A Consumer Premises End User Interface for OSA/Parlay Applications

Thabo Machethe

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

Johannesburg, December 2005

DECLARATION

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

Signed this ___day of_____20___

_____

Thabo Machethe

ABSTRACT

The NGN is a multi-service network which inter-works with the Public Switched Telephone Network (PSTN), the voice network and the data network provided by Internet. Through network independent APIs such as OSA-Parlay, the NGN slowly migrates and converges Telecoms and IT networks, voice and Internet, into a common packet infrastructure. The OSA/Parlay group defines a softswitch architecture which provides network independent APIs or SCFs that enable cross network application development The Parlay softswitch provides connectivity to underlying transport networks for application providers. The standard specifies the interaction between application providers and the softswitch. However, the standard does not specify an interface to regulate the interaction between service providers and the consumer/end user domain. This means that applications housed in the service provider domain have no defined interfaces to manage service delivery to the consumer domain. For most service providers, the lack of a non-standardized API set impedes efforts to decrease application creation and deployment time. This research investigates the design and implementation of a standard consumer interface which can be used by application providers within an OSA/Parlay system to deliver service content to end users. The main objectives with regard to the functionality provided by the interface include the integration of facilities which will assist application providers to manage end user access and authentication (to enable users to establish a secure context for service usage), subscription (to handle the subscription life cycle), and service usage management (to enable the initiation and termination of services). The TINA-Consortium (TINA-C) has developed a service architecture to support the creation and provisioning of services in the NGN. The TINA architecture offers a comprehensive set of concepts and principles that can be used in the design of NGN services. The architecture consists of a set of reusable and interoperable service components encapsulating a rich and well defined set of APIs aimed at supporting the interaction between application providers and consumers. TINA's session concepts, information structures, interfaces and service components can be used to support the design of a consumer premises end user interface for OSA/Parlay. This research also aims to explore the feasibility of using the TINA API within an OSA/Parlay system to support consumer domain service delivery. In order to implement the consumer interface for Parlay applications, the ability of the TINA service architecture to provide Access and Authentication management; Subscription and Profile management; and Service Usage management was investigated. The report documents the design and implementation of an OSA/Parlay consumer interface utilizing TINA service components and interfaces.

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, Siemens Telecommunications and the Department of Trade and Industries THRIP programme. This financial support was much appreciated.

I would like to thank my supervisor Prof. Hu Hanrahan for his guidance and support throughout the duration of this research project, and my colleagues at CeTAS for their valuable counsel during this research. I would also like to thank my parents, whose support has always been unwavering and without whom the opportunities I have received in life would never have been possible, and my sisters for their continued love and support. Finally, I would like to thank my girlfriend and close friends for their love, support and patience throughout my time as a master's student. Without all these people, I would not have been able to achieve my goals and aspirations.

TABLE OF CONTENTS

DECLARATION ..... **I**
ABSTRACT ..... **II**
ACKNOWLEDGEMENTS ..... **III**
TABLE OF CONTENTS ..... **IV**
LIST OF FIGURES ..... **VIII**
LIST OF TABLES ..... **I**
1 INTRODUCTION **1**
1.1 INTRODUCTION TO THE NGN 2
1.2 NGN SERVICE PROVISIONING 3
1.2.1 How have services been developed and delivered before the NGN? *3*
1.2.2 How has this changed with the advent of the NGN? *3*
1.3 NGN SERVICE ARCHITECTURE 4
1.3.1 Distributed Network Intelligence *4*
1.3.2 Architectural Layering *5*
1.3.3 Open Services Interface/API *6*
1.3.4 Context Aware Service Delivery in the NGN *8*
1.3.5 Personal Mobility *10*
1.4 PROBLEM STATEMENT 11
1.5 PROJECT REPORT OBJECTIVES 12
1.6 STRUCTURE OF THE REPORT 13
2 LITERATURE REVIEW **15**
2.1 THE OSA/PARLAY ARCHITECTURE 15
2.1.1 The Framework (or Framework Interfaces) *15*
2.1.2 OSA/Parlay Service Capability Features (SCFs) *16*
2.1.3 OSA/Parlay and Context Awareness *20*
2.1.4 Supported Services *20*
2.1.5 The OSA/Parlay Business Model *21*
2.1.6 Limitations of the OSA/Parlay Service Architecture *22*
2.1.7 Current Problems Deploying OSA/Parlay Applications *23*
2.1.8 Implementing an End User Interface as a Solution *24*
2.2 THE TINA SERVICE ARCHITECTURE 24
2.2.1 The TINA Business Model *25*
2.2.2 TINA Service Components and Interfaces *28*

