (e.g call leg), as a result:

- An ASP needs to make sure that the interaction between the call manager objects, call objects and call-leg objects leave the gateway in a consistent state.
- An ASP needs to know the methods applicable to each computational object and how these methods combine to provide a service.

Familiarizing a non-telecomm expert with these computational objects and their interfaces will be difficult, which defeats the purpose of introducing the Parlay gateway. This research finds a solution to the problem.

To elaborate on the complexity we provide a class diagram of Parlay MPCC API in figure 2.3. As can be seen, there are six classes. Each class contains its own methods. Therefore the application programmer will need to understand the functionality attained by invoking a particular method on a particular interface. For example:

- one would need to set a notification point to be notified of network events during service provisioning.

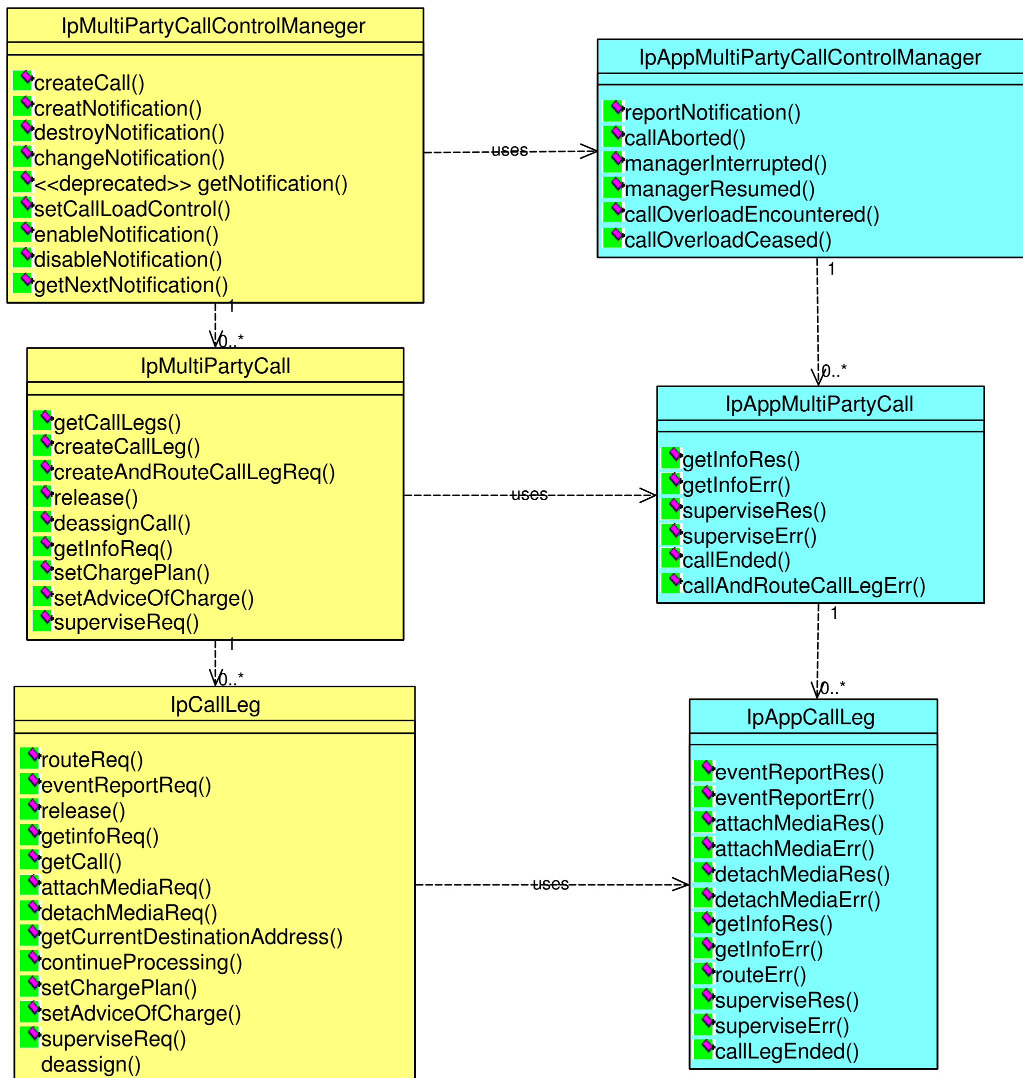

Figure 2.3: MPCC API class diagram

- to establish a call, one would need to create a call manager object, a call object, a call-leg and attach media to enable communication.

All these procedures and interaction of the interfaces make Parlay complex for developers without telecommunication knowledge as they need to coordinate their sequence of calls to the Parlay SCF interfaces.

We illustrate this complexity by providing an example of a Click-to-Dial application using the Parlay SCF in Figure 2.4 (A Click-to-Dial call is a two-party web call).

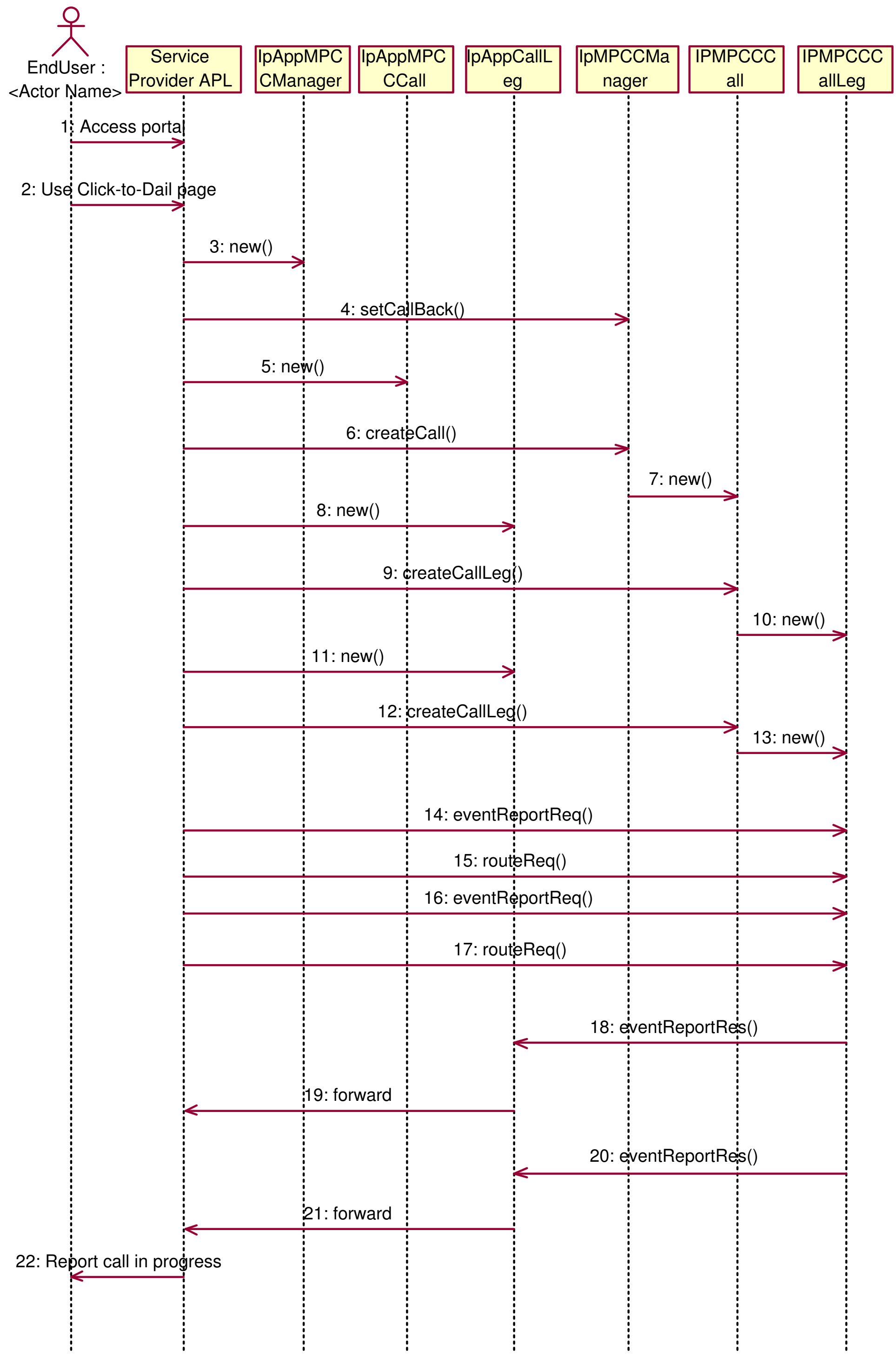

Figure 2.4: Click-to-Dial implementation on Parlay

- Figure 2.4 shows the *EndUser* starting a Click-to-Dial service (2). The service provider application logic (APL) provides a two-party call by invoking a number of operations on the Parlay gateway. Firstly it executes *new()* on *IpAppMPCCManager* (3), which creates a callback object and forwards the reference to *IpMPCCManager* through the method *setCallback()*(4).

- (5) The service provider application logic then creates another call back object *IpAppMPCCCall* and (6) forwards the call back object interface reference to *IpMPCCManager* through the *createCall()* operation, which then creates a new *IpMPCCCall* object (7).

- Messages 8,11 and 9,10 are then used for the creation of call back and call leg objects respectively. Thereafter the application routes to the call legs target addresses using messages 14-17. The corresponding responses are sent in messages 18-21. This is then indicated on the web page as call in progress (22).

As can be seen operations 3-21 need to be coordinated by the application service provider logic which implies the ASP programmer require some telecomm knowledge.

2.2 Parlay X

In order to provide a simple API for web developers, Parlay X API was conceived late in 2001 to provide a higher level of abstraction and specific capabilities that are not currently provided by the Parlay API. The Parlay API caters for the common aspects of fixed, mobile and packet networks but does not account for specific aspects such as support for SMS (Short Messaging Service). Also, programming on the Parlay interfaces to provide simple services such as Click-to-Dial is complicated due to the interaction of different components and interfaces.

The Parlay X interface therefore seeks to produce predominantly simpler APIs with reduced capabilities. This implies that developers who need advanced control over a SCF will use the Parlay API. As a result, the Parlay X API is to be predominantly deployed in a Web environment. This approach encourages the creation of application by Web developers by providing APIs with a level of telecommunication

knowledge suited to them.

Due to the fact that Parlay X provides specific capabilities some capabilities will not be mediated through the Parlay gateway and will rather be made available on the resource layer [8]. Figure 2.5 illustrates the key points mentioned.

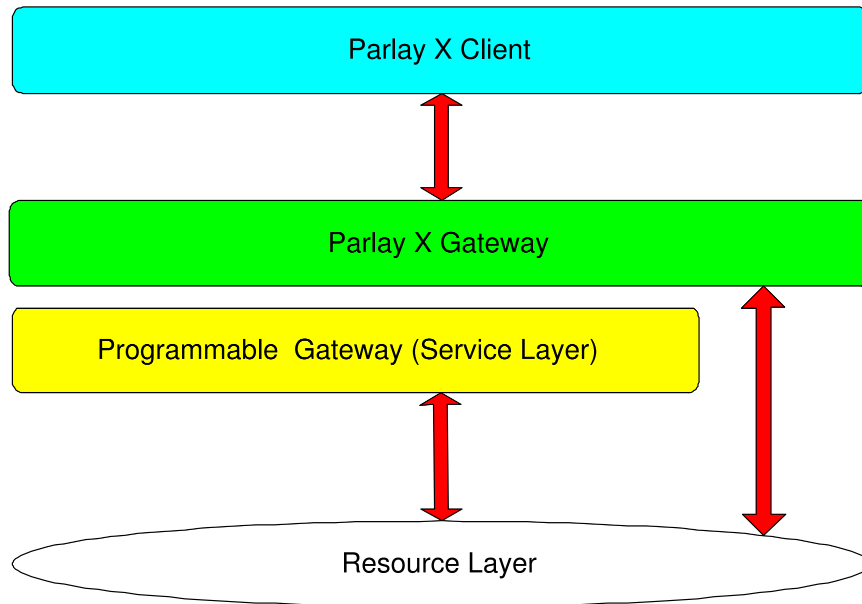

Figure 2.5: Parlay Architecture

Figure 2.6 shows the Parlay X implementation of the Click-to-Dial application. In this example the *MakeCallRequest()* message is invoked on the Parlay X API *ThirdPartyCallWebService*, which makes requests to set-up a voice call between two addresses (Calling Party and Called Party), provided that the invoking application is allowed to connect them. The results to this request is passed back in messages 5 as *MakeCallResponse()* which returns a call identifier for the call to be created. The call identifier is used by the service application logic as a parameter to *GetCallInformationRequest()*. The *GetCallInformationRequest()* method is then invoked to obtain information on the status of the call, therefore the service provider application logic may have to invoke it a number of times, before obtaining a favorable response, such as call in progress. On invoking *MakeCallRequest()* on the *ThirdPartyCallWebService*, it communicates with the Parlay gateway with a number of messages between 3-30. This message sequence is similar to messages 3-21 in figure 2.4 which is explained in section 2.1.4.

As compared to the Click-to-Dial in section 2.1.4 the telecomm knowledge required

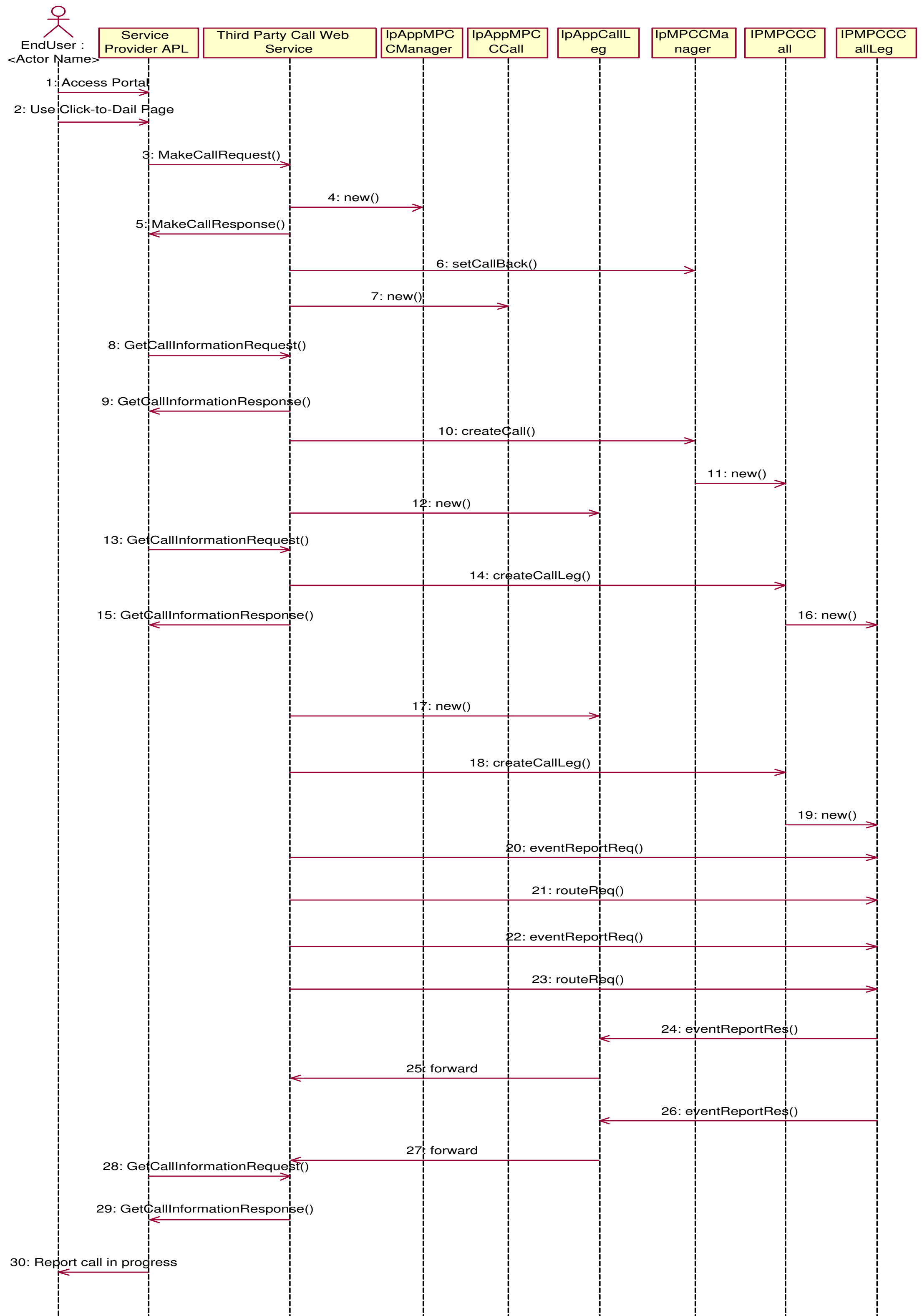

Figure 2.6: Click-to-Dial using Parlay X API

is minimal as a single call to *MakeCallRequest()* performs process 3-21 in figure 2.4. But due to the high level of abstraction the ASP programmer does not have access to certain Parlay objects such as the call leg and hence cannot perform operations that require it to modify the state of a particular call leg. This limits the programmer's control as he/she cannot perform operations (e.g. add party, suspend and create) on a particular party. Hence there is loss in functionality when compared to the Parlay Gateway. The service provider application logic also needs to continuously poll the *ThirdPartyCallWebService* in order to obtain information on the call status, this increases the complexity of Parlay X as the ASP programmers need to add polling facility to their programs.

2.3 Telecommunication Information Networking Architecture (TINA)

The TINA consortium was started in 1992 by about 40 companies in "an international collaboration aiming at defining and validating an open architecture for telecommunications systems for broadband, multi-media, and information era" [20]. The TINA architecture is made up of a number of sub-architectures. The architecture this project focuses on is the Service Architecture. This architecture provides for management, implementation, design and specification of telecommunication services. The service architecture in TINA encapsulates a number of concepts and principles. One relevant to this discussion is the *session concept*. A session is a period of time during which activities are performed to accomplish service goals. Of interest are the *service session* which provides management for a single service and a *user session* which provides the management of a user within a service session. TINA uses defined computational objects to implement sessions. The Service Session Manager (SSM) contains the service logic while a User Session Manager (USM) represents each party in the service. The SSM is used for overall service coordination and the USM is used for user specific service management [20]. TINA also defines feature sets by grouping interfaces on the SSM (and USM) by service types of different complexity. For example TINA defines a set of methods to suspend, modify and create calls. TINA is limited in that the available feature sets (FSs) are confined to call control and that the SSM and USM are complex computational objects, containing both generic and application specific logic. Work in [9]

demonstrates the separation of the TINA's USM and SSM into application-specific and generic logical parts for different feature sets.

2.3.1 TINA service features

The TINA session model defines a number of feature sets and the interfaces they work on. A number of feature set descriptions and what they are dependant on is tabulated in table 2.1

The *BasicFS* is suitable for single party calls and only allows the application to end or suspend the call. The *multipartyFS* is suitable for calls with two or more parties whilst the *multipartyIndFS* allows a party to communicate the operation it invoked to other parties in the session. The *VotingFS* allows parties to vote on the decisions to be taken in a session, whilst the *ParticipantSRBS* and *ParticipanSBIndFS* allows applications to manage their streams. Table 2.1 contains the feature set, allowed operation, feature sets dependency and computational objects this feature sets apply to. Feature sets dependency works by allowing simpler feature sets usage as the basis for more sophisticated features sets, for example a *multipartyFS* depends on the *BasicFS*.