2.2.3 The TINA Session Concept ... 36
2.3 TINA AND CONTEXT AWARENESS ... 39
2.4 TINA AND PERSONAL MOBILITY ... 40
2.5 OVERALL CONTRIBUTIONS OF TINA TO OSA/PARLAY ... 41
2.5.1 What support can a TINA Derived Consumer Interface give to Parlay Applications ... 41
2.5.2 What can it allow users to do? ... 42
2.5.3 Proposed Service Architecture ... 43
3 REQUIREMENTS SPECIFICATION FOR THE OSA/PARLAY END USER INTERFACE 45
3.1 REQUIREMENTS FOR ACCESS AND AUTHENTICATION MANAGEMENT ... 45
3.2 SUBSCRIPTION AND PROFILE MANAGEMENT REQUIREMENTS ... 46
3.3 SERVICE USAGE MANAGEMENT REQUIREMENTS ... 46
3.3.1 Service Usage Management Feature Sets ... 47
3.4 CONTEXT AWARE SERVICE DELIVERY REQUIREMENTS ... 48
3.5 SUMMARY ... 48
4 DESIGN OF THE ACCESS AND AUTHENTICATION API ... 49
4.1 ACCESS AND AUTHENTICATION INTERFACES ... 49
4.1.1 The i_Initial interface ... 50
4.1.2 The i_Authenticate interface ... 51
4.1.3 The i_Access interface ... 52
4.2 PROPOSED IMPLEMENTATION SCENARIOS ... 53
4.2.1 Contact a Provider ... 54
4.2.2 Login to a Provider ... 56
4.2.3 Logout from a Provider ... 60
4.3 CHAPTER SUMMARY ... 61
5 DESIGN OF THE SUBSCRIPTION AND PROFILE MANAGEMENT API ... 63
5.1 SUBSCRIPTION AND PROFILE MANAGEMENT INFORMATION ... 63
5.2 SUBSCRIPTION AND PROFILE MANAGEMENT INTERFACES ... 67
5.2.1 The i_Subscribe interface ... 67
5.2.2 The i_SubscriberInfoMgmt Interface ... 68
5.2.3 The i_ServiceContractInfoMgmt interface ... 70
5.3 PROPOSED IMPLEMENTATION SCENARIOS ... 73
5.3.1 Subscribe a New Customer (Becoming a Subscriber) ... 73
5.3.2 Modify Subscriber Information ... 76
5.4 CHAPTER SUMMARY ... 79
6 DESIGN OF THE SERVICE USAGE MANAGEMENT API ... 81
6.1 INITIATING AND TERMINATING A SINGLE PARTY SERVICE SESSION ... 81
6.1.1 The i_Access Interface ... 82

6.1.2 The i_SSManage Interface ... *82*
6.1.3 The Basic Feature Set ... *83*
6.1.4 OSA/Parlay Interfaces ... *83*
6.1.5 The IpAppLogic interface ... *83*
6.1.6 Proposed Implementation Scenarios ... *86*
6.2 INITIATING AND TERMINATING MULTI PARTY SERVICE SESSIONS ... 90
6.2.1 The TINA Multiparty and MultipartyInd Feature Sets ... *91*
6.2.2 The i_Invitation interface ... *93*
6.2.3 OSA/Parlay Interfaces ... *94*
6.2.4 Proposed Implementation Scenarios ... *95*
6.3 CONTEXT AWARE SERVICE DELIVERY ... 102
6.3.1 Device Characteristics ... *102*
6.3.2 User Preferences ... *103*
6.3.3 User State ... *103*
6.3.4 Proposed User Context Implementation Scenario ... *104*
6.4 CHAPTER SUMMARY ... 107
7 IMPLEMENTATION ENVIRONMENT ... 108
7.1 THE CORBA DPE ... 108
7.2 NETWORK IMPLEMENTATION ... 109
7.3 IMPLEMENTATION RESULTS ... 110
7.3.1 Access Session APIs: ... *110*
7.3.2 Subscription and Profile Management ... *111*
7.3.3 Service Usage Management ... *112*
7.3.4 Context Awareness ... *112*
7.3.5 Personal Mobility ... *112*
8 CONCLUSION ... 114
8.1 DISCUSSION ... 114
8.2 CONCLUSION ... 115
8.3 RECOMMENDATIONS FOR FUTURE WORK ... 118
8.3.1 Content Adaptation ... *119*
8.3.2 Federation ... *119*
REFERENCES ... 121
APPENDIX A ... 1
A.1 ACCESS AND AUTHENTICATION ... 1
A.2 SUBSCRIPTION AND PROFILE MANAGEMENT ... 2
A.3 SERVICE USAGE MANAGEMENT ... 4
A.3.1 Single Party Services ... *4*

A.3.2 Single Party Services *5*
APPENDIX B **7**
B.1 THE USER LOCATION CASE 7
B.2 UPDATING USER CONTEXT DURING ACCESS SESSION SETUP 9
B.3 IDL SPECIFICATION FOR I_USERCONTEXTMANAGEMENT INTERFACE 10
APPENDIX C **12**
C.1 ASUAP 12
C.2 PA 14
C.3 SPF 16
C.4 UA 18
C.5 SSUAP 25
C.6 SF 31
C.7 SUB 32
C.7 APP 36
C.8 UA2 38

LIST OF FIGURES

FIGURE 2.1: THE OSA/PARLAY SERVICE ARCHITECTURE 19
FIGURE 2.2: THE OSA/PARLAY BUSINESS MODEL 22
FIGURE 2.3: THE TINA-C BUSINESS MODEL 27
FIGURE 2.4: THE SERVICE COMPONENTS IN THE TINA SERVICE ARCHITECTURE 29
FIGURE 2.5: LIFETIME DEPENDENCIES AMONG SESSIONS. REDRAWN FROM [4] 38
FIGURE 2.6: THE HYBRID OSA/PARLAY-TINA SERVICE ARCHITECTURE 44
FIGURE 4.1: ACCESS AND AUTHENTICATION MANAGEMENT USE CASE SCENARIOS 54
FIGURE 4.2: CONTACT A PROVIDER SEQUENCE DIAGRAM 56
FIGURE 4.3: LOGIN TO A PROVIDER SEQUENCE DIAGRAM 58
FIGURE 4.4: LOGOUT FROM A PROVIDER 61
FIGURE 5.1: SUBSCRIPTION AND PROFILE MANAGEMENT USE CASE SCENARIOS 73
FIGURE 5.2: SUBSCRIBE A NEW CUSTOMER SEQUENCE DIAGRAM 75
FIGURE 5.3: MODIFY SUBSCRIBER INFORMATION SEQUENCE DIAGRAM 78
FIGURE 6.1: TINA'S SERVICE SESSION COMPONENTS AND PARLAY'S GENERIC MESSAGING SCF 85
FIGURE 6.2: SINGLE PARTY SERVICE USAGE MANAGEMENT USE CASE SCENARIOS 86
FIGURE 6.3: INITIATE A SERVICE SESSION SEQUENCE DIAGRAM 87
FIGURE 6.4: TERMINATE A SINGLE PARTY SERVICE SESSION SEQUENCE DIAGRAM 89
FIGURE 6.5: TINA SERVICE SESSION COMPONENTS AND PARLAY'S GENERIC AND MULTIPARTY CALL CONTROL SCFS 94
FIGURE 6.6: MULTIPARTY SERVICE USAGE MANAGEMENT USE CASE SCENARIOS 95
FIGURE 6.7: INVITE A USER TO JOIN A SERVICE SESSION SEQUENCE DIAGRAM 97
FIGURE 6.8: JOIN A SERVICE SESSION WITH AN INVITATION SEQUENCE DIAGRAM 99
FIGURE 6.9: TERMINATE A MULTIPARTY SERVICE SESSION SEQUENCE DIAGRAM 101
FIGURE 6.10: USER CONTEXT MANAGEMENT USE CASE SCENARIOS 104
FIGURE 6.11: UPDATE USER CONTEXT (TERMINAL CAPABILITY UPDATE) SEQUENCE DIAGRAM 106
FIGURE B.1 UPDATE USER CONTEXT (USER LOCATION UPDATE) SEQUENCE DIAGRAM 7
FIGURE B.2 USER CONTEXT UPDATE DURING ACCESS SESSION SETUP 9

LIST OF TABLES

TABLE 1: ACCESS AND AUTHENTICATION MANAGEMENT INTERFACES, OPERATIONS, AND ATTRIBUTES..... 2

TABLE 2: SUMMARY OF SUBSCRIPTION AND PROFILE MANAGEMENT INTERFACES, OPERATIONS, AND ATTRIBUTES 3

TABLE 3: SUMMARY OF SINGLE PARTY SERVICE USAGE MANAGEMENT INTERFACE, OPERATIONS, AND ATTRIBUTES 4

TABLE 4: SUMMARY OF MULTIPARTY SERVICE USAGE MANAGEMENT INTERFACES, OPERATIONS, AND ATTRIBUTES 6

TABLE 5 FIELDS CONTAINED WITHIN THE LOCATIONS DATA STRUCTURE. REDRAWN FROM [35] 11

1 INTRODUCTION

In today's telecoms and IT infrastructures access to services is provided through a variety of access networks. For example, telephony is usually accessed through the PSTN, mobile telephony or messaging through a GSM network, and the Internet through an IP network. The range of applications is increased through steps already taken to standardize access to services through a single service delivery platform [1, 2]. In particular, the Parlay Group, an open, multi-vendor telecoms forum has been organized to create network independent APIs in order to enable cross network application development [1]. The APIs form the interface between the application layer or Service Network and the core network. Applications are logically positioned in the Service Network and can be deployed independent of the core network and access network that the end user is using. This means that instead of the current approach where applications are tied to one specific network (PSTN, Mobile, IP) applications can be accessed and used from different types of networks or domains [3]. The end user domain and its interface to Parlay-based applications has however received little attention. The Parlay standard only specifies the interaction between a service provider and a network connectivity provider.

Along with network independent APIs, service architectures to support the rapid creation and delivery of cross network applications have been developed. One such is the TINA-C service architecture developed by the TINA-Consortium (TINA-C) [4]. The TINA-C architecture provides a set of concepts and principles to be applied in the specification, design, implementation, deployment, execution, and operation of software components in large scale distributed networks. The main goal of the TINA-C architecture is to ensure separation of service network and communication network concerns by hiding underlying transport network technologies from end users, application developers and application service providers. In the context of this research it is important to note that the TINA-C service architecture documentation provides a detailed description of the end user domain. This is done by describing a set of end user domain related components and their interactions with a service provider network.

The convergence of the IT and telecoms infrastructures facilitated by Parlay and the TINA-C offers new opportunities to Telcos in terms of service delivery. Currently, the telco is able to provide national and international services by federating with other operators. The value-added services are primarily voice-centric, examples include freephone, voicemail, conference calling, operator assistance and customer calling. Simple data services such as fax and e-mail also operate over this network using circuit connections [5].

Telcos will be able to deliver a range of services to end users through a single access network. For example, access to telephony, mobile telephony, and Internet may all be offered in a single service. This will imply the availability of more data-centric services such as video on demand (VoD) and multiparty video conferencing to end users.

Furthermore, new types of services may also be offered. One such is the *Context-driven service*, which allows the delivery of a service (service components, parameters and content) in differing ways depending on the context. For example, a service delivery context may be user preference (audio/video quality), terminal capability (display resolution, color depth, network card) or network capability (end to end QoS level) [6]. Another is the *Location-based service*, which allows the delivery of a service based on the geographical location of an end user. For example, a freephone service could be offered at a conference for anyone within the area where the conference is being held. The location based service could also be used to, for example, notify subscribers of a "buddy" when other "friendly" subscribers are within the same geographical region [7].

In this project, we examine the design of an end user interface for applications using the Parlay gateway for connectivity. We propose that in this implementation the user should be allowed to access cross-network services through a single customizable user interface. It is proposed that service components specified in the TINA-C service architecture be used. Furthermore, the end user interface should accommodate Next Generation network capabilities such as context awareness, location based service delivery, and personal mobility. This implementation must also allow the user greater involvement in service management, for example the user should be able to change service profiles within permitted limits, and to subscribe on line to new services. The system will be illustrated through a proof of concept case study.

1.1 Introduction to the NGN

The NGN can be thought of as a packet-based network where the packet switching and transport network are logically and physically separated from the service or application network [8].

Most of the present NGN paradigms implement a softswitch architecture where the switching functionality is distributed across several nodes. All emerging NGN architectures share a common goal of rapid service creation and delivery. Softswitch architectures such as Open Services Access (OSA) or Parlay achieve this through the separation of service intelligence into service dependent and service independent logic across an open Application Programming Interface (API) [9]. The re-use of service-independent logic across the API by third party application service providers facilitates rapid service creation and delivery. TINA-C's service architecture, describes a set of feature-rich interfaces and operations useful in third party service provision. A subset of these interfaces are implemented to provide the functionality required in this project.