The level of abstraction provided by TINA service feature is close to what we want to achieve but is not high enough because:

- one still needs to keep information on the interface identity i.e whether a USM or an SSM interface object and their corresponding object reference.
- TINA's FS can only be mapped (super imposed) to Parlay call control SCFs because TINA does not cater for the other SCFs provided in the Parlay gateway. Parlay call control SCF mapping to TINA FS exists because TINA service architecture uses TINA FSs as a means of specifying capability levels supported by different service components, whilst Parlay achieves a similar concept by splitting the call control SCF into GCC, MPCC, MMCC and CCC.

We therefore use TINA FSs as guidelines in providing a higher level of abstraction. The level of abstraction sought after is higher than that of TINA FSs, since we seek to provide all methods on a single simple interface. Communicating with a

Table 2.1: TINA Feature sets and allowed operation

Feature Set (FS)	Allowed operation	Dependent On	Computation Object
BasicFS	*end* *suspend*	*Mandatory*	SSM
MultipartyFS	*create* *suspend* *modify* *end*	*BasicFS*	SSM USM
MultipartyIndFS	*create* *suspend* *modify* *end* *resume*	*MultipartyFS*	USM
VotingFS	*create* *suspend* *modify* *end*	*MultipartyIndFS*	SSM USM
ParticipantSBFS	*create* *suspend* *modify* *end*	*BasicFS*	SSM USM
ParticipantSBIndFS	*create* *suspend* *modify* *end* *resume*	*ParticipantSBFS*	USM

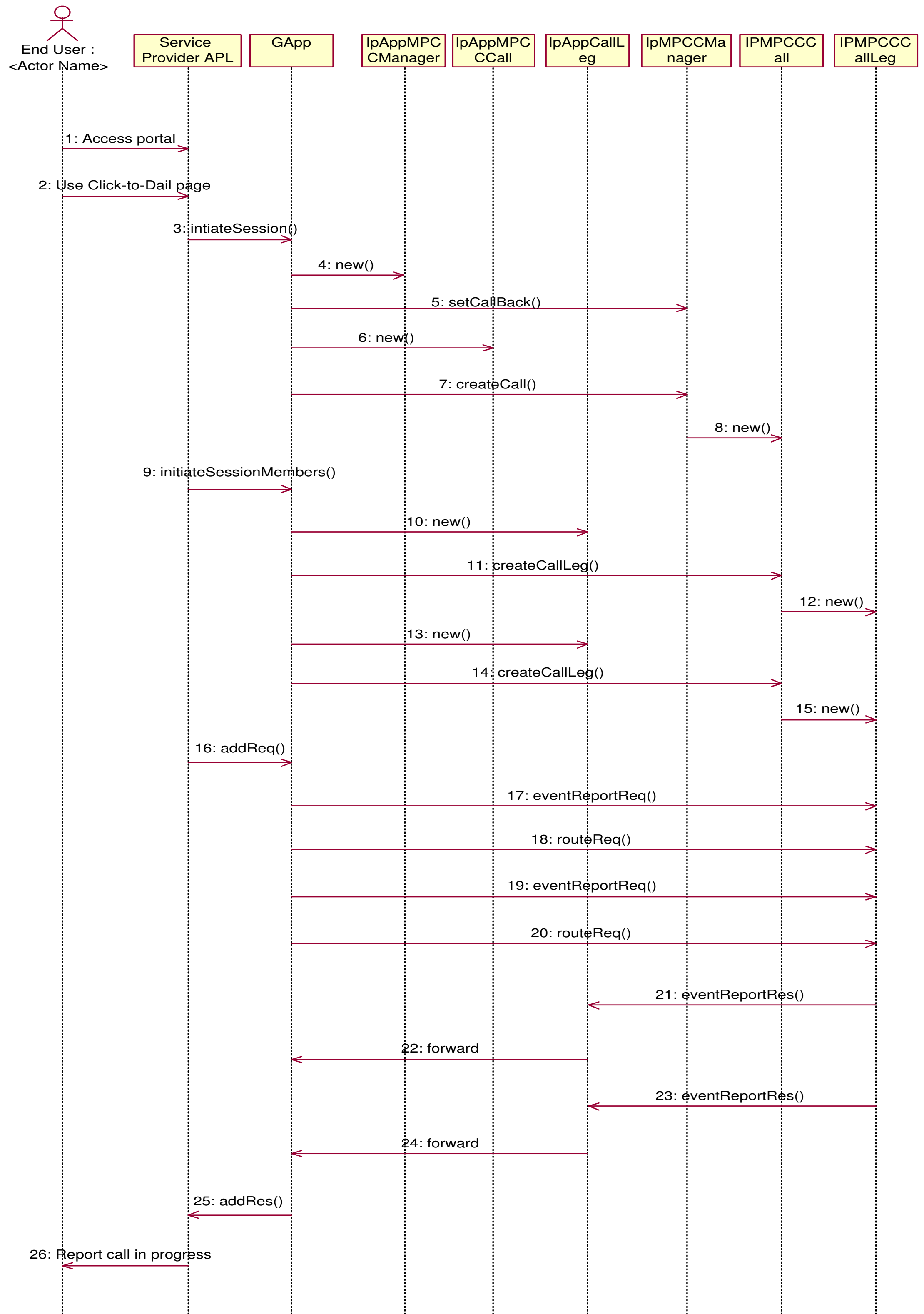

Figure 2.7: Click-to-Dial implementation on proposed GApp interface

single simple interface enables abstraction from the identity of the SCF objects that implement a service. Having this in mind the Click-to-Dial example is illustrated in figure 2.7 using TINA-like FSs with the components identity abstracted. In figure 2.7 GApp stands for an interface where TINA-like FS operations are triggered, the *initiateSession()* method serves the same purpose as a *create()* invoked on a SSM *MultiPartyFS* interface, the *initiateSessionMember()* method is equivalent to a *create()* invoked on a USM *MultiPartyFS* interface and the *addReq()* method is equivalent to a *create()* invoked on a *ParticipantSBFS* interface. On invocation of the Click-to-Dial application, the *initiateSession()* method is invoked on the GApp interface. This is achieved on the Parlay gateway through messages 4-7. Thereafter the *initiateSessionMember()* method is invoked to create objects required for a two party call (9). The user then adds media connection through the *addReq()* method in 16. The response is returned in 25, which is then forwarded to the end user as call in progress (26).

The abstraction provided here still leaves the application programmer with control as he/she can still perform operations on each party in the call, for example we can add and suspend parties. This and other functionality are elaborated on in chapter 4.

TINA provides this research with the logical groupings required to provide an abstracted interface and how this logic can be extended from simple to a more complex interfaces. Functions obtained from TINA include *create(), resume(), suspend()* and *end()*. These functions are used as guidelines when searching for reusable blocks in the Parlay specification. TINA is used as it provides facilities for call services. Starting from its abstracted interfaces (service features) as guide line, we could provide a simple call by going through GCC specification; thereafter move on to provisioning more complex services.

2.4 Chapter Summary

This chapter gives an overview of Parlay by providing its objectives and detailing the architectural view in term of the computational division. It then details Parlay X as a method of providing a higher level of abstraction for the Parlay gateway and finally examines the Telecommunication Information Networking Architecture

(TINA) and its service features as a possible contributor to an API of greater abstraction.

Chapter 3

Architecture and Design Principles

This chapter formally states the research question and details the architecture and design principle used to create the Generic Application Programming (GApp) interfaces. The architectural details include a structural and session model. The design principles are provided as rules used as guidance for reusable block provisioning and extendibility.

3.1 Research Focus

The number of SCFs and the richness of methods of each SCF make the Parlay API powerful. A resulting problem is that the method calls used to provide services proves complex for developers without telecommunication knowledge. While the SCF contains the detailed control logic, the application must coordinate the sequence of calls on the SCF interface. We hypothesize that there will be recurring logic in the application domain to implement this coordination. This research therefore seeks to improve service development and deployment by introducing a higher layer of generic services that can be accessed by applications through simpler calls. The research question is:

- **Starting with Parlay APIs, how can one create reusable blocks that provide a higher level of abstraction:**
  - **to enable rapid application creation and deployment by Application Service Providers (ASPs)**

- **with the same level of functionality as the Parlay gateway**

In the following section we provide a new architecture that provides a higher level of abstraction on the Parlay gateway through a new programming interface called GApp.

3.2 Architectural Overview

This section gives an overview of the architectural design for the extended (i.e GApp API included) Parlay model. The architectural overview is provided by looking at the interaction of components in the structural model. Figure 3.1 shows the separation proposed in the application layer, with each application having its own Particular Application (PApp) logic and using the GApp logic provided in the GApp layer. As a result, the ASP does not need detailed knowledge of the Parlay APIs, and simply uses the methods provided in the GApp layer.

There are two types of abstraction identified in this research:

- Identity: This is an abstraction brought about by hiding the identity of the Parlay computational objects.
- Logical: This abstraction is brought about by grouping Parlay methods, to provide a new simpler method.

In this research a GApp API is provided to interact with all call control, framework and UI APIs. The analysis for reusable blocks should be extended to cater for the other APIs in the Parlay gateway.

3.2.1 Structural

The structural design was accomplished by dividing the GApp layer into two sub-layers:

- The lower layer handles the complexity of Parlay APIs. In order to communicate with the APIs in a structured form, each Parlay API has a GApp API

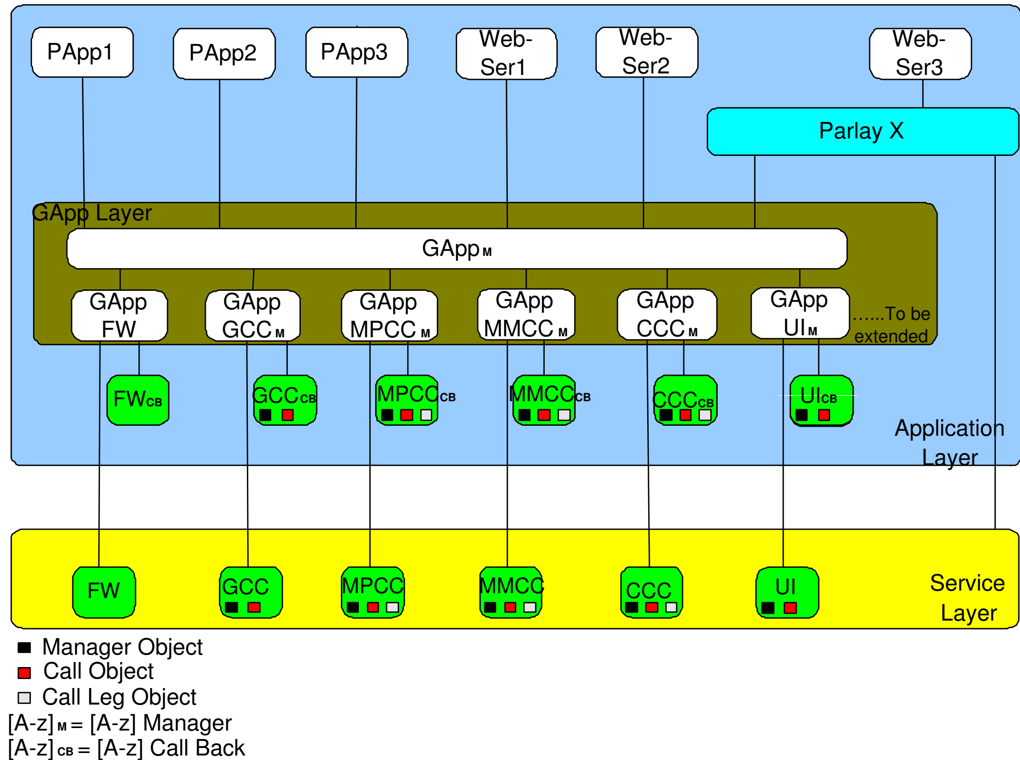

Figure 3.1: Architectural View

manager which manages complexities brought about by it specialization (e.g GAppFW manages the interaction complexity for the Frame Work API).

- The higher layer (GApp$_M$) provides a simplified interface for communication with the ASP. The ASP communicates with the GApp API managers through this layer, hence it hides GApp's structure, therefore providing identity abstraction within the GApp layer.

We aim to provide abstraction by having SCF functions that do not refer explicitly to objects that implement the service session being accessed from the GApp Manager interface; otherwise a function is created in the GApp layer. This function acts as a mediator containing the logical interaction of the SCF objects therefore hiding its complexity from the application.

Figure 3.1 shows the architectural view of each of the components. The GApp layer has seven components which are:

- GApp$_M$ hides GApp structural information from the ASP. It is included in the design to provide a simple interface for the ASP, therefore it provides identity

abstraction for GApp API managers. The rest of the GApp interfaces (e.g GAppFW, GAppGCC$_M$, GAppMPCC$_M$) provide a simple set of interfaces for GApp$_M$ (north bound) and interact with the network operator interfaces and call back through a more complex interface (south bound).

- GAppFW manages creation and removal of services at the ASP's request; also provides service discovery functionalities.
- GAppGCC$_M$ manages interaction with the Generic Call Control (GCC) API. Therefore GAppGCC$_M$ hides GCC interface complexity.
- GAppMPCC$_M$ manages interaction with the Multi-Party Call Control (MPCC) API. Therefore GAppMPCC$_M$ hides MPCC complexity.
- GAppMMCC$_M$ manages interaction with the Multi-Media Call Control (MMCC) API. Therefore GAppMMCC$_M$ hides MMCC complexity.
- GAppCCC$_M$ manages interaction with the Conference Call Control (CCC) API. Therefore GAppCCC$_M$ hides CCC complexity.
- GAppUI$_M$ manages interaction with the User Interaction (UI) API. Therefore GAppUI$_M$ hides UI complexity.

As shown in figure 3.1 the specific application logic that interacts with GApp may be a Particular Application (PApp) logic (written for a specific service), a logic provided for web service (web-ser) application or the Parlay X gateway.

3.2.2 Service Session

This section analyses the active components during a session and provides session identification details. Each of the active members has operations applicable to it. Going through figure 3.2, the Session Member is a collection of participants and is an abstract class [1]. A Session Member can be a party (Call-leg end) or resource. Session Members are associated through calls, whilst the operation that can be accomplished is specified in the control SR. The Control SR also acts as an abstraction provider by providing exception messages to the user at the level required. Therefore it is a part of the GApp$_M$ interface shown in figure 3.1. Figure 3.2 provides a

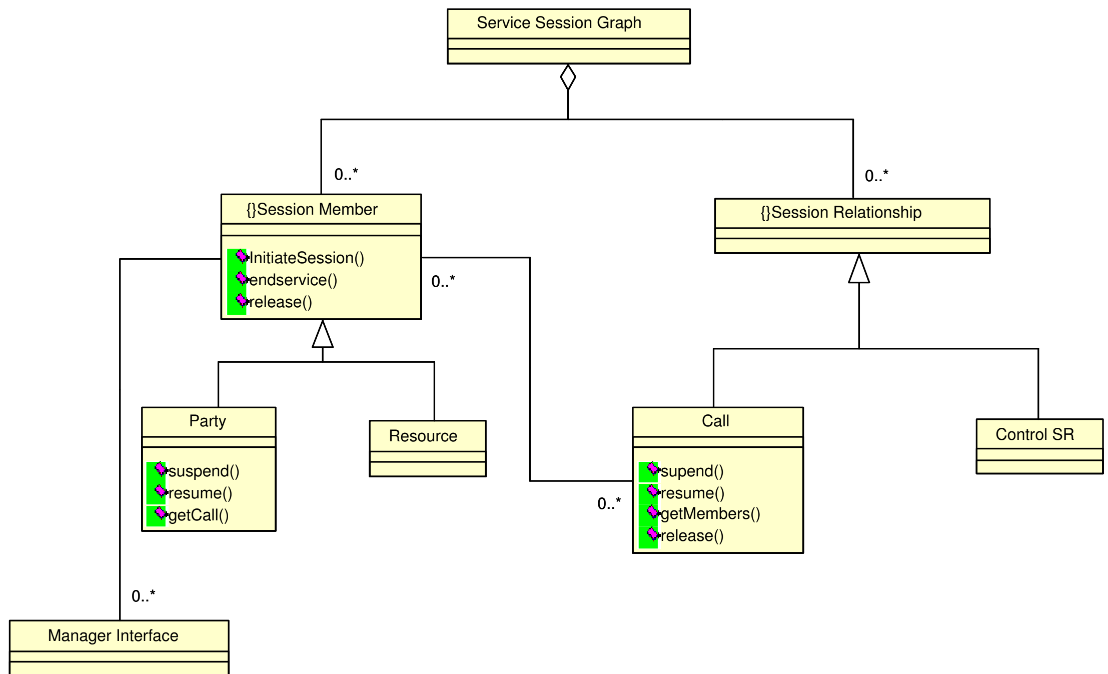

Figure 3.2: Service Session Graph [1]

global view of the interaction between SCF objects and the GApp methods. For example a party (SCF call-leg object) can be suspended, released etc.

Sessions are used to identify relationships between parties. Therefore, relationships such as calls and call-legs are identified through *sessionIDs* using 32 bit integers. Parlay requires that a session ID be "unique within the context of a specific instance of an SCF" [21]. We adopt *sessionID* in GApp interface to uniquely identify each call and call-leg objects in a GApp Manager instance. This is done through structurally naming the call and the corresponding call-legs that belong to the call. The call is identified by a call ID which is a 32 bit integer whilst the call-leg is identified by appending the call ID to the call-leg's (target or originating depending on whether the leg is an originating or terminating leg) E164 number. On invoking a service with the call ID or call ID with the E164 number appended, the ID is mapped by GApp to the appropriate call or call-leg identifier respectively. An identifier is a location and a session ID; the location is used to find the SCF instance that performs the logic and the session ID is used to identify the session.

3.2.3 Overview of GApp use case analysis

A number of generic GApp functions were identified by observing the different stages that occur in the execution of services offered on the Parlay gateway. These stages include:

- Creation of the service:
  1. Obtain the SCF Manager.
  2. Creation of the service instance:
     (a) Enabling the notification points to prompt session
     (b) Application creation of session
- Service Usage:
  1. Detection of the notification point.
  2. Initialization of appropriate interface.
  3. Usage of this interface to perform the required service.
  4. Deletion of the interfaces thereafter.
- Disabling Service:
  1. Removal of the service instance
     (a) Disabling or destruction of the Notification points.
     (b) Deletion of the session.
  2. Deletion of the SCF manager.

We apply these three identified use cases for all GApp manager classes through the help of sequence diagrams in chapter 4.

3.3 The GApp Design Principle

We need to provide reusable logic in a structured and extendable way. To do this we provide a clear way of defining logical blocks that hold the relationships between different SCF. Structural provisioning of the reusable blocks was accomplished by

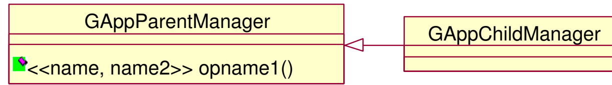

Figure 3.3: Class diagram for GApp manager inheritance

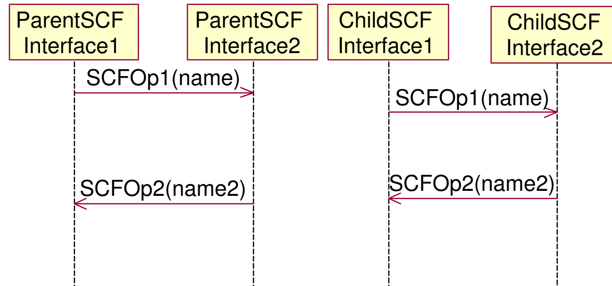

Figure 3.4: Sequence of events used to accomplish $opname1()$ in figure 3.3

analyzing the specification and sequentially combining SCF functions to provide logical blocks (GApp functions) that provide the application developer with powerful functional interfaces.