To allow for independent service development and delivery, as well as for changes in the communications technology, it is necessary for the application/service network to be decoupled from the transport network [8].

This report documents the design and implementation of an end user interface to support the delivery of multimedia multi-party OSA/Parlay services to end users or consumers.

The main focus in this hybrid architecture will be the definition of consumer domain components and interfaces for use in end user access and usage of 3rd party multimedia services.

1.2 NGN Service Provisioning

1.2.1 How have services been developed and delivered before the NGN?

Service provision in traditional Networks has usually been tightly coupled with the network elements and access devices specific to that network. This has led to services only being accessible to consumers with customer premises equipment (CPE) specific to the access and core networks. Cross network services utilizing multiple transport networks were uncommon and usually complex to develop. The major thrust of traditional network service providers has been to offer the mass market basic transport of information between end users involving narrowband voice calls, and simple data services such as fax and email, with various voice centric value-added capabilities such as freephone, operator assistance, and customer calling [15].

In traditional networks, the telecommunications environment consisted almost exclusively of telecommunications monopolies in which public Telcos acted as both network operators and service providers. Value added services were often developed completely within a single service provider's domain operated by the Telco. The relative speed at which services could be created and deployed was limited by a lack of availability of skills in the market. This was due to the fact that the telecommunications infrastructure was mostly based on vendor specific network equipment with proprietary interfaces. New value added services required skilled telecommunications experts with extensive knowledge on proprietary network technologies [30]. Niche and short-lived value-added services were often overlooked by Telcos as being commercially unfeasible. The cost involved in rolling out specialized services used by a handful of customers for a short duration of time could not be justified [5].

1.2.2 How has this changed with the advent of the NGN?

The traditional view of services is changing, while existing services remain part of service providers' offerings, more advanced broadband multimedia and information intensive services are also becoming available. With the advent of the NGN, a

convergence of the previously independent networks is taking place. The NGN facilitates for several types of convergence; application convergence, network convergence, and access device convergence. The focus in this research is application convergence. Application convergence indicates an aggregation of traditional service provider (or application provider) domains into a single domain capable of delivering multi-network services. This aggregation is facilitated by an architectural layering system which allows for application developers in the NGN to abstract from the underlying transport network in the design and implementation of application layer services. This allows for the development of services utilizing multiple transport network capabilities and functionalities, as well as the provision of services to multiple access network end users. Several key factors have contributed to the shift from traditional vertically aligned networks to today's horizontally aligned NGN architecture, the next section presents some of the most relevant of these factors.

1.3 NGN Service Architecture

One of the primary goals of NGNs is to provide a ubiquitous and open multi-provider telecommunications environment that can support multiple types of services and management applications over multiple types of transport. This section describes three critical characteristics of this Next Generation environment.

1.3.1 Distributed Network Intelligence

Increasing competition and the gradual convergence of IT and telecoms technologies has led to an increased focus on the rapid creation and deployment of advanced multimedia telecoms services with enhanced functionality utilizing different network technologies, end systems, communications protocols, operating systems, and programming language environments [37]. In an NGN services environment, a key technology solution to this problem is the Distributed Processing Environment (DPE). The main goal of a DPE is to enable distributed processing between services in a geographically distributed telecommunications system without concern for the underlying environment. This location transparency allows for the distribution of service intelligence across the network and the uncoupling of network intelligence from physical network elements. Commonly, DPEs which offer abstraction from the heterogeneity of the underlying environment (in terms of heterogeneous hardware, systems platforms, programming languages, and management policies) are referred to as *middleware* [37]. Several architectures such as CORBA, COM/DCOM, and Java RMI that provide distribution transparency and support to distributed applications have been defined. In this research, CORBA is used to provide the required DPE functionality.

1.3.2 Architectural Layering

The concept of architectural layering is central to NGN environments. The NGN infrastructure uses a layering system similar to that in the internet stack to separate functionality. Five principal layers are used [38]:

- Application Layer.
- Service Capability Functionality (SCF) Layer.
- Network Service Capability Functionality (NSCF) Layer.
- Switching Layer.
- Transmission Layer.

The application layer contains basic functions and service features that can be logically combined to create applications/services. The most important characteristic of this layer is that services and connection control are separated from the transport network. Feature servers and Application servers are utilized to create NGN compatible Intelligent Network services and 3rd party service provider applications respectively.

The Service Capability Functionality (SCF) layer provides open and secure API which provide multi-network capabilities to 3rd party application developers. The APIs are used to communicate with resources from the underlying transport networks. They enable the setup of call legs and connections in various types of networks. JAIN, 3GPP, and Parlay define examples of SCF layer API. Using OSA/Parlay to model the SCF layer we can say that network capabilities provided by the APIs are housed within SCFs (Service Capability Functions) implemented in a softswitch. The combination of the Application and SCF layers form the NGN Service Architecture.

The Network Service Capability Functionality (NSCF) layer consists of components implementing the capabilities to address and control network entities and special telecoms resources such as, for example, media gateways, interactive voice response systems, and mailbox systems. The NSCF layer provides stream flow binding for SCFs in the SCF layer.

The Switching Layer provides the means to route transport flows. The Transmission layer provides “the means of carrying high volumes of packets as well as TDM streams between elements such as switches” [38]. The switching and transmission layers form the NGN transport network. In the remainder of this paper, we focus only on the application and SCF layers.