Extensibility is provided by exploiting the relationships that exist between different SCF interfaces (inheritance) and reproducing the same set of relationships between the GApp API manager classes. We needed a way to transform inheritance relationships between SCF classes to inheritance relationship between GApp API manager Classes. Hence a child GApp API manager inherits all its parent GApp API manager's functions e.g. GAppMMCC will inherit all GAppMPCC functions. This is elaborated in figure 3.3 and 3.4

To enable a structural design, we provide a rule for use when developing new or modified functions for the child GApp API manager classes. A new function is a function that provides a new functionality to the child GApp API manager class relative to the parent, whilst a modified function is one that has the same functionality but the logic to achieve this functionality has being modified from the parent to the child GApp class. The rule states a new or modified function is created in the child GApp API manager class if logical changes occur in comparison to the parent's logical blocks. Logical changes occur when:

- A new SCF interface is introduced in the child SCF class. This is illustrated in figure 3.5 and 3.6.

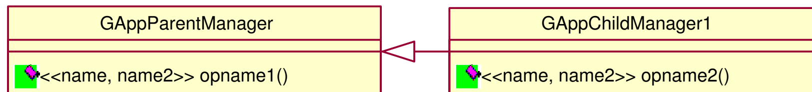

Figure 3.5: Class diagram for a GApp child Manager with a new functional block

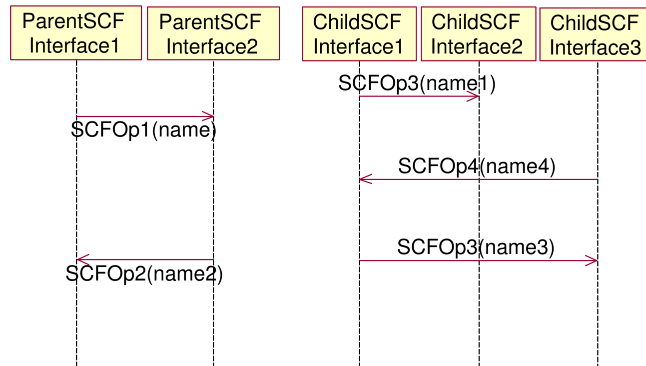

Figure 3.6: Sequence diagrams to accomplish parent and child class methods of figure 3.5 on Parlay SCF

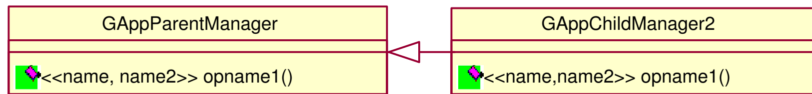

Figure 3.7: Class diagram for a GApp child Manager with a changed functional block

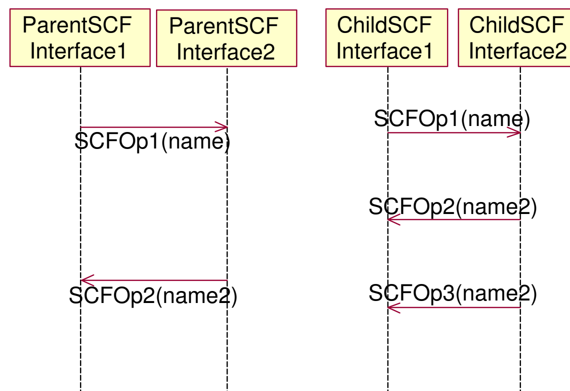

Figure 3.8: Sequence diagrams to accomplish parent and child class methods of figure 3.7 on Parlay SCF

- the method calls made to accomplish a function by the child GApp API manager on the SCF interfaces are different relative to the parent function. This is shown in figures 3.7 and 3.8.

Using this design principle we obtained the GApp class diagram in figure 4.5. In the next chapter (Chapter 4), we show how GApp methods for each GApp class are derived.

3.4 Chapter Summary

This chapter details the research question and principle used to generate the GApp interfaces. This includes rules to provide reusable blocks and extendibility of it. There exists a GApp interface manager for each SCF manager in the server. The structural arrangement of SCF interfaces was used as a model for the GApp interface arrangement. Therefore, inheritance in the SCF interface (e.g., between MPCC and MMCC) translates to inheritance in GApp manager interface (e.g GAppMPCC and GAppMMCC respectively).

Chapter 4

GApp Components Design

This chapter details the design of components GAppFW, GAppGCC, GAppCCC, GAppMPCC, GAppMMCC and GAppUI, through the principles stated in section 3.3.

4.1 GApp Frame Work

The GApp Frame Work (GAppFW) main function is to manage the creation and deletion of services, therefore it provides the ASP with access to:

1. Service list.
2. Service description.
3. A reference to a new SCF manager.
4. Disposition of SCF managers and resources.

This section therefore provides methods to accomplish these functionalities. **The $\mathbf{GApp_M}$ is not included in all sequence diagrams in this chapter for simplicity, but it is the only interface that the application communicates with. Hence the application communicates with other GApp interfaces through $\mathbf{GApp_M}$**. Figure 4.1 shows the methods available in the GAppFW class. To provide a service list we require two functions, the first provides access to the service discovery interface

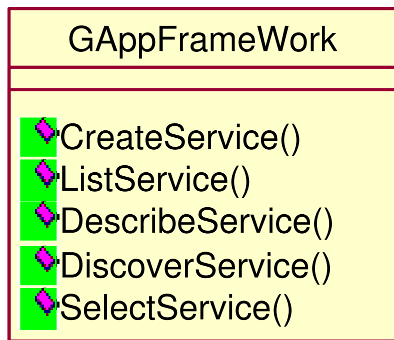

Figure 4.1: GApp Framework class

(*CreateService()*) whilst the other is to list services available on this interface (*List-Service()*). Figure 4.2 shows the way GApp is used to provide a service discovery interface. A service discovery interface is provided through the following process:

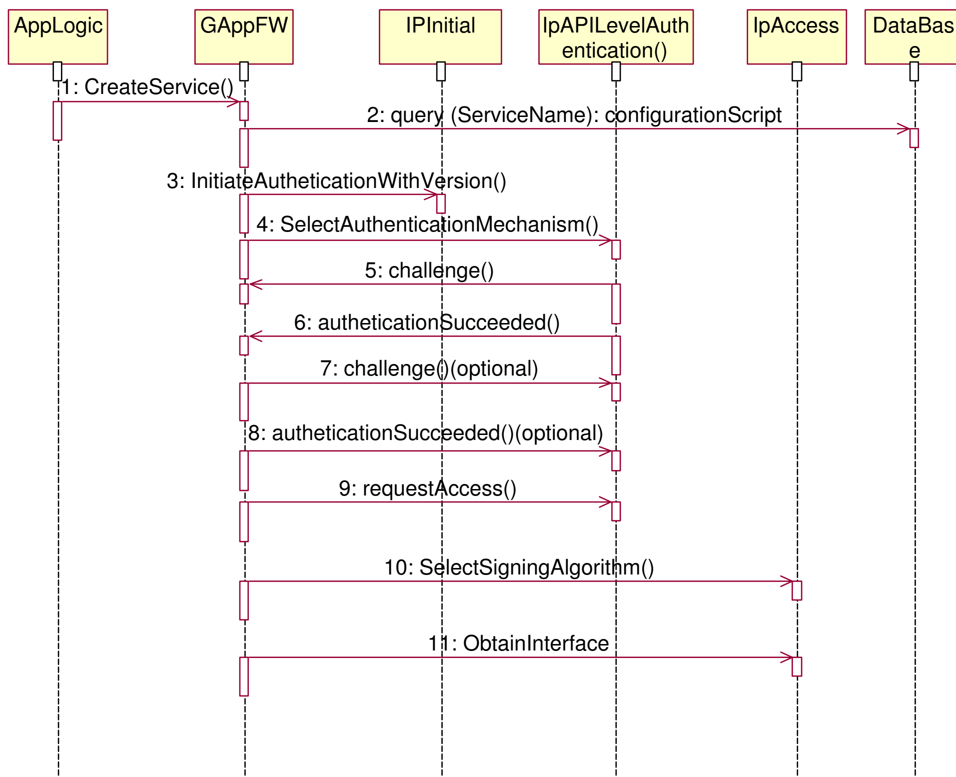

Figure 4.2: Application Initiates Service

- The application logic executes *createService()* (1) on the GAppFW interface. (2) GAppFW then queries the database to obtain the configuration script for

this particular service. The configuration script contains information such as domain ID authentication interface, authentication mechanism, signing algorithm, agreementText etc (These are information specific to the ASP, for the SCF to be obtained).

- GAppFW then triggers *initiateAuthenticationWithVersion()* (3) which initiates authentication with a publicly available framework (through url or orb) by passing the client domain's identifier, the authentication type, the framework version number and returns the Frame Work identifier.

- The next operation invoked is the *selectAutheticationMechanism()* (4) which provides the Frame Work with a list of authentication mechanisms the ASP supports. The Frame Work selects one from the list and passes it back as the return parameter. (5) Depending on the authentication mechanism a number of challenges are invoked on the ASP, which it must respond correctly to using the return parameter. (6) The *authenticationSucceeded()* is then invoked to inform the ASP of its success. $(7, 8)$ Thereafter, depending on the ASP's policy it might authenticate the Frame Work with a similar process.

- The ASP then invokes the *requestAccess()* (9) which returns the reference to the Frame Work access interface. The Frame Work access interface provides access to other Frame Work interfaces. (10) Before the client can use the access interface it must provide a list of all signing algorithms it supports for use in cases where digital signature is required through the operation *select-SigningAlgorithm()*. The Frame Work selects one from the list and passes it back as a return parameter. (11) The client then invokes *obtainInterface()* which returns a reference to the required Frame Work interface, in this case the service discovery interface, which is described in the following section.

In figure 4.2 message 3-11 is the normal sequence that was accomplished by the ASP previously but know is provisioned by GAppFW through a simple call to *createService()*.

4.1.1 Service Discovery

The Service Discovery interface is used by the application to obtain a *serviceID* for the SCF of interest as shown in figure 4.3. This ID is obtained by performing a number of operations, which might include obtaining a list of available SCFs through *ListServices()* $(1, 2)$ (usually a list of service names "P_MPCC") and asking the framework to describe the service types through *DescribeService()* $(3, 4)$ which return properties of the services of interest. $(5, 6)$ Thereafter the application fine tune these properties to reflect its requirement and passes it to the framework through *DiscoverService()*. The frame work then returns a list of *serviceID* matching the needs put forward by the application.

These three operations $(2, 4, 6)$ need to be performed by the application as they provide functionality to the application logic as a result they are already at an appropriate level of abstraction i.e the granularity provided by the Parlay gateway for these functions is appropriate.

syncronous messages
Optional Process only required if user does not already have required infomation such as serviceID, list etc.
All parameters.
listServiceTypes () : TpServiceTypeNameList
describeServiceType (name : in TpServiceTypeName) : TpServiceTypeDescription
discoverService (serviceTypeName : in TpServiceTypeName, desiredPropertyList : in TpServicePropertyList, max : in TpInt32) : TpServiceList
listSubscribedServices () : TpServiceList

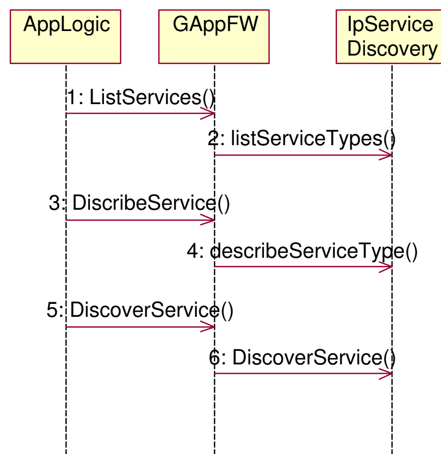

Figure 4.3: Obtaining Service Information

4.1.2 Service Selection

The application selects the required service by passing the service ID as shown in figure 4.4. GAppFW then handles the service agreement signing, between the application domain and the framework. As a result, a reference to the SCF manager is returned to GAppFW in message sequence 5. GAppFW then creates a GApp API manager (e.g GAppMPCC$_{M}$, GAppCCC$_{M}$ etc) to manage the SCF manager by querying the database to obtain a list of the appropriate interfaces and required logic. A *ManagerID* is then assigned to the newly created GApp API manager. The identity (reference and the *ManagerID*) is returned to GApp$_{M}$, which returns a *ManagerID* to the application.

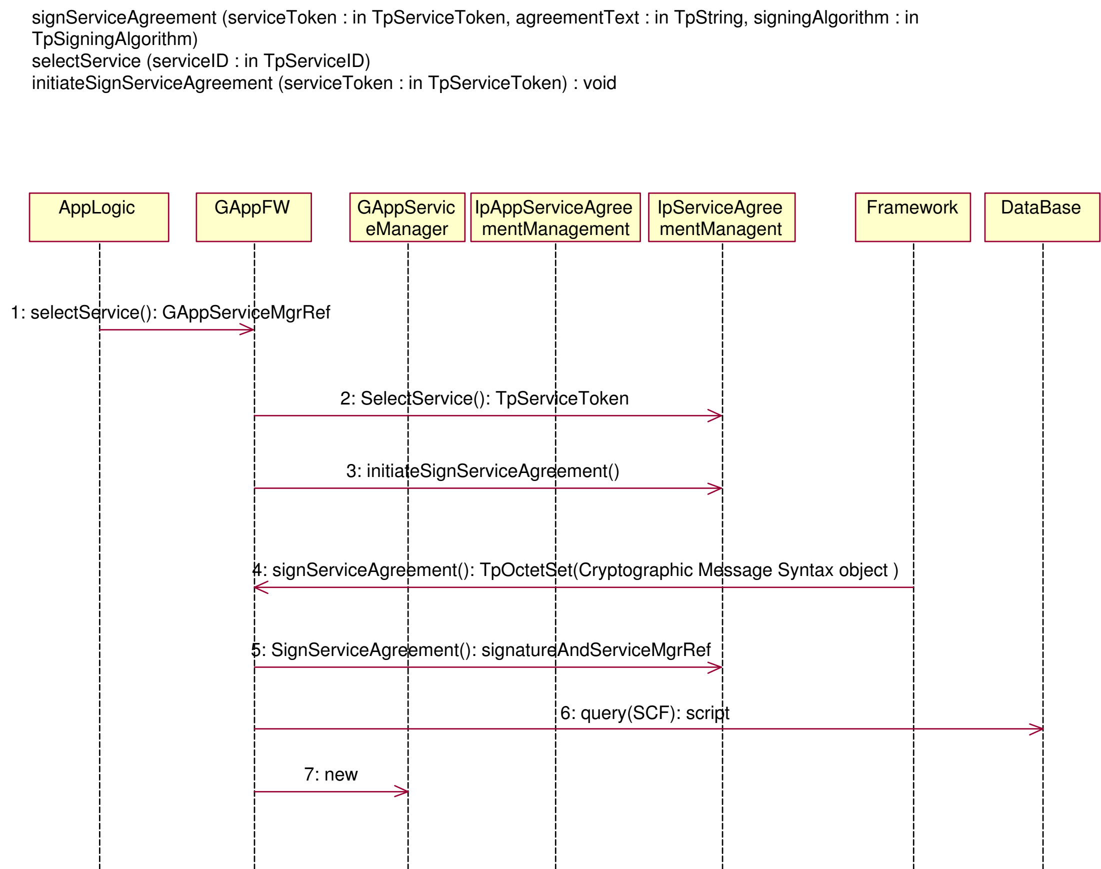

Figure 4.4: Selecting Service

GApp API manager contains translation of higher level methods provided on its interface to methods that are executed on the SCF and vice versa. Therefore when application logic executes an operation on GApp$_{M}$, it is forwarded to the appropriate

GApp API manager interface which checks its list of operation and executes the logic for that operation. The logic of the operation is explained in the form of sequence diagrams in this chapter. A GApp API manager exists for each service manager obtainable in Parlay. Therefore there is a one to one relationship between the GApp API managers and the service managers in the SCS.

4.2 GApp API Manager Classes

Firstly we provide a general view of Parlay service logic by discussing GAppServiceManager. As shown in figure 4.5, GAppServiceManager is an abstract class that forms the bases for all GApp API manager Classes. It contains methods that all GApp API manager Classes should possess. This was obtained by studying GCC (Work on GCC has been discontinued), MPCC, MMCC, CCC and UI. GCC is dealt with in this research to provide a general view of the mechanism used to deploy services in Parlay. It was identified that the logic used for service deployment in GCC is the same with the newly enhanced SCF (e.g MPCC), but some of the methods have been further enhanced for example *enableCallnotification()* is now implemented in MPCC by *createNotification()* and *enableNotification()* whilst *routeReq()* is now provided through *eventReportReg()* and *routeReq()*. The study of GCC provides us with the general concept of the logical groupings of reusable blocks that can be obtained in Parlay. As a result, the design is robust as it is easily comprehensible and extendable [22, 23, 21, 24, 25].

Figure 4.5 shows the overall view and relationship between GApp API manager Classes. This was modeled using the relationship between GCC, MPCC, MMCC, CCC and UI. The relationship is such that CCC inherits from MMCC which inherits from MPCC. Therefore as the Parlay call control interfaces get more complicated (through provisioning of more functionality) we can follow the inheritance structure to structurally provide enhancement to the reusable logic obtained. As a result, we obtain a basic set of reusable blocks for the base class and extend these blocks to cater for the functionality provided by the more complex Parlay interfaces. Therefore the complexity brought about in the child SCF class is handled by extension of the logic from the parent GApp class and the creation of new logic in the child GApp class as required.