1.3.3 Open Services Interface/API

Another essential aspect of the NGN architecture is open, secure, and standardized architectures and interfaces. NGNs provide an application layer which allows for the abstraction of underlying transport network functionality through standardized open APIs. In this context, an API is a means by which a programmer writing an application program can make requests of an underlying communication infrastructure. By using standard APIs, NGN applications can be built with standard programming languages and tools. This helps to shorten the time span for service development. Open APIs also enable the provision of services to a broader market of users because they allow the deployment of NGN services onto multiple networks. In the following sections, we describe four API environments:

- 3GPP OSA and Parlay.
- Java APIs for Integrated Networks (JAIN).
- Telecommunications Information Network Architecture (TINA).

The main focus in this paper is the OSA/Parlay Architecture. We explore the TINA architecture to discover its suitability as a supporting architecture for OSA/Parlay. We discuss key concepts resulting from these API environments and briefly touch on the JAIN initiative.

1.3.3.1 OSA/Parlay

The main purpose of the OSA/Parlay architecture is to facilitate the convergence of wireless, PSTN, and IP networks through transport network technology independent APIs. The OSA/Parlay API allows enterprises to access and control a selected range of Network Service Capability layer functions. The set of API are grouped into a single logical gateway, which may be physically distributed among multiple servers.

A consequence of the introduction of the OSA/Parlay API is the simplification of telecommunications application development. The abstraction provided by the APIs to Network Service Capability layer functionality allows application developers to disregard programming aspects related to specific transport network technologies. This aids the rapid creation and deployment of NGN applications.

The API is composed of two major types of interfaces, Framework interfaces and Service interfaces which are grouped into Service Capability Features (SCFs). The SCFs provide OSA/Parlay applications with generic functionality to access and use network resources. Hence the SCFs hide the distributed complexity of the underlying network from the applications.

The OSA/Parlay standard describes two major roles within its business model, an enterprise operator (who manages 3rd party applications), and a gateway operator (who manages the framework and SCFs). A standardized API is detailed to support the interaction between these roles. It is important to note that an API to regulate the provision of OSA/Parlay applications to the consumer domain is not specified. The main focus in this research is to explore the design and implementation of a standard end user interface which can be used by enterprise operators/application providers within an OSA/Parlay system to deliver applications to end users.

1.3.3.2 The TINA Architecture

The main goal of the TINA-C service architecture is to provide a set of concepts, principles, rules and guidelines for the construction, deployment and operation of NGN services. It is important to note that in contrast to OSA/Parlay, the TINA service architecture defines a complete business model within the application layer. TINA defines a set of administrative domains, as well as their roles and responsibilities within the application layer. The TINA-C service architecture also identifies a set of reusable and interoperable service components to be used in a Distributed Processing Environment (DPE) in order to build services as well as how these components interact and can be combined to support the instantiation, management, and use of NGN services [4]. These service components encapsulate the functionality provided by the TINA API set to enable the administrative domains to carry out their defined roles and responsibilities. In the context of this research, we intend to explore the feasibility of utilizing the TINA service architecture to define an end user interface for Parlay.

The TINA-C architecture is composed of four main sub-architectures, the service architecture, network resource architecture, computing architecture and management architecture. The service architecture's major concern is the definition of service independent aspects that mainly support the service session segment such as service management, user subscription management, charging and accounting for service usage. For instance, in a Video on Demand (VoD) service functionality to start, stop, suspend or resume a movie, as well as to monitor accounting information would be provided. Within the NGN architectural layering model, the service architecture conforms to the Application and SCF layers. The service architecture accommodates NGN service requirements by hiding underlying transport network technologies from end users, application developers and application service providers. The TINA service architecture provides a well defined API defining the interaction between the service provider and consumer domains. For this reason, the focus on TINA in this research will be on the service architecture.

The network resource architecture mainly deals with resource control. The control and management of network resources in a transport technology independent way is described here. The main concern is the connection management in order to service requests made by applications from the service architecture. This part of the architecture spans both the NCSF and Switching layers [18].

The software or computing architecture ensures the development of interoperable and portable software components by defining a Distributed Processing Environment (DPE) [5]. The management architecture provides concepts and principles for effectively managing services, networks and computing infrastructures.

The TINA architecture has not been widely embraced by industry, and has largely remained at the research level. As such no telco implementations exist, however, a sufficient number of trial outputs do exist which validate the architectural concepts and principles. These trials also validate the well-defined set of TINA interfaces.

1.3.3.3 JAIN

JAIN provides a set of integrated network and resources APIs and a framework for building integrated services that span the public switched telephone network (PSTN), wireless, and packet networks. The API defines a programming interface to next-generation converged networks and as such is designed to hide the details of the specifics of the underlying network architecture and protocols from the application programmer [19, 50]. The API is also independent of network signaling and transport protocols. Thus applications may use various call control protocols and technologies such as SIP, H.323, SS7, INAP, and others [50, 53, 54]. Using the JAIN API services may be easily deployed on a PSTN, packet (IP or ATM) network, a wireless network, or a combination of these without affecting their integrity.
The main components of JAIN include Java Call Control (JCC), Java Coordination and Transaction (JCAT) and Connectivity Management (CM), which reside in the communications network. The Java Service Logic Execution Environment (JSLEE) and Service Creation Environment (SCE) reside in the Application Service Provider domain.
The JAIN community also cooperates closely with Parlay/ETSI/3GPP in the specification process, and their Parlay/OSA APIs realized in Java are referenced in the standard. The objective is to produce '*one API for one developer community*' and the widespread adoption of one single open API throughout the telecommunications industry [53]. Towards this goal, the JAIN SPA specification aims at mirroring the Parlay standard by producing a JAVA based API replica of the Parlay Framework and SCFs.

1.3.4 Context Aware Service Delivery in the NGN

The emergence of the NGN brings about new possibilities for service delivery. One such is context aware service provision. In this section we explain context awareness in service delivery and detail some reasons why it is important in an NGN environment.

1.3.4.1 Why is Context Important in an NGN Environment?

The NGN promises a pervasive computing environment in which users are able to access computation and information from anywhere, at any time. One of the main factors differentiating pervasive services from others is the inherent variability within pervasive environments. There are three factors that contribute to this variability [25]:

- *Device Heterogeneity*. Client devices have different modalities and use different presentation formats. Hence, the same content is often presented differently on different devices.
- *Network Infrastructure:* There is large variation in the physical characteristics of wireless channels and this affects the performance perceived by the end user. This is due not only to the number of different such technologies available today but also to inherent properties of wireless channels like multi-path fading problems, distance between client and base stations, and interference problems resulting from shared spectrum.
- *User Context*: Services available to the user may change over time and with the user's location. For example, services accessed in a professional environment are very different from the ones accessed in a home environment.

Heterogeneity and mobility pose new challenges for information delivery applications in a pervasive environment. To meet the demands in this heterogeneous environment, it is necessary for the information to be customized or tailored according to the user's preferences, client capabilities and network characteristics [32].

1.3.4.2 What is Context?

The inherent variability in pervasive environments provides new challenges and opportunities for service providers. One such opportunity is the capability to deliver services based on context. In this research context can be defined as [27]:

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves."

According to this definition, almost any information available at the time of an interaction can be seen as context information (e.g. the current time and date, the usage history, the user profile, the temperature at the user location, the position of the user, etc). We however, use the following characteristics to categorize context [25]:

- *Network Characteristics:* refers to the information about the network infrastructure, including access network, bandwidth and QoS.
- *Device Characteristics:* refers to the information about the hardware device, which include the device type, processor speed and type, memory capacity, screen resolution and depth, and operating systems.
- *User Context:* includes all the user characteristics related to the user's environment. User context can be manually and/or automatically acquired.

The User Context is divided into:

- *User Preferences:* refers to information about the human user, including browsing preference, language preferences, display preferences, QoS preferences, age group and gender.
- *User State:* is composed of three attributes: the physical location of the user (ie.outdoors/indoors, building,); the state of the applications that she is currently accessing (eg. application specific information, server specific information); and which devices are active in her environment.

The main benefit of context aware service delivery is that services can adapt themselves to better suit the needs of the user and their task. For example, [33] notes several applications for context aware service delivery such as:

- presenting the context information itself as content to the user (e.g. a maps showing the current position).
- adaptation of presentation of information and services to a user (e.g. a GUI suitable for the mobile phone the user is using currently).
- triggering actions on the occurrence of context events.
- tagging context to information for later retrieval (e.g. weather information when taking a picture in order to let the photo lab adjust the development process).

1.3.5 Personal Mobility

A further capability offered by the NGN is personal mobility. This section introduces the concept of personal mobility and gives some reasons as to why it is important.

1.3.5.1 What is Personal Mobility?

Universal Personal Telecommunications (UPT) [55] describes personal mobility as the ability of a user to access telecommunications services at any terminal based on a personal identifier and the capability of that network to provide those services in accord with the user's service profile. Personal mobility "enables users to use services that are personalized with their preferences and identity ubiquitously, independently of both physical location and specific equipment." [4], such that, "the level of service obtained is dependent only on the capabilities of the access equipment and/or method, and restrictions imposed by the retailer and subscription." [4].

1.3.5.2 Why is Personal Mobility Important?

In an NGN environment, the convergence of multiple transport networks, service networks and access devices has led to an explosion in the number of devices able to access the same services. Unlike the traditional vertical networks, such as the IN, PSTN, or Mobile networks, where only a few devices of very similar capability were able to access a service network with services designed specifically for that network, the NGN model is based on a horizontal model which provides a single service network which can be accessed by a multitude of devices [15]. This means that users of NGN services are likely to access services based on device preference (rather than an inability of the device to connect to the service). Their preference may be based on device capabilities such as screen size, resolution, or simply the accessibility of the device at the current time. It is therefore important that unless the capability of a device is insufficient, or some service provider or user imposed restrictions are in place, NGN users' ability to access services not be limited by the type of device they use to access a service.

1.4 Problem Statement

This research investigates the design of a multi-service end user interface to support OSA/Parlay application creation and provisioning to the consumer domain. Given the limitations and current difficulties faced in the creation and provisioning of OSA/Parlay applications, the problem can be stated as:

- ***How can OSA/Parlay Applications be delivered to the Consumer Domain using a standard Consumer to Service Provider API set?***

The project intends to explore the feasibility of using the TINA service architecture to support the design and implementation of a Consumer to Provider API set for OSA/Parlay Applications.
The project also aims to explore how context awareness and personal mobility can be integrated into service delivery. Using the TINA service architecture, it is intended that an end user interface incorporating context awareness and personal mobility as well as providing consumer premises management functionality for OSA/Parlay applications can be implemented. Therefore the following subproblems can be stated:

1. ***Can the TINA Service Architecture provide the required functionality to design and implement a Consumer to Service Provider API set for OSA/Parlay applications?***

2. ***How can context awareness and personal mobility be integrated into the OSA/Parlay Consumer to Provider API set using the TINA Service Architecture?***

We intend to take the following steps to deal with the main problem:

1. Define a complete system of administrative domains and business roles within the OSA/Parlay application layer. With the definition of a consumer domain/role as the main focus.
2. Provide a well defined relationship between the administrative domains and business roles. Since the OSA/Parlay standard already defines a relationship between the service provider and connectivity provider domains/roles, the definition of a relationship between the consumer and service provider domains/roles is the main focus.
3. Design a set of APIs based on the defined relationships to govern the interaction between the different administrative domains. The interaction between the service provider and connectivity provider roles is standardized by Parlay, this research focuses on the definition of an API to support the interaction between the consumer and service provider roles/domains.
4. Implement an OSA/Parlay end user interface based on the defined APIs.