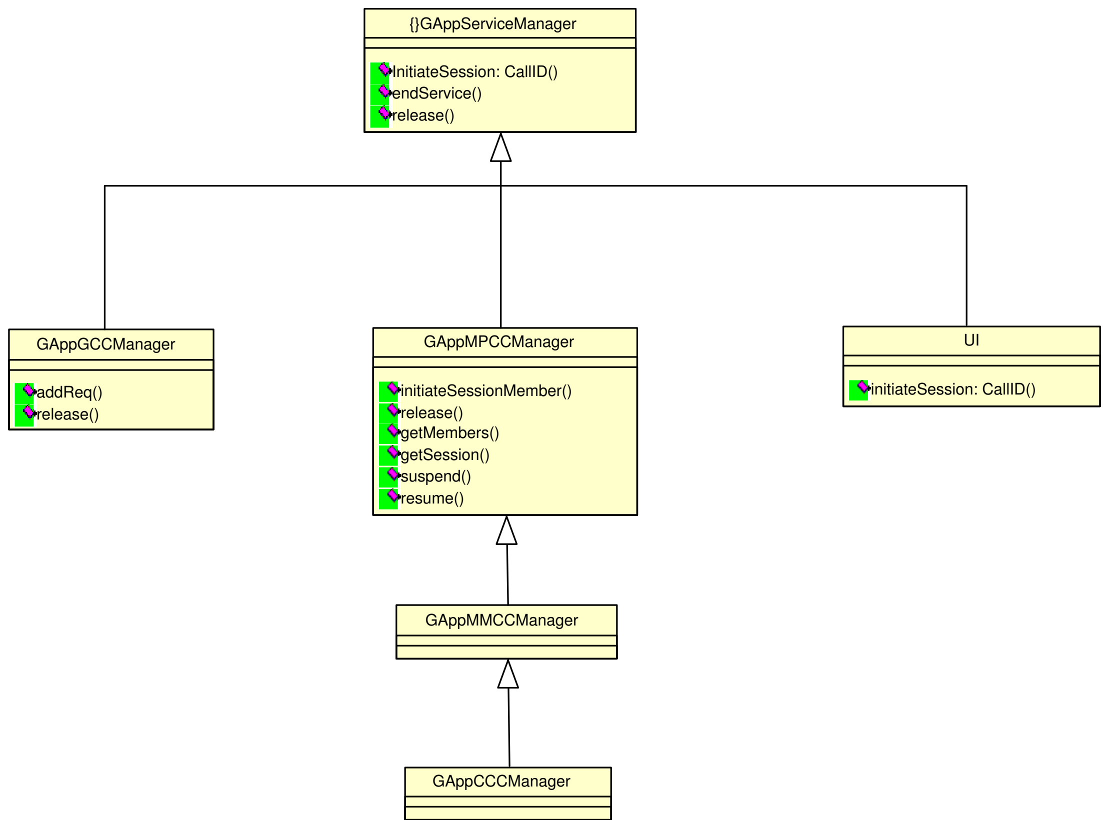

Figure 4.5: GApp Call manager inheritance Structure

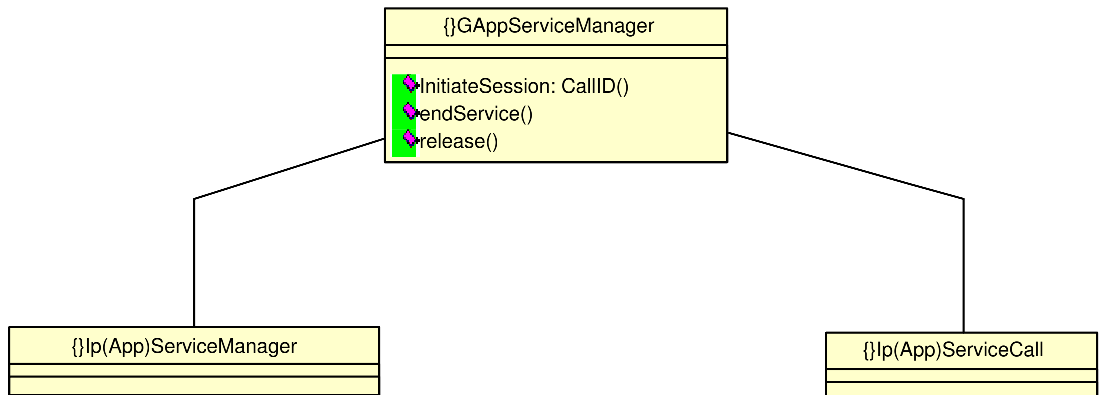

Figure 4.6: GApp service relationships

The functions in the *GAppServiceManager* are the functions which are common to all GApp API managers, these functions are used for object initialization and removal similar to TINA's create and delete.

Figure 4.6 shows the relationship between abstract classes. *GAppServiceManager*

is an abstraction of the GApp API managers whilst *Ip(App)ServiceManager* and *Ip(App)ServiceCall* are abstractions of SCF managers and call classes with the callbacks taken into consideration by indicating App in braces (make the diagram neater). Moving onto the sequence diagram we try to follow the principles given in section 3.3. Firstly we look at initiation of service i.e how an application gains control of a call.

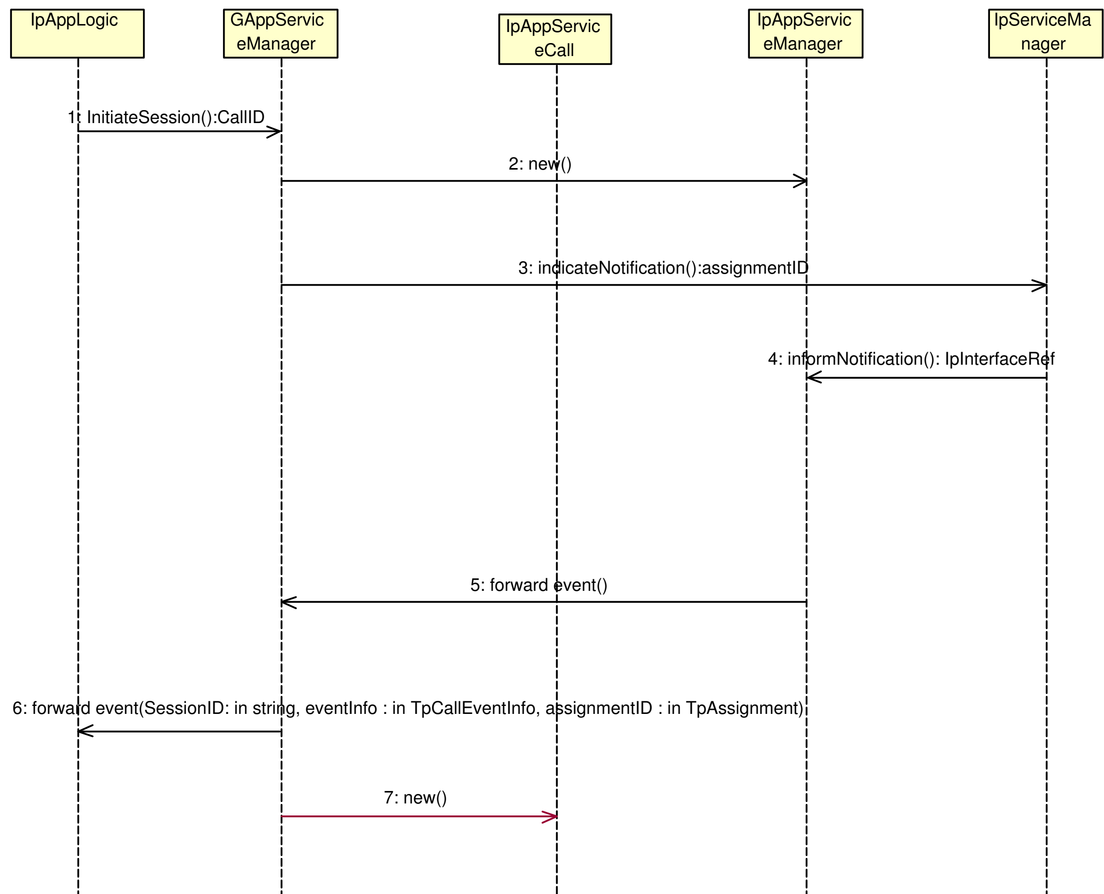

Figure 4.7: Application Gains control of the Call through notification

One way of starting an application service is by obtaining notification of a call which satisfies the required criteria, as a result being passed the call. This is done by creating or enabling notification points in interrupt mode. This process is shown in figure 4.7. (1) Application Logic (APL) executes initiateSession() on the *GAppServiceManager* interface. *GAppServiceManager* then triggers the appropriate logic for this service (2, 3, 6, 7). (2) It creates a callback *IpAppServiceManager* interface

with the operation new() which returns the object reference. (3) The object reference is then passed to the SCF service Manager object through the *indicateNotification()* (*createNotification()* or *enable(Call)Notification()*). *indicateNotification()* specifies the criteria for the application to be triggered and returns an assignment ID. The *asssignmentID* enables correlation between the *indicateNotification()* and *informNotification()*(*reportNotification()* or *callNotify()*). (4) An event occurs satisfying the criteria specified and *informNotification()* is executed by the *IpServiceManager* on the *IpAppServiceManager* passing the call object identifier of the call satisfying the criteria specified as an input parameter. (5, 6) *informNotification()* is then forwarded to *GAppServiceManager*, which forwards it to GAppM which then forwards it to *IpAppLogic* at the right level of abstraction (not shown in sequence diagram). (7) *GAppServiceManager* also triggers the creation of a call back interface for the call object identifier passed to it. The call back object reference is passed back to the *IpServiceManager* through the return parameter of *informNotification()*.

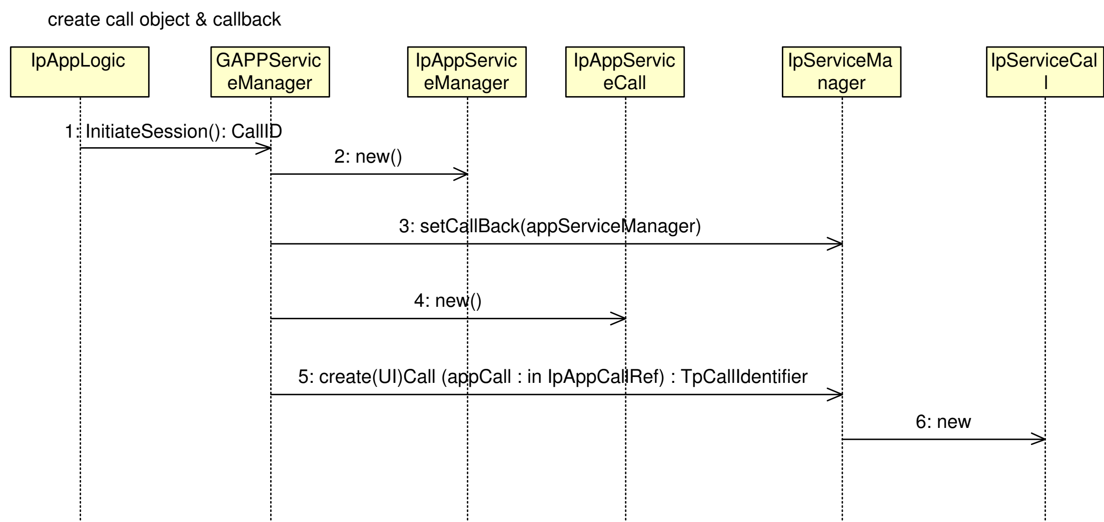

Figure 4.8: Application creates a call object

Another way of initiating an application service is by creating the call. Figure 4.8 shows the APL starting a session by executing (1) *initiateSession()* on the *GAppServiceManager* interface which then triggers the following logic. (2, 3) It creates a callback object *IpAppServiceManager* and forwards the reference to *IpServiceManager* through the method *setCallback()*. (4) Thereafter it creates another call back object *IpAppServiceCall* to handle the session. (5) Forwards this object interface to *IpServiceManager* through the *create(UI)Call()* operation which create a

new *IpServiceCall* object.

In general in this research multiple functions with the same name (e.g *initiateSession()*) within a class are resolved through polymorphism as the function will use different parametric sets.

release removes call related objects and network connections whilst infomation concerning the call can still be sent to the callback interfaces. deassign releases the application from controlling the call whilst it goes on in the network. the call related info requested is discarded

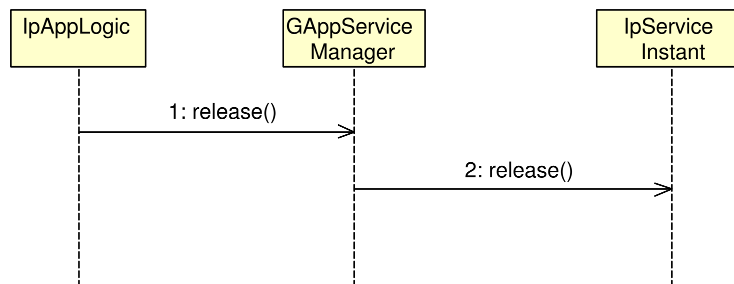

Figure 4.9: Application releases Call

Figure 4.9 shows the application releasing a call through *release()* which triggers release.

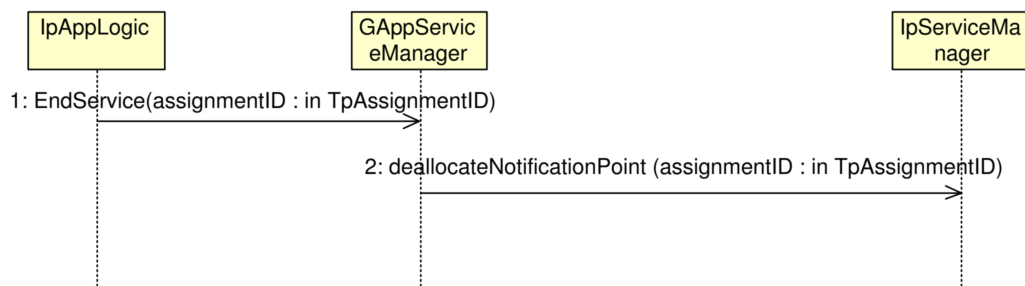

Figure 4.10: Application ends service

Figure 4.10 shows how a service is disabled using *Endservice* on a *GAppServiceManager* interface. GAppServiceManager performs the request by triggering *deallocateNotificationPoint* (actually translates to *disable(destroy)Notification* depending on whether *enable(create)Notification* was used in creating the service).

Figure 4.11 shows sequence of events which apply to services that can be accomplished on the SCF interfaces asynchronously; where the level of abstraction provided by Parlay is already adequate.

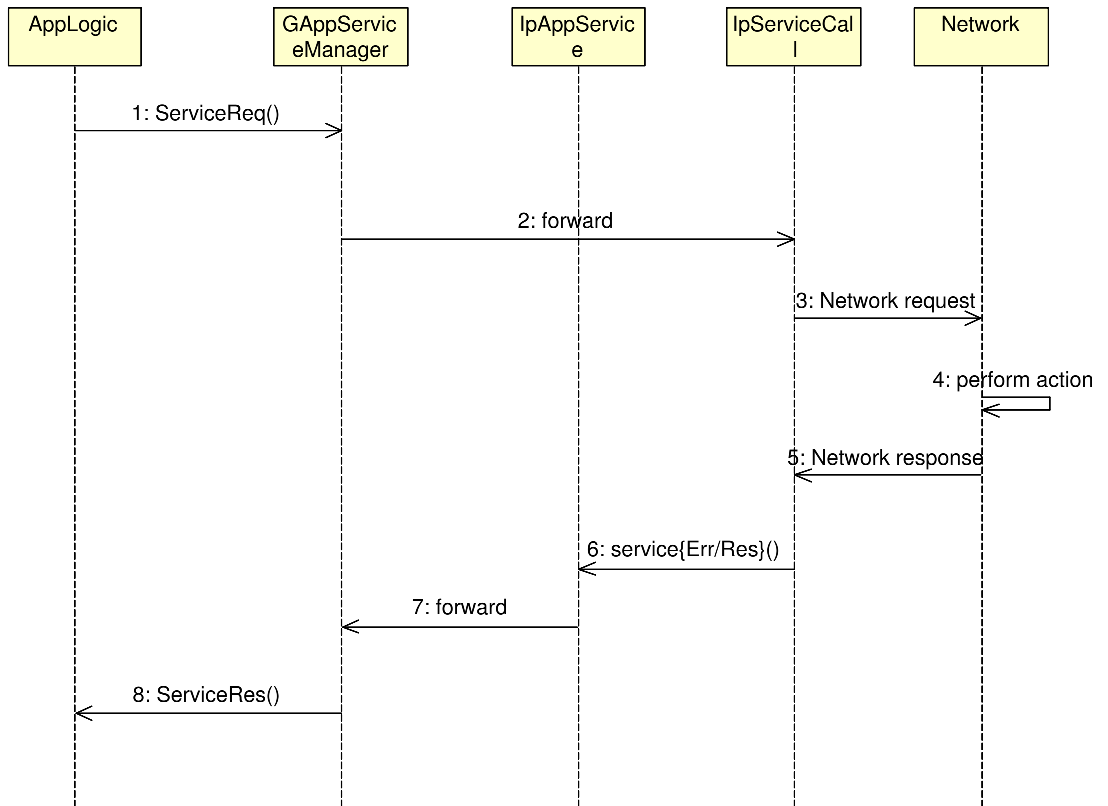

Figure 4.11: Asynchronous Methods

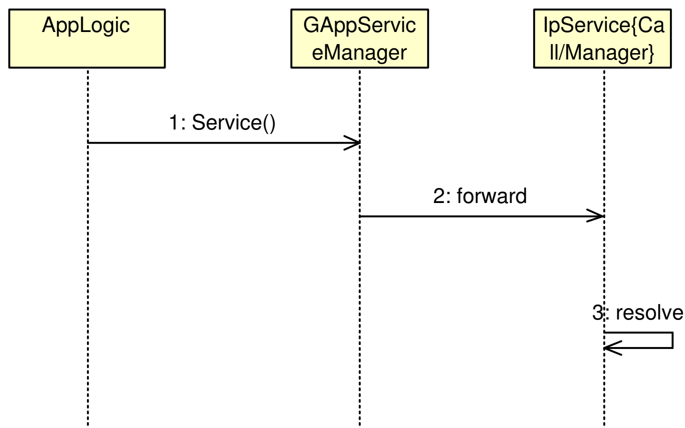

Figure 4.12: Synchronous methods on the SCF interface

Figure 4.12 shows sequence of events which apply to services that can be accomplished on the SCF interface synchronously; where the level of abstraction provided by Parlay is already adequate.

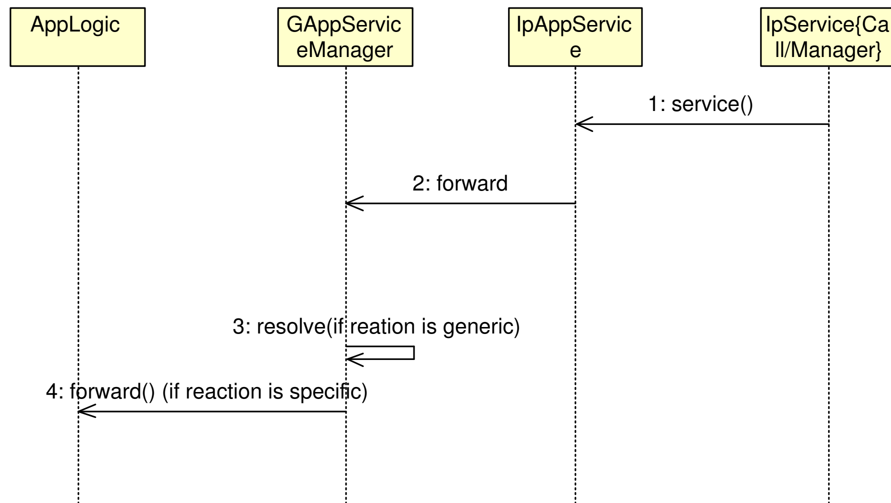

Figure 4.13: Synchronous methods on the Application call back interface

Figure 4.13 shows sequence of events which apply to services that can be accomplished on the callback interface synchronously, where the level of abstraction provided by Parlay is already adequate.

In this research identity based abstraction is achieved in all cases (including synchronous and asynchronous methods with adequate level of abstraction). This is true as the user only needs knowledge of the $GApp_M$ class, whilst logical abstraction is achieved in all case apart from synchronous and asynchronous methods with adequate level of abstraction.

4.2.1 GApp GCC Manager

$GAppGCC_M$ allows calls to be set up by the application through the *initiateSession()* (shown in Appendix A) method on its interface and if the call is already setup, the application can gain control of the call through the same methods but with different parameters [21]. Each of these sequences returns a call object reference to the application. In the second scenario (i.e. call already setup) the Application Logic (APL) has to use the *routeReq()* method on the call object to indicate interested in other events during the context of this particular call. We explain the reusable logic provided in $GAppGCC_M$ by building on the logical reusable blocks that are provided in section 4.2. Logical blocks are reused by replacing abstract interfaces and there methods with GCC specific methods and interfaces. Appendix A shows

logical blocks that map directly from the GAppServiceManager to GAppGCC$_M$. Direct Mapping means that the sequence of events to accomplish the logic is the same (e.g *initiateSession()*). In this section we only go through logical blocks that are not direct maps and new logical blocks that are created due to the specialization of the GCC relative to the abstract service classes in section 4.2. Extension to the logic provided is shown in:

1. Figure 4.14.

   The GCC SCF *IPcall* object triggers *callEnded()* on the application callback which signals call ending in the network (e.g user hangs up). This is forwarded to the GAppGCC$_M$ which then issues a release on the call object to free resources used for the call. The application in addition to invoking release can also relinquish control of the call through *releaseControl()*.

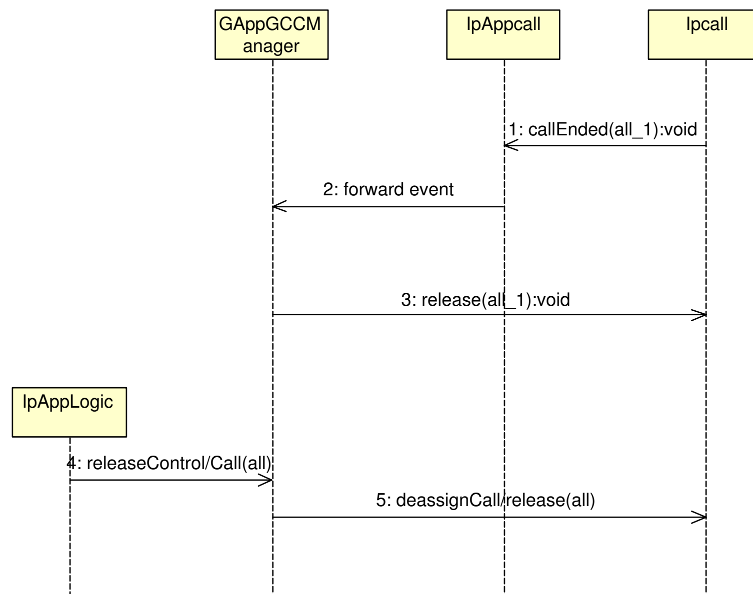

Figure 4.14: Application releases call

2. Figure 4.15

   A completely new logical block is introduced to accomplish the *addReq()* method in figure 4.15. The logic takes the form of an asynchronous request method in figure 4.11 but differs since it provides a higher level of abstraction. Abstraction is provided since the application only provides one or two (if the application is setting up the call) target addresses. *addReq()* then adds the parties to the call by routing to the required destinations.

routeReq (callSessionID : in TpSessionID, responseRequested : in TpCallReportRequestSet, targetAddress: in TpAddress, originatingAddress : in TpAddress, originalDestinationAddress : in TpAddress, redirectingAddress : in TpAddress, appInfo : in TpCallAppInfoSet) : TpSessionID
routeRes (callSessionID : in TpSessionID, eventReport : in TpCallReport, callLegSessionID : in TpSessionID) : void
routeErr (callSessionID : in TpSessionID, errorIndication : in TpCallError, callLegSessionID : in TpSessionID) : void

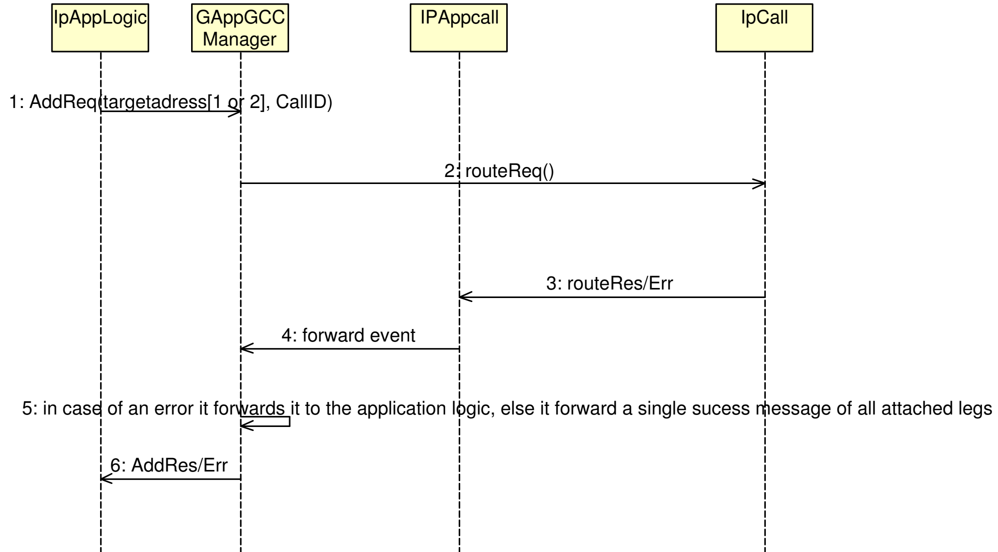

Figure 4.15: Application adds party

4.2.2 GApp MPCC Manager

In this section we provide details of GAppMPCC$_M$ design [22]. As in section 4.2.1 we will only go through the changes brought through specialization. The reusable blocks that are direct maps of the *GAppServiceManager* are given in appendix B.

If an application obtains a call through notification, it has to restart the existing call legs on the call (since their processing was interrupted). This is accomplished as shown in figure 4.16. The application invokes *activateSession()* with the *callID* or the *userID[]* as input parameters. The former activates all existing call legs associated with the *callID* by invoking *continueProcessing()* a number of times equal to the number of call legs. In the later scenario GAppMPCC$_M$ converts the user IDs obtained from the *userID[]* array to the appropriate session leg identifier (which is known since this method can only be invoked after a notification, with notification returning parameters call and call leg identifiers to GAppMPCC$_M$).

Figure 4.17 shows two logical blocks which are:

continueProcessing (callLegSessionID : in TpSessionID) : void

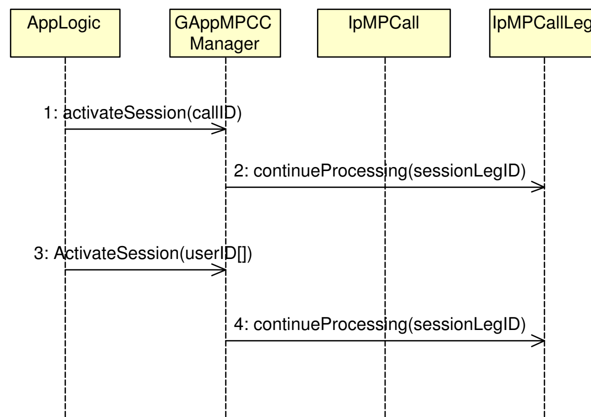

Figure 4.16: Application starts existing session members

- *initiateSessionMember:* This logical block is an enhancement to *initiateSession()* since there are new objects that require creation in the MPCC SCF. The new object is the call leg as a result *initiateSessionMember()* is used to create one or more call legs. *initiateSessionMember* is invoked on GAppMPCC$_M$ with input parameter that signify the target address(es) in an array and the *callID* (for the call to add the users to). If an error occurs during the creation of a leg it is forwarded to the APL.

- *addReq:* This is a logical block that allows the addition of one or more parties to the call. The user invokes *addReq()* on GAppMPCC$_M$ which then performs action 6 and 7 one or more times depending on the number of user IDs given as input parameter with addReq(). If an error occurs during the routing of a leg it is forwarded to the APL.

Figure 4.18 shows *IpMultiPartycall* and *IpMultiPartyCallLeg* triggering *callEnded()* and *callLegEnded()* respectively on the application callback interfaces. This signals call ending in the network (e.g user hangs up) which is forwarded to the GAppMPCC$_M$. GAppMPCC$_M$ then issues a release on the call Leg and call object to free resources used for the call. The application can also relinquish control of the call through *releaseControl()*.

To obtain members participating in a giving call session, *getMemebers()* is invoked on the GAppMPCC$_M$ as shown in figure 4.19. The method returns the list of the

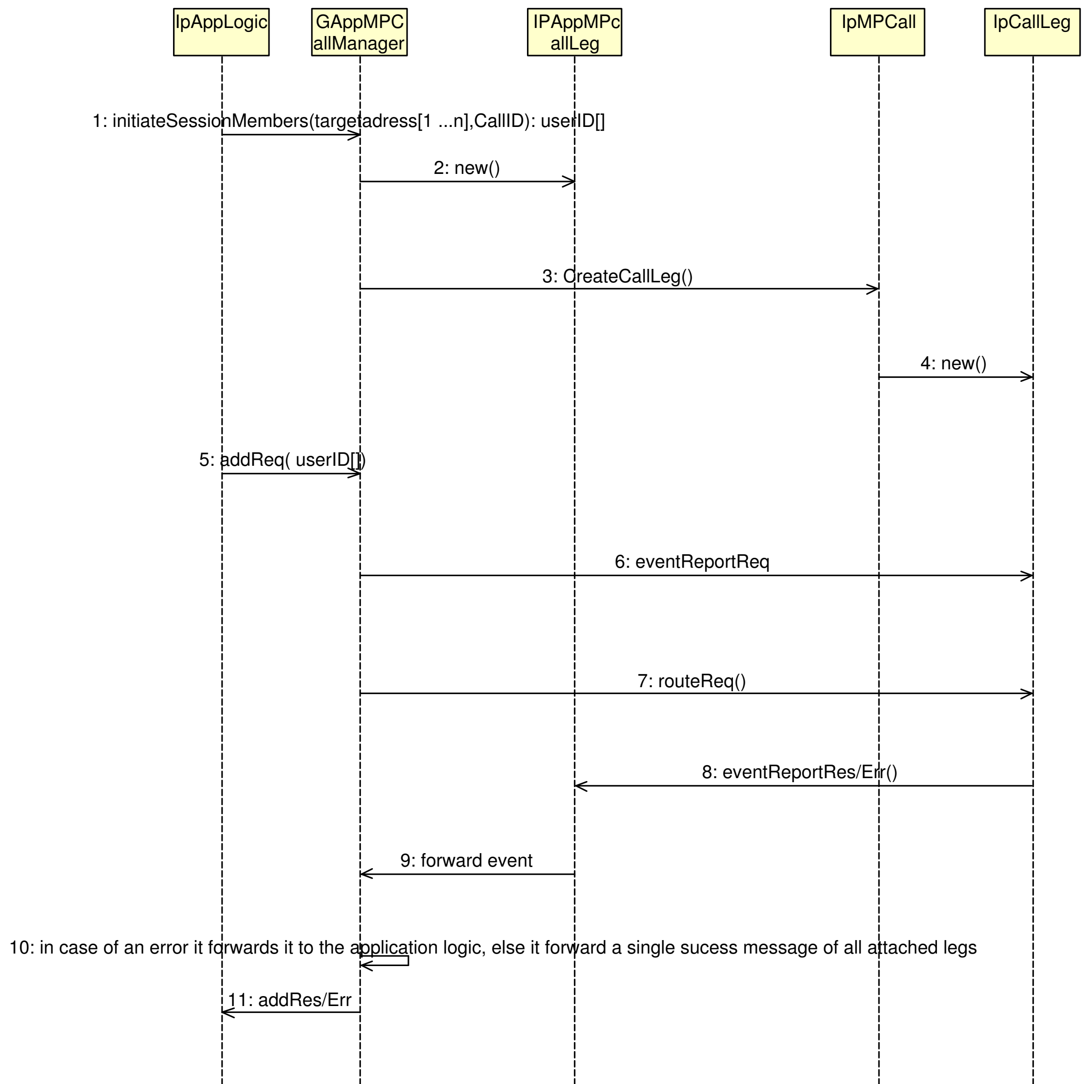

Figure 4.17: Application adds new members to call object

release (callSessionID : in TpSessionID, cause : in TpReleaseCause) : void
deassignCall (callSessionID : in TpSessionID) : void
callEnded (callSessionID : in TpSessionID, report : in TpCallEndedReport) : void
release (callLegSessionID : in TpSessionID, cause : in TpReleaseCause) : void
deassign (callLegSessionID : in TpSessionID) : void
callLegEnded (callLegSessionID : in TpSessionID, cause : in TpReleaseCause) : void

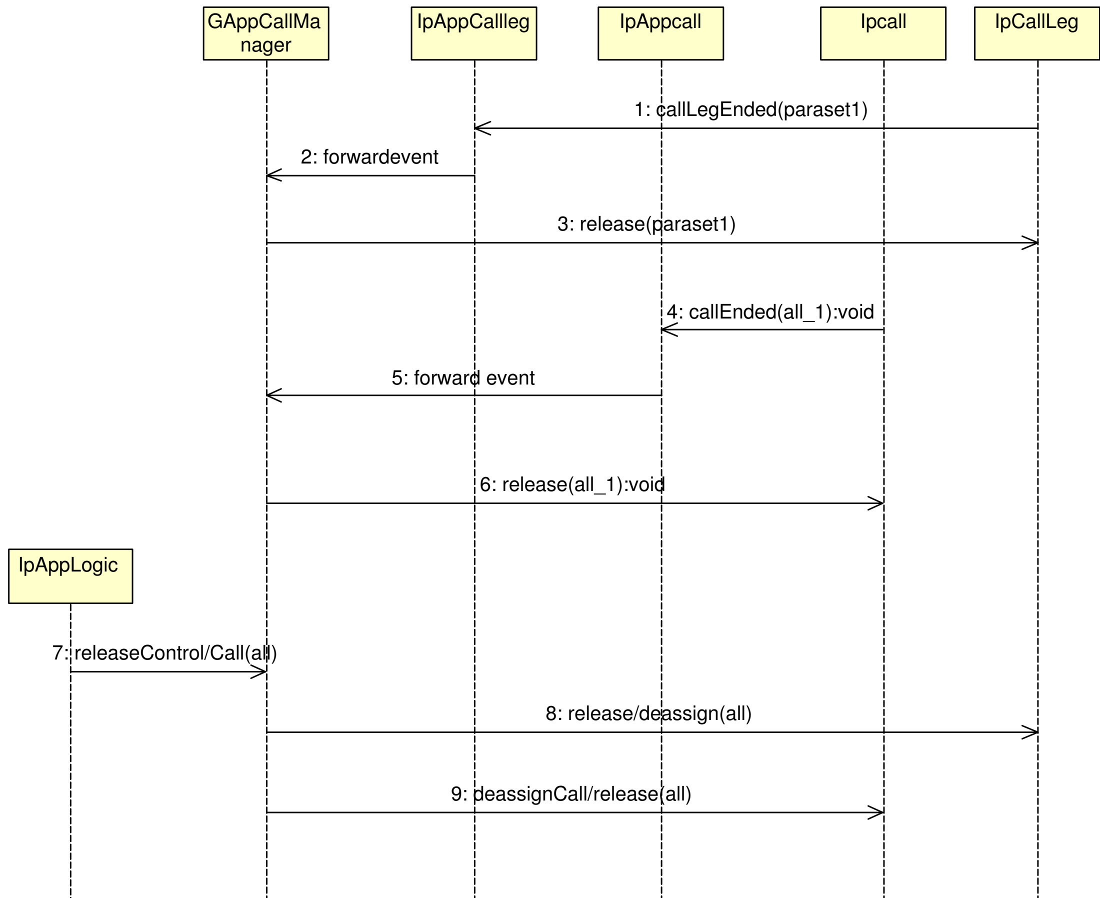

Figure 4.18: Application releases call

users by returning an array with user IDs in the sequence of creation.

getSession() returns the *callID* for all active session in MPCC SCS when called with no input parameters as shown in figure 4.20. In the event of it being called with a user ID array *userID[]* it returns a call ID array *callID[]* matching the sequence of user IDs in *userID[]*. Action 2 in the figure may be performed a number of times depending on the number of user IDs supplied.

Figure 4.21 shows how an application can suspend (resume) the participation of a member(s) from the call. There are two ways to accomplish this:

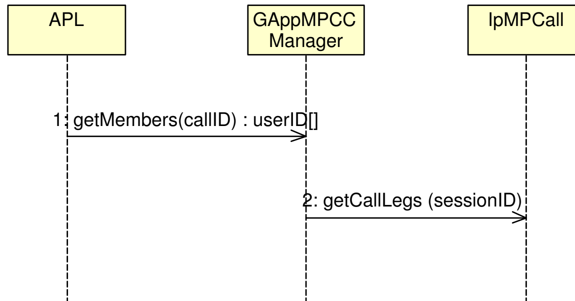

Figure 4.19: Application obtain member's information

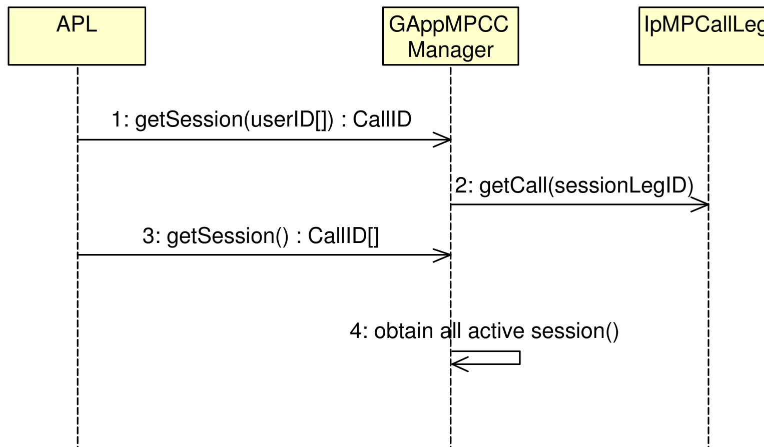

Figure 4.20: Application obtain session information

- the user can invoke *suspend(resume)Req()* with the *callID* (which implies suspension (resumption) of the call as a whole)
- the user can invoke *suspend(resume)Req()* with the *userID[]* (which suspends(resumes) the users specified in the array).

The first scenario is accomplished by obtaining all the call Legs related to the call and detaching (attaching) them, which implies action 3 is invoked a number of times equal to the number of call legs. In the later scenario GAppMPCC$_M$ performs the translation of user IDs to *sessionLegIDs* and the rest of the sequence (9-13) is the same as actions (3-7).