1.5 Project Objectives

The main objective in this research is the design and implementation of a consumer interface for OSA/Parlay applications. We state several other objectives with regard to the functionality we would like the OSA/Parlay consumer interface to provide:

1. The OSA/Parlay consumer interface should provide an Access and Authentication API which allows for service providers to manage consumers' access to services, as well as for the authentication of users and service providers during access.
2. The OSA/Parlay consumer interface should provide Subscription Management API to allow service providers to manage the set of consumers subscribed to their services.
3. The OSA/Parlay consumer interface should provide Service Usage Management API to support the initiation and termination of OSA/Parlay services by consumers.

The main goals of this research with respect to enabling context awareness in service delivery is to:

- develop a services platform that will support the creation, deployment and management of context-aware services.
- allow personalized and adaptive delivery of services.
- support context-aware features (including location-awareness).

1.6 Structure of the Report

Chapter 2: In this chapter we review the Parlay service architecture. We provide an overview of the Parlay's Framework and SCFs, support for context awareness, and the Parlay business model. Parlay's limitations with respect to consumer premises service provisioning are highlighted. We also provide an overview of TINA's business model, computational model, session concept, and ability to support context awareness and personal mobility is given. The main aim of the TINA review is to determine the feasibility of utilizing TINA service concepts and principles to support the construction of a consumer interface for the OSA/Parlay service architecture. The chapter concludes with a presentation of the system concept for the proposed OSA/Parlay consumer interface.

Chapter 3: The requirements for the OSA/Parlay consumer premises end user interface are presented. Requirements in the areas of the Access and Authentication, Subscription and Profile management, and Service Usage management, and context awareness are presented.

Chapter 4: A lightweight enterprise, computational, and information view of the Access and Authentication API segment of the consumer interface is presented. The interfaces and service components for access and authentication management are described. Use cases and sequence diagrams are then used to show the component interaction used to realize the access and authentication functionality in the consumer interface.

Chapter 5: This chapter details the design of a subscription and profile management API for Parlay's end user interface using TINA concepts, information structures and interfaces. The subscription and profile management APIs provide the service provider with the capability to manage the set of consumers subscribed to their services.

Chapter 6: The Service Usage management API are presented in a similar fashion to the previous two chapters. This chapter describes service usage and focuses on how the end user interface is linked to OSA/Parlay applications.

Chapter 7: In this chapter we discuss the implementation environment used to realize the OSA/Parlay consumer interface. We discuss the distributed processing environment used to implement the service components.

Chapter 8: The paper closes with a conclusion chapter consisting of a summary of the paper, a discussion of the results, and recommendations for future work.

2 LITERATURE REVIEW

In this chapter we first review the Parlay service architecture. We provide an overview of the Parlay architecture with reference to the Framework and Service Capability Features, support for context awareness, and the Parlay business model. Parlay's limitations with respect to consumer premises service provisioning are also highlighted. Next, the TINA service architecture is reviewed. An overview of TINA's business model, computational model, session concept, and ability to support context awareness and personal mobility is given. The main aim of the TINA review is to determine the feasibility of utilizing TINA service concepts and principles to support the construction of a consumer interface for the OSA/Parlay service architecture. The chapter concludes with a discussion of the proposed end user interface for OSA/Parlay applications.

2.1 The OSA/Parlay Architecture

The OSA/Parlay Architecture enables application convergence through the specification of open standard APIs which allow access for applications to transport network functionality in a manner independent of the transport network used. In this section, we look at the OSA/Parlay interfaces, their structure, and how they can be used to utilize multi-network transport functionality.

2.1.1 The Framework (or Framework Interfaces)

The framework provides applications with service access and authentication capabilities. It is also responsible for the management and administration of the SCFs offered by the OSA/Parlay gateway. It is important to note the difference between the definition of *Service* and "*Application*". In the scope of Parlay, Services refer to APIs/SCFs that provide the network resource abstraction to be used by Applications, which then use these resources intelligently to access multiple networks. In this paper however, the term *Service* will be used interchangeably with *"Application"*, whereas OSA/Parlay SCFs will

only be referred to as either APIs or SCFs. The OSA/Parlay framework mainly consists of the following interfaces [11];

- Authentication

Once an off-line service agreement exists between the application provider and the framework, the application can access the authentication interface. The application must be authenticated before it is allowed to use any other OSA interface. Multiple authentication techniques are supported to authenticate access by application providers to SCFs.

- Discovery

After successful authentication the application may access the Framework's Discovery interface for a list of available SCFs. The client application and Authentication Framework then negotiate a service agreement defining conditions of use for the discovered SCF.

- Event Notification

OSA/Parlay Applications may register callback interfaces with the Framework, and vice versa, for the purpose of event notification. The framework event notification mechanism is used to notify the application of SCF related events that have occurred and the application event notification interface is used by services to inform the application of a service related event.

- Integrity Management

The Integrity Management interface is used by the OSA/Parlay framework to monitor SCF load conditions, and to inform the application of SCF load condition changes or failure. A load balancing policy is agreed upon by both the application and framework before SCF use. The framework may also monitor the load conditions of applications and take appropriate action based on the load balancing policy in case of application failure or load condition change.

The Integrity Management interface may also be used by the application to report load conditions to the framework. Applications may also use this interface to query SCF load conditions.

2.1.2 OSA/Parlay Service Capability Features (SCFs)

The Service Interfaces provide the interfaces which expose the capabilities of the underlying transport networks. The current specification details 14 main SCFs:

- Call Control

    - Generic Call Control SCF
    - Multi-Party Call Control SCF
    - Multi-Media Call Control SCF
    - Conference Call Control SCF
- User Interaction
- Mobility
- Terminal capabilities
- Data session control
- Messaging
- Connectivity Management
- Account Management
- Charging
- Presence and Availability Management
- Policy Management

These APIs expose much of the core functionality of the underlying telecommunications network to an application developer, enabling the creation of new applications that use a standardized set of APIs. Most important to this research are the call control, terminal capabilities, mobility, and generic messaging API. These SCFs are summarized below.