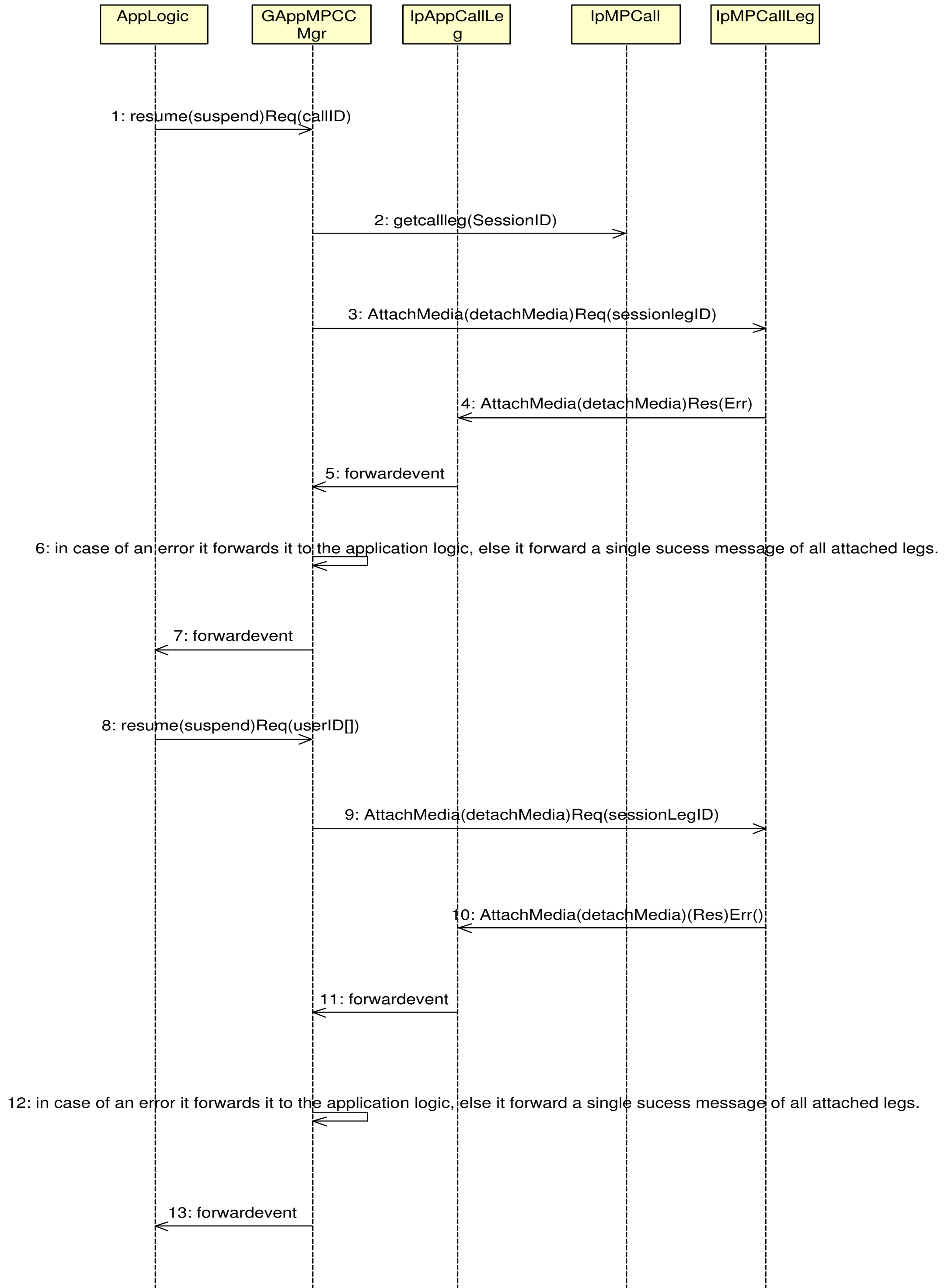

Figure 4.21: Application suspend or resumes call

4.2.3 GApp MMCC Manager

The MMCC does not add additional interfaces to the call control logic. This is because the new interfaces provided in MMCC are interfaces that enable applications with media logic. Media logic is the process required to provide an appropriate media type for a particular application (e.g. video for video conferencing). As this research focuses on abstraction in terms of call control, we will not delve into abstraction obtainable with media logic. One point to note is that abstraction with media logic might be obtained by providing the appropriate media streams for different categories of application, therefore the application will be used to determine the media stream required. All the GAppMMCC interfaces are inherited from the GAppMPCC and thus can be used for call creation, manipulation and tear down.

createMediaNotification (appInterface : in IpAppMultiMediaCallControlManagerRef,notificationMediaRequest : in TpNotificationMediaRequest) : TpAssignmentID
reportMediaNotification (callReference : in TpMultiMediaCallIdentifier, callLegReferenceSet : in TpMultiMediaCallLegIdentifierSet, mediaStreams : in TpMediaStreamSet, type : in TpMediaStreamEventType, assignmentID : in TpAssignmentID) : TpAppMultiMediaCallBack
InitiateSession(notificationMediaRequest : in TpNotificationMediaRequest):CallID

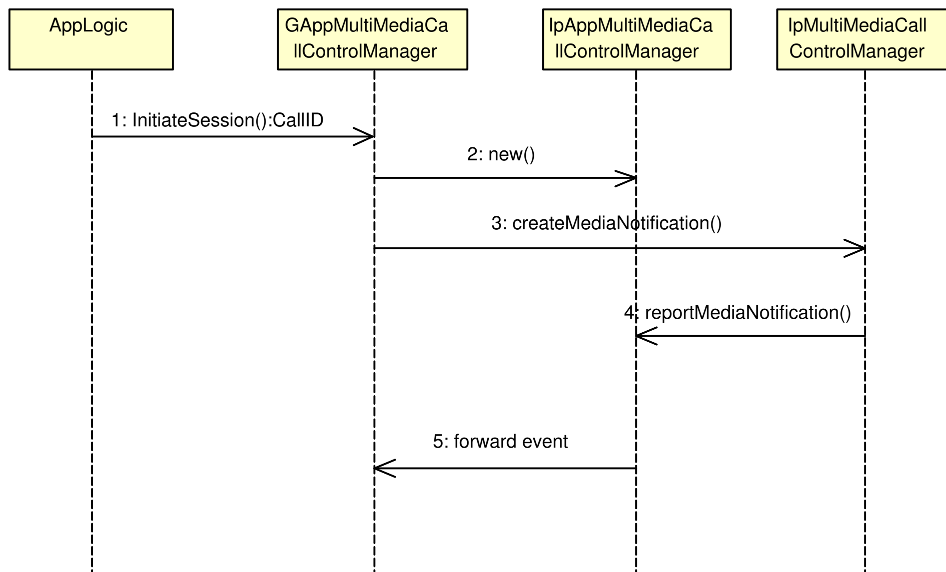

Figure 4.22: Media Session Initiation

The GApp media logic for creation of a media service follows from *initiateSession()* in GAppMPCC. There is little difference between figure 4.22 and figure 4.8. The difference is *createMediaNotification()* is used in figure 4.22 to indicate interest in media steams during call setup or media setup.

initiateSession() is invoked on $GApp_M$ (not shown). The $GApp_M$ then resolves the message by polymorphism(parameters) and forwards it to $GAppMMCC_M$. GApp-$MMCC_M$ then performs the required logic by creating a call back object, enabling a media notification point. An event happens (e.g. SIP INVITE) in the network matching the criteria specified in the *createMedianotification()*, *reportMediaNotification()* is then invoked on the call back interface and is used to pass the call identifier of the call requiring media to GAppMMCC.

4.2.4 GApp CCC Manager

We firstly would like to define the two central terms this section is built around as:

- Conference: Provides an application with the ability to manipulate sub-conference within it.
- Subconference: Provide groupings for call legs within a conference. Therefore only legs in the same sub-conference can have bearer connection (i.e. communicate).

There are two ways of starting a conference in Parlay; firstly the resources required for the conference can be reserved for commencement of the conference at a specific time. Secondly the conference is started as needed. When a conference is started an implicit subconference interface is created to enable communication between parties in the conference [25]. The application can perform the following functions on subconferences within the same conference:

- create new subconferences within the conference, either as an empty subconference or by splitting an existing subconference into two subconferences.
- move legs between subconferences.
- merge subconferences.
- get a list of all subconferences in the call.

createConference (appConferenceCall : in IpAppConfCallRef, numberOfSubConferences : in TpInt32, conferencePolicy : in TpConfPolicy, numberOfParticipants : in TpInt32, duration : in TpDuration) :TpConfCallIdentifier
reserveResources (appInterface : in IpAppConfCallControlManagerRef, startTime : in TpDateAndTime,numberOfParticipants : in TpInt32, duration : in TpDuration, conferencePolicy : in TpConfPolicy) : TpResourceReservation
conferenceCreated (conferenceCall : in TpConfCallIdentifier) : IpAppConfCallRef
getSubConferences (conferenceSessionID : in TpSessionID) : TpSubConfCallIdentifierSet
createSubConference (conferenceSessionID : in TpSessionID, appSubConference : in IpAppSubConfCallRef, conferencePolicy : in TpConfPolicy) : TpSubConfCallIdentifier
InitiateSession(startTime : in TpDateAndTime,numberOfParticipants : in TpInt32, duration :...

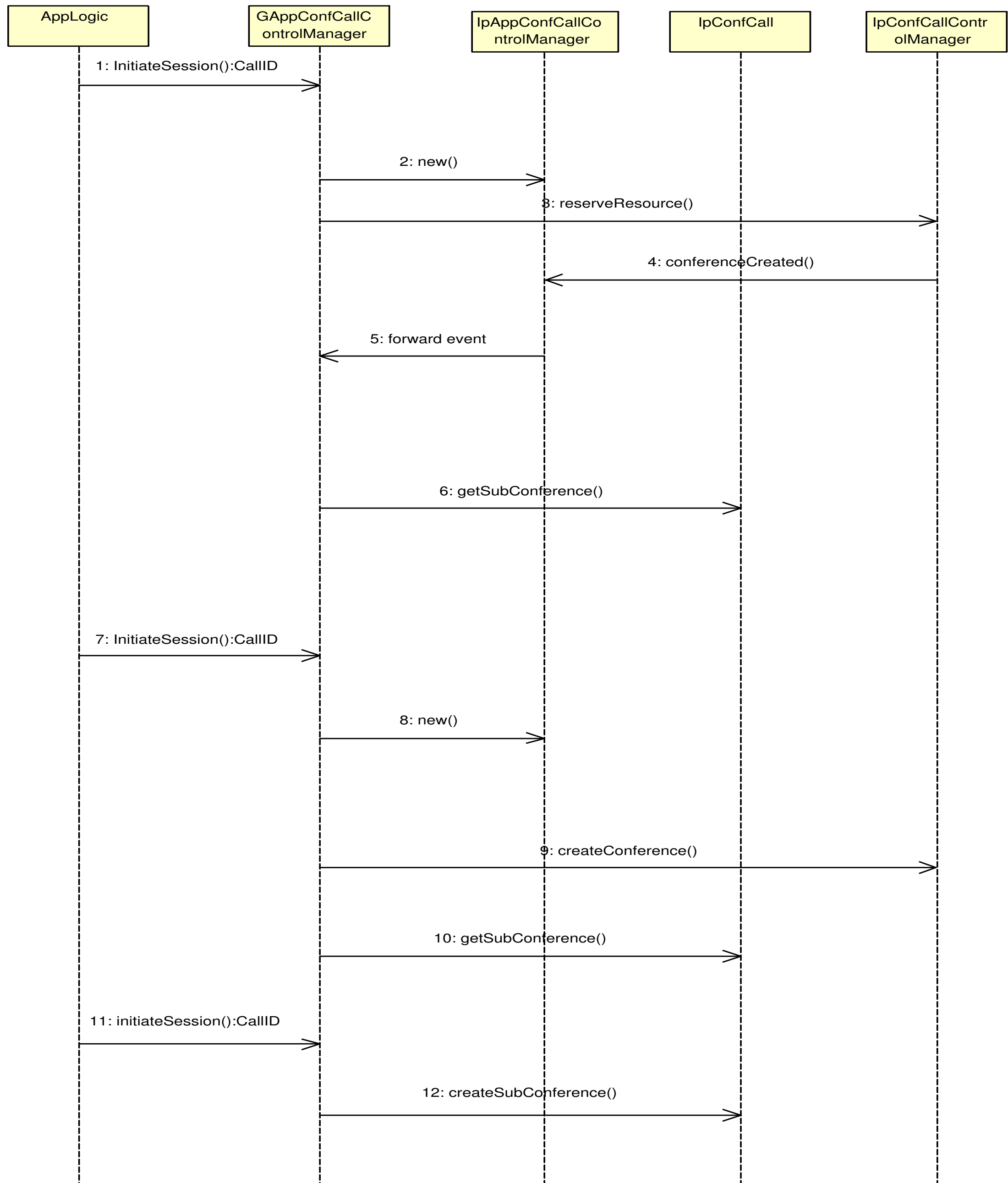

Figure 4.23: Initiate Conference Service

GApp provides abstraction on CCC interfaces by hiding details of the physical and structural representation of objects that perform conference call control. A conference is created by initiating a conference session. For each conference or subconference created, a unique *CallID* is allocated by GApp. The relationship between conferences and subconference is maintained within GApp by associating the *CallID* for all subconferences within the same conference. This is possible since the creation of a conference implicitly creates a “principal” sub-conference, and all other sub-conference created within this conference can simple be associated to this sub-conference. Therefore a user creating a new conference will specify whether he requires an association with an existing conference by specifying its CallID. Specifying a conference *CallID*, tells GApp to create the required sub-conference. As a result the user can be abstracted from the internal objects that provide conference calls (i.e sub-conferences) and it reference, all the user need know is their exist a conference X which has a relationship with conference Y. If conference Y in turn has been created with a relationship with conference Z, then X has a relationship with Z, since within the Parlay gateway they all form sub-conferences within the same conference.

Thus GApp (as illustrated in figure 4.23) can be used to create conferences in three ways

- Scheduling and starting a conference with no relationship to an existing conference. Message 1 is invoked on GApp with *startTime*, *numberOfParticipants* and *duration*.
- Starting the conference as required with no relationship to existing conferences. Message 7 is invoked on GApp with *numberOfParticipants* and *duration*.
- Starting a conference with a relationship to an existing conference. Message 11 is invoked on GApp with *CallID*.

Otherwise other methods such as *splitConference()*, *mergeConference()* etc, can be applied through the GApp interface as synchronous or asynchronous methods as explained in section 4.2

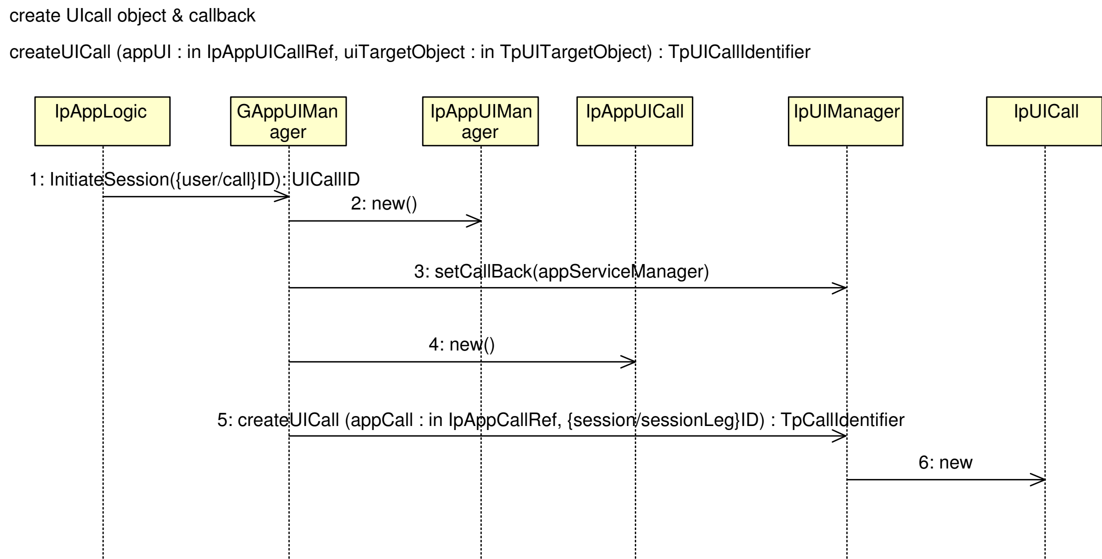

Figure 4.24: Application creates a user interaction call

4.2.5 GApp UI Manager

In this section we detail our work on GAppUI$_M$ by looking at its aspects that affect user call processing. Therefore we provide reusable blocks for UI-Call based processing. But the logic is similar for non UI-Call based processing. In conformance with previous sections we only provide information on logical blocks that have changed in relation to GAppServiceManager. The other blocks are given in appendix C.

In order to make user interaction work with our model and provide abstraction by abstracting leg management information, we need to provide the ASP with the ability to associate *userID* and *callID* with a UI object. This is done as shown in figure 4.24

The user needs to specify the *userID* or *callID* when invoking *initiateSession()* on GAppUI$_M$ as shown in figure 4.24. This is then converted to the appropriate identifier and used by the GAppUI$_M$ as the target address in the method *createUICall()* that takes a target object as one of its parameters [24].

4.3 Chapter Summary

This chapter details the design of the components GAppFW, GAppGCC, GAppCCC, GAppMPCC, GAppMMCC and GAppUI. Using the principles stated in section 3.3 new logical blocks were identified for these GApp interfaces. The logical blocks include *initiateSessionMember()*, *initiateSession()* etc. The logic used to provide this logical blocks is then detailed through sequence diagrams.

Chapter 5

Implementation of GApp layer

In this chapter we provide details of GApp layer implementation by detailing the experimental setup and the functionality of the equipments within the setup.

5.1 Overview

The experimental setup includes three machines. The operating system running on the three machines was SUSE 9.2 Linux. The three machines served the following purposes:

1. Machine 1 served as the ASP. It served as home to the GApp API and callback interfaces of the Parlay APIs.
2. Machine 2 served as the network operator, with the service layer part of the API (SCF) implemented on it. The implemented SCF in this project was the multi-party call control SCF.
3. Machine 3 served as the Domain Name Server, it is used for registration of active server manager objects i.e. it holds information about the location of registered server objects. This server can be queried by a client to obtain a server manager object.

Figure 5.1 illustrates the experimental arrangement and the workings of each machine within the simulation.

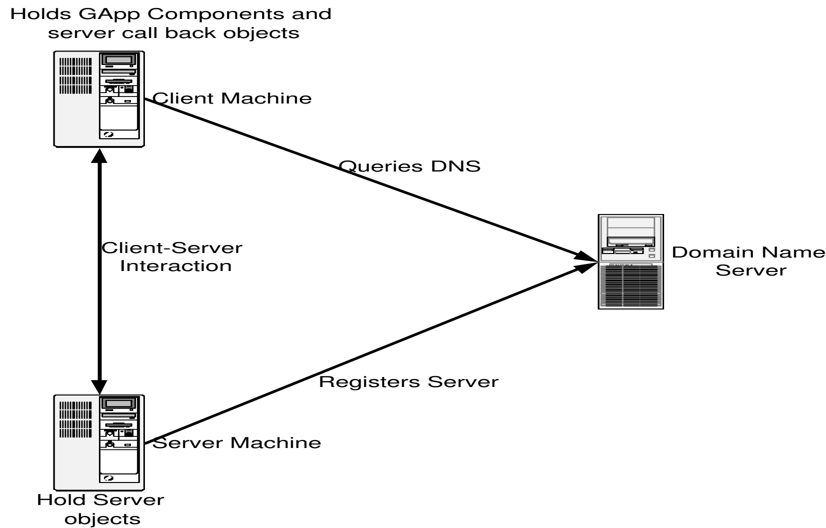

Figure 5.1: Experimental Setup

In order to implement GApp we have to implement new classes on top of the Parlay API. Since this is a proof of concept we choose to implement classes that simplify communication with Parlay MPCC API. Hence in this simulation we implemented the GApp MPCC API. This was done by using the MPCC IDL definition from the standards and mapping it through CORBA into a C++ implementation of the MPCC API. The skeletal output was filled with procedures that enable indication of process taken place. The new GApp MPCC class methods were then made by logically combining calls to the Parlay MPCC API. In the following sections we give details of GApp MPCC class implementation and there functionality.

5.2 Client (ASP) and Server (SCS)

In this section we provide details of the simulated ASP and the functionalities that were tested. GApp was implemented with the aid of a helper data structure (Linked list) created for the purpose of storing reference of server objects. There are two types of linked lists maintained:

1. one is used to hold the reference to all call object.
2. the other is used to hold the reference to the call leg objects.

The relationship between the two is each call object in the call list has a call leg list associated with it. As a result for each call created and added to the call list a corresponding call leg list is created to hold the parties participating in the call. A linked list class diagram is shown in figure 5.2

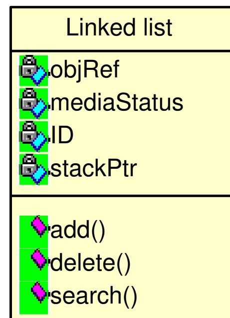

Figure 5.2: Linked List Data Type

The data structure has

- **ID** uniquely identifies this object.
- **objRef** is a reference to the server object.
- **media_status** indicates whether media communication is enabled on objRef.
- **stackPtr** is a pointer to a linked list contained in this linked list object. This is to enable storage of call leg references in the call object associated with it.

As a result, one can use a unique ID i.e call and call leg ID as explained in section 3.2.2 to address existing call and call legs.

GApp is implemented as specified in the sequence diagrams in chapter 4. The service layer functionality is obtained by querying the DNS for a service manager object reference. We then use this reference to perform the service functionality needed by GApp methods. This includes:

- initiateSession: This includes the creation of a call object and its call back. On creation a unique callID is created which is stored in the linked list together with the reference to the call object.
- endService: This includes removing the callback for the service manager and the service manager reference from the GApp manager.

- release: release works with two objects. These are
  - call: This step is accomplished by querying the call linked list for the reference of the call to be released. On obtaining the reference we then delete the call object from the linked list and use the reference to release the resource held by it in the server.
  - call leg: This step is accomplished by querying the call linked list for the reference of the call, and thereafter searching the call leg's linked list for the reference of the call leg to be released. On obtaining the call leg reference we then delete the call leg object from the linked list and use the reference to release the resource held by it in the server.
- initiateSessionMember: This includes the creation of a call leg object and its call back object. On creation a unique callLegID (callID appended to E164 number) is created which is stored in the linked list together with the reference to the call object.
- getmembers: returns all the callLegID for all the parties associated to a call object. This is achieved by going through the call linked list to obtain the appropriate call and then going through its call leg's list and returning all the parties (callLegIDs) in the call.
- getsession: returns all callIDs for all calls within a service. This is achieved by going through the call linked list and returning all callIDs for all the call objects in the list.
- suspend: suspend works with two objects. These are
  - call: This step is accomplished by querying the call linked list for the reference of the call to be suspended. On obtaining the reference we check the status of the call object whether active or not. If active we obtain all the call legs associated with this call and detach the media stream from each one. The call status is then set to inactive.
  - call leg: This step is accomplished by querying the call linked list for the reference of the call, and thereafter searching the call object's call leg linked list for the reference of the call leg to be suspended. On obtaining the call leg reference we check the status of the call leg object whether active or not. If active we detach the media stream from it. The call leg status is then set to inactive.

- resume: resume works with two objects. These are
  - call: This step is accomplished by querying the call linked list for the reference of the call to be resumed. On obtaining the reference we check the status of the call object whether active or not. If inactive we obtain all the call legs associated with this call and add a media stream to each one. The call status is then set to active.
  - call leg: This step is accomplished by querying the call linked list for the reference of the call, and thereafter searching the call object's call leg linked list for the reference of the call leg to be resumed. On obtaining the call leg reference we check the status of the call leg object whether active or not. If inactive we add a media stream to it. The call leg status is then set to active.

5.3 Usage Scenario

In this section we work through the application developed for GApp usage. This is a simple text interaction menu application prompting the user for inputs depending on the GApp functionality the user wants to test. This application enables users to test the simplification attained by using GApp on top of the Parlay SCF layer. The first menu is a simple welcome menu shown in figure 5.3 The second menu provides the

```
                    Welcome to GApp MPCC Gateway


Initialising system components....
Found the Naming Service
MPCCM reference obtained
Creating call back for MPCCM
MPCCM callback created
```

Figure 5.3: Welcome Menu

functionality attainable and prompts the user to choose from the options provided. This is shown in figure 5.4.

Therefore if the user chooses to create a call session he will simple enter 1 and GApp handles the whole call creation process for the user. This includes communication with the server, creation of a call object on the server and return of the object

```
       ***********************************
Enter 0 to quit
Enter 1 to create a call session
Enter 2 to create a session member
Enter 3 to connect a session member to the session
Enter 4 to remove a participant from a call session
Enter 5 to end call session
Enter 6 to suspend a user from a call session
Enter 7 to enable a user to resume participation in a call session
Enter 8 to view all call session Identity (getcall)
Enter 9 to view all participants of a call session (getcallleg)

                     ***********************************
```

Figure 5.4: Main Menu

reference to the GApp interface. The GApp processing displayed on the client and server is displayed in figure 5.5 and 5.6 respectively

```
Creating a call back interface for the call object to be created
Callback created
Sending the Create Call message to the Parlay Gateway
Created call with Call ID 1
```

Figure 5.5: Client output for create call

```
Trying to create a call object
Call object with session ID 1 created
Call back set for call object 1
```

Figure 5.6: Server output for create call

5.4 Chapter Summary

This chapter provides an overview of the implementation of a GApp interface. This is done by detailing the process used to implement the GApp MPCC interface. We provide details of functions such as release, initiateSessionMember, initiateSession, suspend, resume etc. We also provide screen shots of a simple text application that uses GApp, and give explanations on the simplification attainable.

Chapter 6

Conclusion

The question answered in this research states:

- **Starting with Parlay APIs, how can one create reusable blocks that provide a higher level of abstraction:**
  - **to enable rapid application creation and deployment by Application Service Providers (ASPs)**
  - **with the same level of functionality as the Parlay gateway**

This document provides the details of the mechanism by which this research question is answered.

In chapter 3 we document details of a solution to the problem. The solution provides an architecture that adds a new layer called the GApp layer to the Parlay architecture. The GApp layer is a simple model that solves the Parlay complexity problem by becoming a mediator between the Parlay SCFs and the ASPs. The result is provisioning of a simple interface for an ASP to communicate with and a complex interface for communication with the SCFs. As a result GApp's internal architecture has two sub layers, this is shown and explained in section 3.2.

After giving the architectural view in chapter 3 we then provide the logic used to provide the GApp layer components in chapter 4. Thereafter we implement an application that uses a GApp interface on the Parlay MPCC SCF. The solution proves that GApp provides a higher level of abstraction with the same level of functionality as the Parlay SCF interfaces. This is true because GApp interfaces where built by

examining the functionality (use case) of Parlay SCFs. As a result more developers can provide Parlay services.

As explained in chapter 2 Parlay provides the developer with a number of methods and interfaces which require the user to possess telecomm knowledge thereby limiting the developer base. Parlay X tries to solve this problem but due to its synchronous message format, it takes control away from the programmer which results in functionality lost.

In assessing the solution provided we observe that the architecture simplifies the Parlay gateway and provides methods that allows one to communicate with the gateway at a granularity level that is equal to the functionality provided by the gateway, therefore we provide abstraction with no functionality lost. Since we have not oversimplified the Parlay gateway we believe it would encourage programmers to program using Parlay APIs and also allow them to explore the full functionality of the Parlay API.

In the future, the implementation of the full GApp design and its usage in creation of complex application such as telemedicine and video conferencing should be explored. A further research may attempt to find functional blocks that make up these complex applications. Therefore one would look for service independent blocks that constitute these applications and check the relationship of these blocks to the functions identified in this research.

References

[1] TINA Consortium, "*Definition of Service Architecture. Computational Model Overview*," June 1997. http://www.tinac.com.

[2] A. R. Modarresi and A. Mohan, "*Control and Management in Next-Generation Networks: Challenges and Opportunities*," *IEEE Communications Magazine*, vol. 38, pp. 94–102, October 2000.

[3] Telcordia Technologies, "*Next Generation Network (NGN) Services*." http://www.mobilein.com/NGN_Svcs_WP.pdf.

[4] P. Nana and S. Mohapi and H.E. Hanrahan, "*Re-usable TINA Service Components based on the Parlay API and TINA for the Next Generation Network*," in *Proceedings of the South African Telecommunications Networks and Applications Conference*, September 2002.

[5] A. Caric and K. Toivo, "*New Generation Network Architecture and Software Design*," *IEEE Communications Magazine*, vol. 38, pp. 108–114, February 2000.

[6] Web ProForum, "*Next Generation Networks*." http://www.iec.org/online/tutorials/next_gen/.

[7] R. Christian, "*Providing User Context Support for Next Generation Network Services*," MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2004.

[8] J. Bakker and D. Tweedie and M. Unmehopa, "*Evolving Service Creation; New Developments in Network Intelligence*," *Telektronikk*, vol. 4, 2002. http://www.argreenhouse.com/papers/jlbakker/Bakker-telenor.pdf.

[9] P. Nana, "*On a Hybrid TINA-Parlay Service Architectecture for Next Generation Networks*," MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2003.

[10] C. Young-Han, "*Next Generation Network activities in ITU-T*," in *Proceedings of the 8th International Conference in Intelligence in Next Generation Networks*, (Bordeaux), pp. 46–51, April 2003.

[11] H.E. Hanrahan and D. Mwansa, "*A Vision for the Next Generation Network*," in *Proceedings of the South African Telecommunications Networks and Applications Conference* , September 2003.

[12] ITU, "*NGN Working definition*." http://www.itu.int/ITU-T/studygroups/com13/ngn2004/working_definition.html.

[13] Parlay Group, "*Parlay X Web Service White Paper* ." http://www.parlay.org.

[14] Parlay Group, "*Parlay Overview*." http://www.parlay.org/imwp/idms/popups/pop_download.asp?contentID=424.

[15] Z. Lozinski, "*Parlay/OSA - a New Way to Create Wireless Services*," May 2003. http://www.parlay.org/docs/2003_06_01_Parlay_for_IEC_Wireless.pdf.

[16] Rococo Software, "*An Introduction to OSA/Parlay* ." http://http://www.rococosoft.com/docs/osa_parlay_wp.pdf.

[17] A. Moerdijk and L. Klostermann, "*Opening the Networks with Parlay / OSA APIS: Standards and Aspects behind the APIS*." http://www.3gpp.org/ftp/tsg_cn/WG5_osa/_Pesentation_Tutorial_Press-Release/Opening_the_networks_with_Parlay_OSA.PDF.

[18] ETSI ES 202 915-3, "*Open Service Access (OSA); Application Programming Interface (API); Part 3: Overview (Parlay 4)*," August 2003.

[19] ETSI ES 202 915-4-1, "*Open Service Access (OSA); Application Programming Interface (API); Sub-part 1: Call Control Common Definition (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

[20] TINA Consortium, "*Overall Concepts and Principles of TINA*," Febuary 1995. http://www.tinac.com.

[21] ETSI ES 202 915-4-2, "*Open Service Access (OSA); Application Programming Interface (API); Sub-part 2: Generic Call Control SCF (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

[22] ETSI ES 202 915-4-3, "*Open Service Access (OSA); Application Programming Interface (API); Sub-part 3: Multi-Party Call Control SCF (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

[23] ETSI ES 202 915-4-4, "*Open Service Access (OSA); Application Programming Interface (API); Sub-part 4: Multi-Media Call Control SCF (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

[24] ETSI ES 202 915-5, "*Open Service Access (OSA); Application Programming Interface (API); Part 5: User Interaction SCF (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

[25] ETSI ES 202 915-4-5, "*Open Service Access (OSA); Application Programming Interface (API); Sub-part 5: Conference Call Control SCF (Parlay 4)*," ETSI Standard V1.2.1, August 2003.

Appendix

Appendix A

GAppGCCLogic

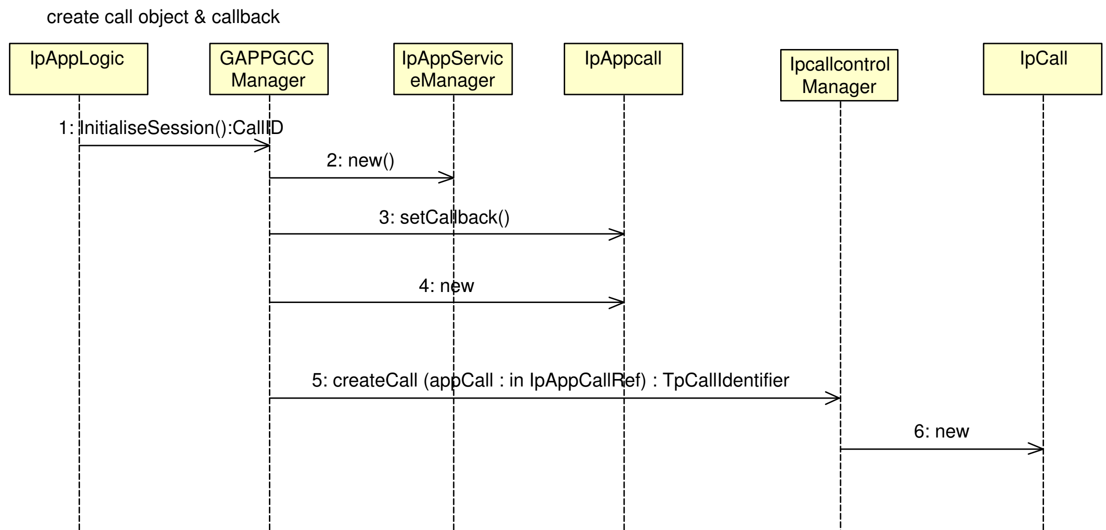

Figure A.1: Application Sets up Call

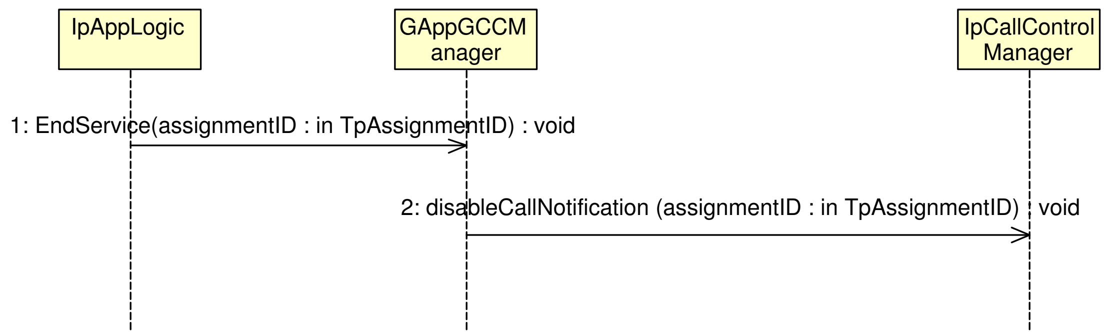

Figure A.2: Application ends Service

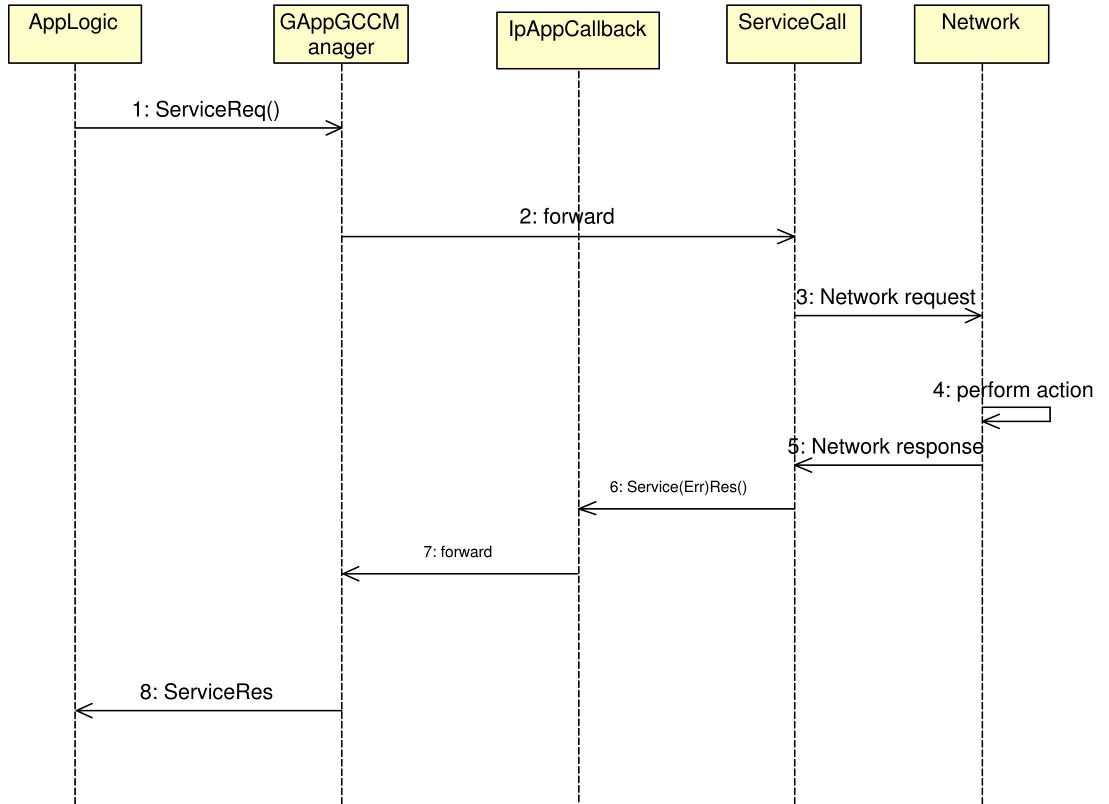

Figure A.3: Asynchronous Methods

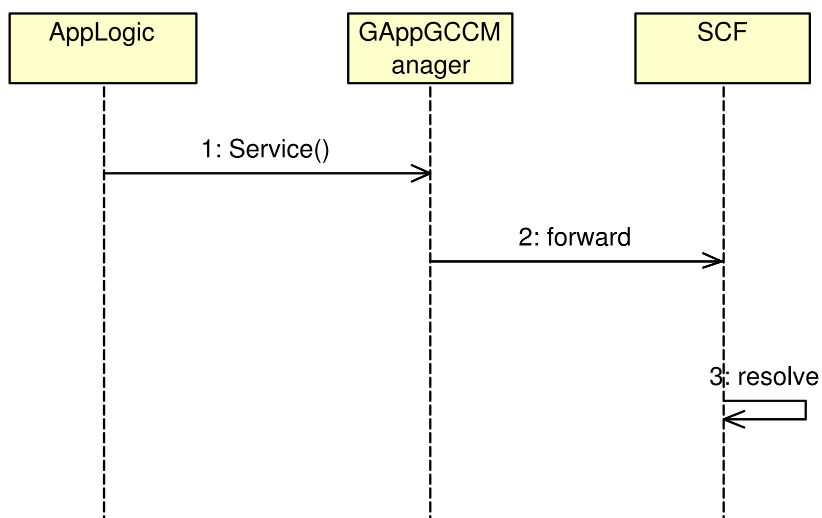

Figure A.4: Synchronous Method on SCF Interface

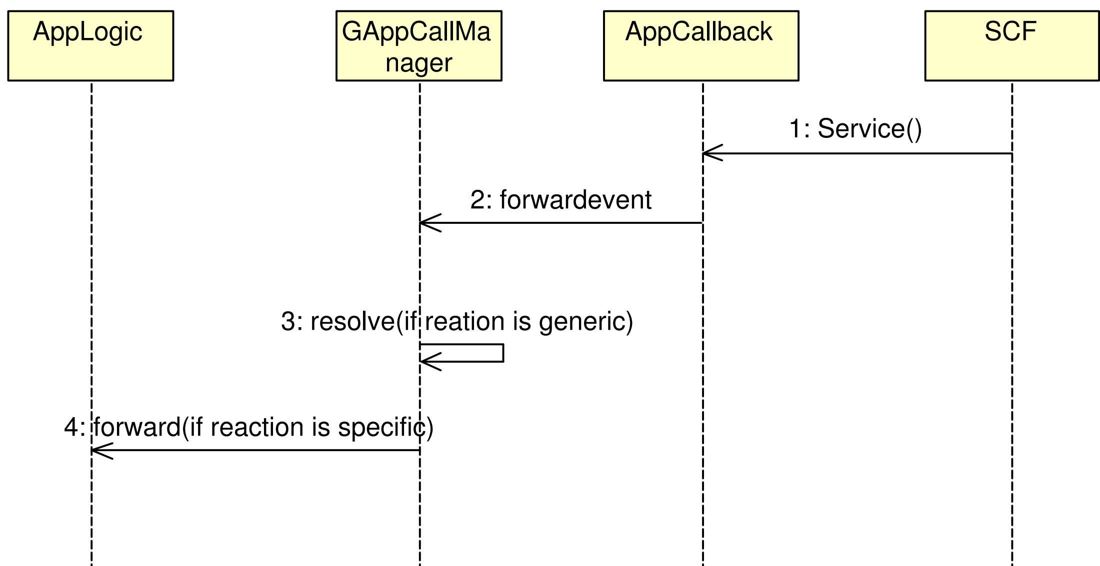

Figure A.5: Synchronous Method on Call Back Interface

Appendix B

GAppMPCCLogic

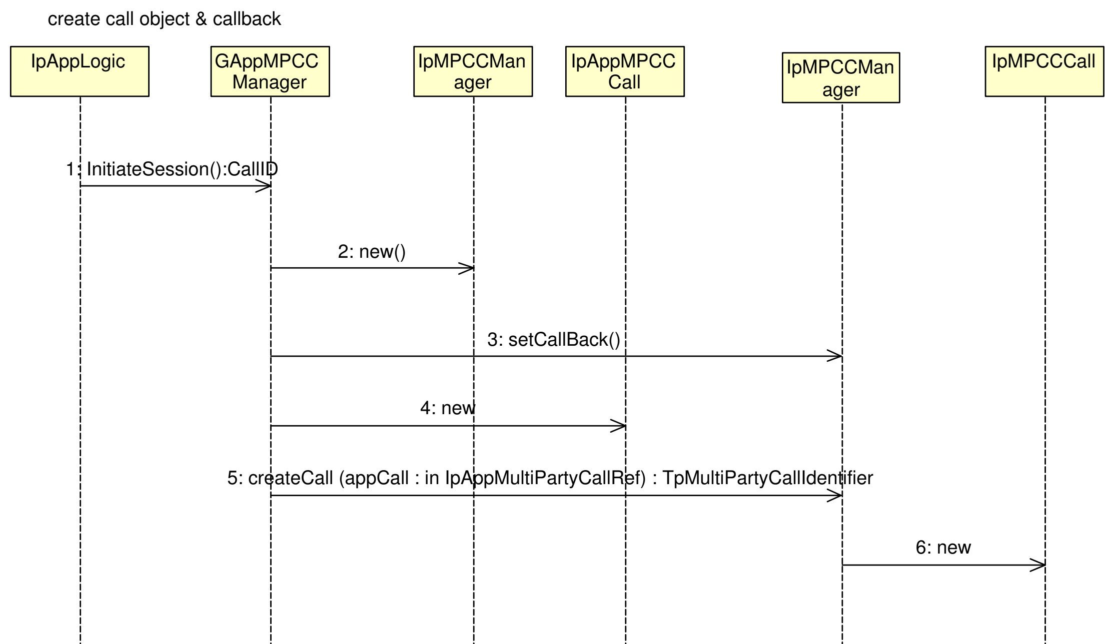

Figure B.1: Application Sets up Call

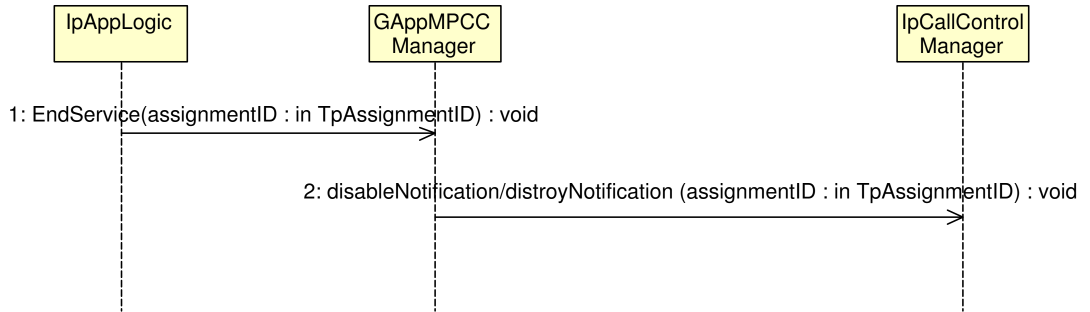

Figure B.2: Application ends Service

getInfoReq (callSessionID : in TpSessionID, callInfoRequested : in TpCallInfoType) : void
superviseReq (callSessionID : in TpSessionID, time : in TpDuration, treatment : in TpCallSuperviseTreatment) : void
eventReportReq (callLegSessionID : in TpSessionID, eventsRequested : in TpCallEventRequestSet) : void
getInfoReq (callLegSessionID : in TpSessionID, callLegInfoRequested : in TpCallLegInfoType) : void
superviseReq (callLegSessionID : in TpSessionID, time : in TpDuration, treatment : in TpCallLegSuperviseTreatment) : void

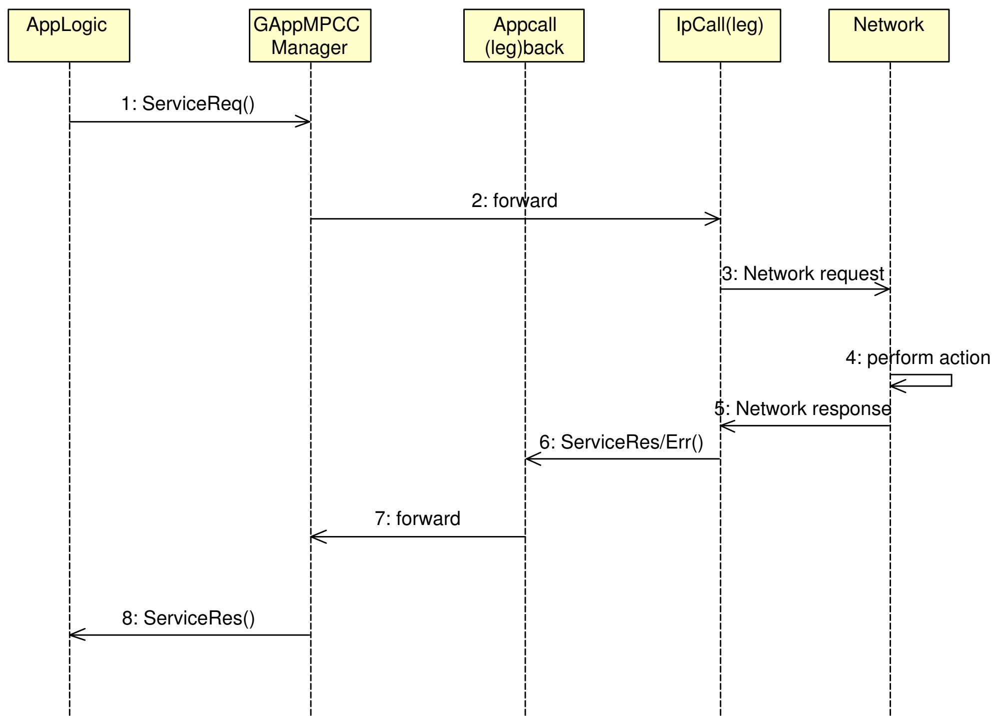

Figure B.3: Asynchronous Methods

These are syncronous methods that have not being abstrated

callleg
getCurrentDestinationAddress (callLegSessionID : in TpSessionID) : TpAddress
getCall (callLegSessionID : in TpSessionID) : TpMultiPartyCallIdentifier
setChargePlan (callLegSessionID : in TpSessionID, callChargePlan : in TpCallChargePlan) : void
setAdviceOfCharge (callLegSessionID : in TpSessionID, aOCInfo : in TpAoCInfo, tariffSwitch : in TpDuration) : void

call

setChargePlan (callSessionID : in TpSessionID, callChargePlan : in TpCallChargePlan) : void
setAdviceOfCharge (callSessionID : in TpSessionID, aOCInfo : in TpAoCInfo, tariffSwitch : in TpDuration) : void

Manager
changeNotification (assignmentID : in TpAssignmentID, notificationRequest : in TpCallNotificationRequest) : void
setCallLoadControl (duration : in TpDuration, mechanism : in TpCallLoadControlMechanism, treatment : in TpCallTreatment, addressRange : in TpAddressRange) : TpAssignmentID
<<new>> getNextNotification (reset : in TpBoolean) : TpNotificationRequestedSetEntry

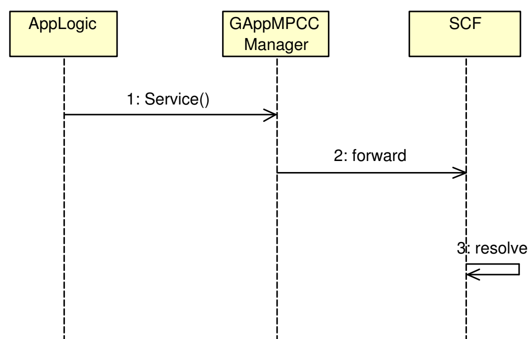

Figure B.4: Synchronous Method on SCF Interface

Manager
callAborted (callReference : in TpSessionID) : void
managerInterrupted () : void
managerResumed () : void
callOverloadEncountered (assignmentID : in TpAssignmentID) : void
callOverloadCeased (assignmentID : in TpAssignmentID) : void

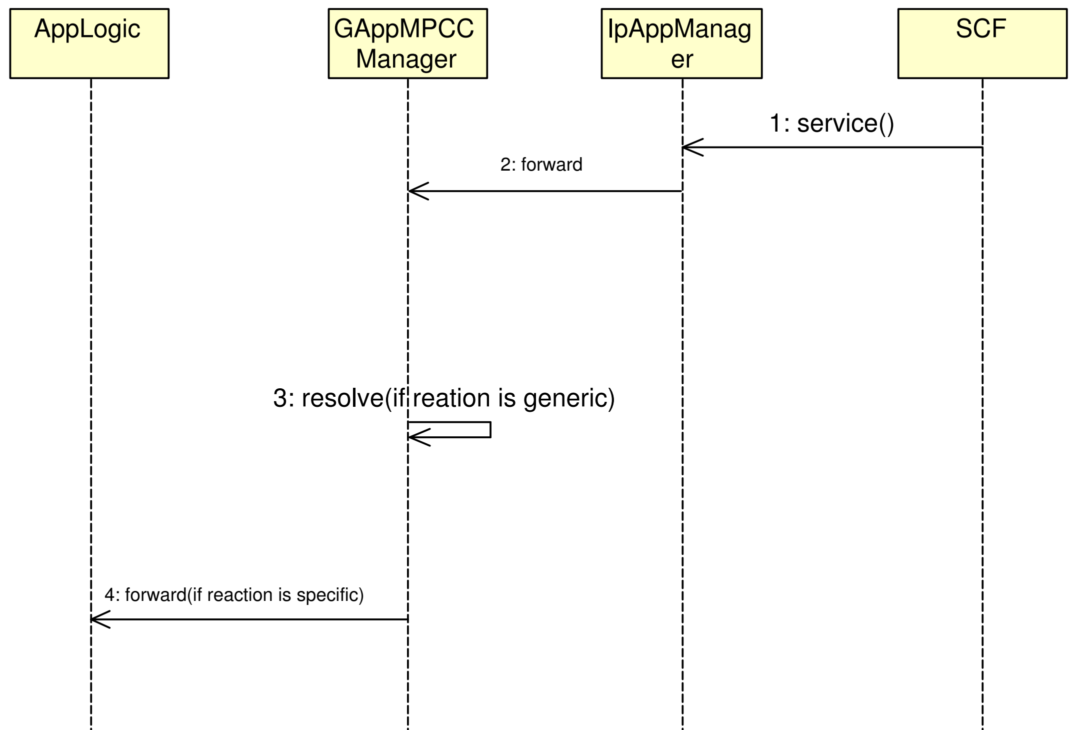

Figure B.5: Synchronous Method on Call Back Interface

Appendix C

GAppUILogic

<<new>> disableNotifications () : void
destroyNotification (assignmentID : in TpAssignmentID) : void

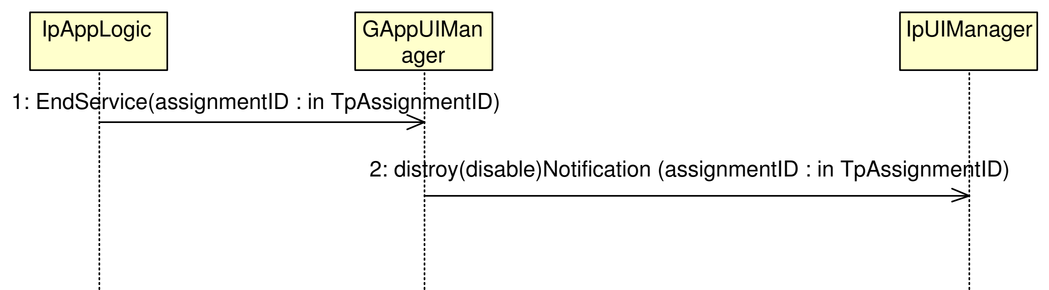

Figure C.1: Application ends Service

sendInfoReq (userInteractionSessionID : in TpSessionID, info : in TpUIInfo, language : in TpLanguage, variableInfo : in TpUIVariableInfoSet, repeatIndicator : in TpInt32, responseRequested : in TpUIResponseRequest) : TpAssignmentID
sendInfoAndCollectReq (userInteractionSessionID : in TpSessionID, info : in TpUIInfo, language : in TpLanguage, variableInfo : in TpUIVariableInfoSet, criteria : in TpUICollectCriteria, responseRequested : in TpUIResponseRequest) : TpAssignmentID
recordMessageReq (userInteractionSessionID : in TpSessionID, info : in TpUIInfo, criteria : in TpUIMessageCriteria) : TpAssignmentID
deleteMessageReq (usrInteractionSessionID : in TpSessionID, messageID : in TpInt32) : TpAssignmentID
abortActionReq (userInteractionSessionID : in TpSessionID, assignmentID : in TpAssignmentID) : void

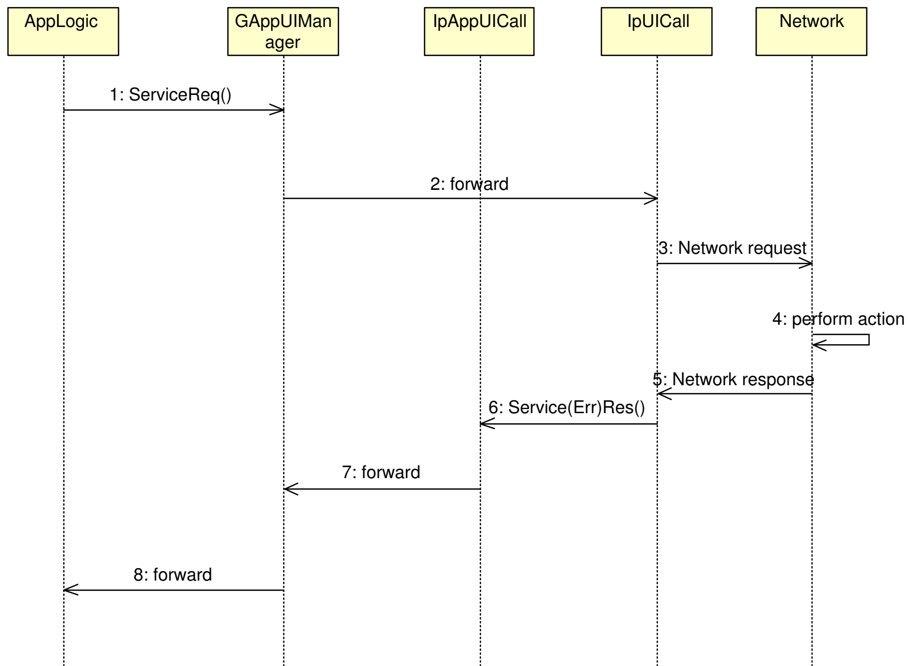

Figure C.2: Asynchronous Methods

changeNotification (assignmentID : in TpAssignmentID, eventCriteria : in TpUIEventCriteria) : void
getNotification () : TpUIEventCriteriaResultSet

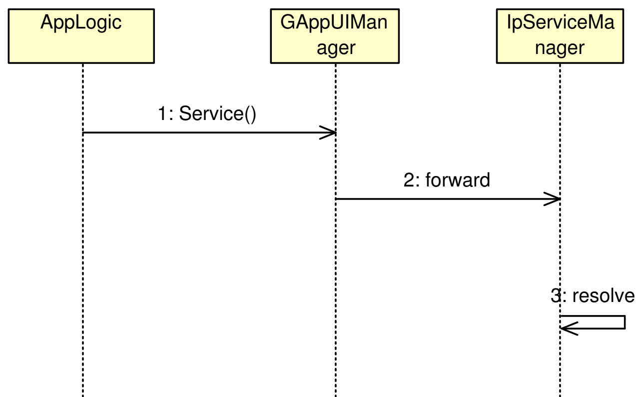

Figure C.3: Synchronous Method on SCF Interface

Manager
userInteractionAborted (userInteraction : in TpUIIdentifier) : void
<<deprecated>> reportNotification (userInteraction : in TpUIIdentifier, eventInfo : in TpUIEventInfo, assignmentID : in TpAssignmentID) : IpAppUIRef
userInteractionNotificationInterrupted () : void
userInteractionNotificationContinued () : void
<<new>> reportEventNotification (userInteraction : in TpUIIdentifier, eventNotificationInfo : in TpUIEventNotificationInfo, assignmentID : in TpAssignmentID) : IpAppUIRef

Call
userInteractionFaultDetected (userInteractionSessionID : in TpSessionID, fault : in TpUIFault) : void

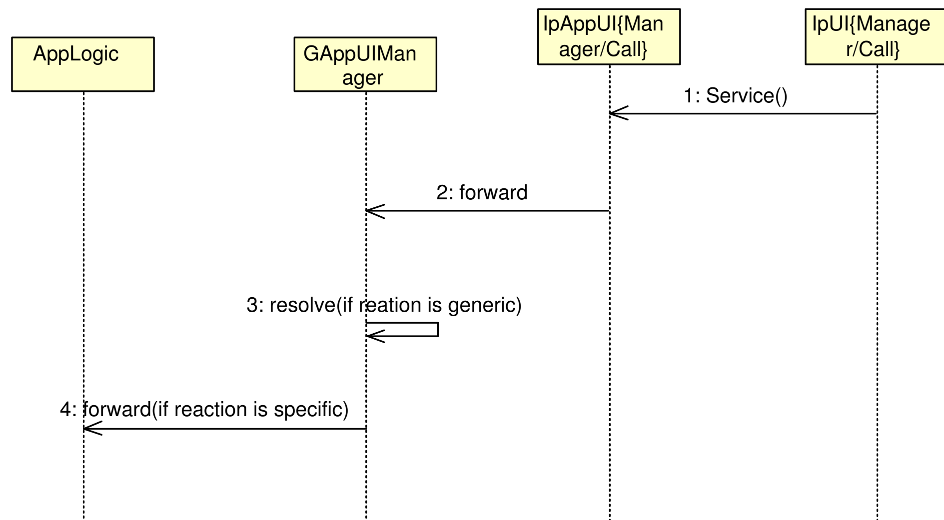

Figure C.4: Synchronous Method on Call Back Interface