2.1.2.1 Generic Call Control

The Generic Call Control Service (GCCS) provides the basic call control service for the API [39]. It allows calls to be initiated from the network and routed through the network. The GCCS supports enough functionality to allow call routing and call management for today's Intelligent Network (IN) services in the case of a switched telephony network, or equivalent for packet based networks. The API for generic call control does not give explicit access to the legs and the media channels. This is provided by the Multi-Party Call Control Service. Furthermore, the generic call is restricted to two party calls, i.e., only two legs are active at any given time. Active is defined here as 'being routed' or connected.

2.1.2.2 MultiParty Call Control (MPCC)

The Multi-party Call Control service enhances the functionality of the Generic Call Control Service with leg management [40]. It also allows for multi-party calls to be established, i.e., up to a service specific number of legs can be connected simultaneously to the same call.

2.1.2.3 MultiMedia Call Control (MMCC)

The MultiMedia Call Control service enhances the functionality of the MultiParty Call Control Service with multimedia capabilities [41]. To handle the multi-media aspects of a call the concept of media stream is introduced. A media stream is bi-directional media stream and is associated with a call leg. These media streams are usually negotiated between the terminals in the call. The multi-party Call Service gives the application control over the media streams associated with the legs

2.1.2.4 Conference Call Control (CCC)

The Conference Call Control Service enhances the multi-media call control service [42]. Each Conference Call Control interface inherits from a Multi Media Call Control interface, which in turn inherits from Multi Party Call Control. It is possible to implement conference call control without any multi-media features, using only those inherited methods which come from Multi Party Call Control, in addition to the Conference Call Control methods. The conference call control service gives the application the ability to manipulate subconferences within a conference. A subconference defines the grouping of legs within the overall conference call. Only parties in the same subconference have a bearer connection (or media channel connection) to each other (e.g., can speak to each other).

2.1.2.5 Mobility

The User Location service (UL) provides a general geographic location service [35]. UL has functionality to allow applications to obtain the geographical location and the status of fixed, mobile and IP based telephony users. UL is supplemented by User Location Camel service (ULC) to provide information about network related information. There is also some specialised functionality to handle emergency calls in the User Location Emergency service (ULE).

2.1.2.6 Terminal Capabilities

The Terminal Capabilities SCF enables the application to retrieve the terminal capabilities of the specified terminal [34]. Additionally it is possible for the application to request notifications when the capabilities of the terminal change in some way.

2.1.2.7 Generic Messaging

The Generic Messaging Service interface (GMS) is used by applications to send, store and receive messages [44]. GMS has voice mail and electronic mail as the messaging mechanisms. The messaging service interface can be used by both.

Figure 2.1 illustrates the positioning of applications, the framework and SCFs within the OSA/Parlay architecture. The network resource layer is also shown to illustrate the relationship between the Parlay SCFs and various network resources.

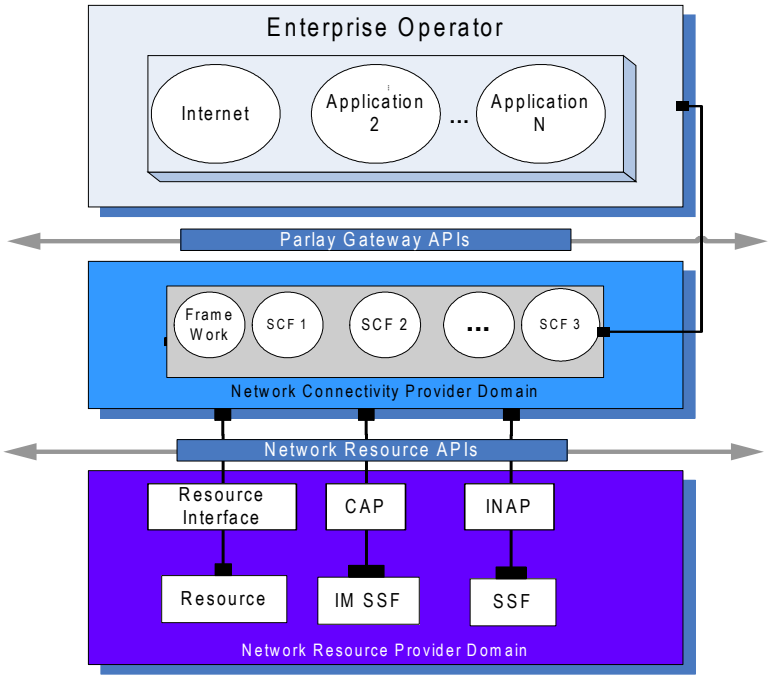

Figure 2.1: The OSA/Parlay Service Architecture

The capability of the Parlay SCFs to provide context awareness is discussed in the next section.

2.1.3 OSA/Parlay and Context Awareness

The Parlay Gateway provides a powerful set of SCFs that can be used to provide context awareness. Below are the categories of context as defined in Chapter 1, and the SCFs that may be used to support their implementation:

Network Characteristics: No SCF specifically provides network characteristics information.

Device Characteristics: The Terminal Capabilities SCF can be used to retrieve terminal information.

User Preferences: No mechanisms exist in OSA/Parlay to obtain user preference information.

User State: Physical location information can be retrieved using the Mobility SCF. Presence and Availability Management SCF can be used to support the management of users and their terminals' states as well as advanced service availability.

The next section presents a listing of the types of services that can be supported using Parlay's SCFs.

2.1.4 Supported Services

Below is a collection of the types of applications that can be supported by an OSA/Parlay gateway:

- **Basic Voice Services**

  Allows the delivery of basic voice services and value added voice services such as voice, call waiting, call forwarding. The GCCS, MPCCS, MMCCS, CCCS, GUIS can all be used to enable this service type.

- **Data Services**

  Allows for the setup of data connections within an IP network to setup services such as the internet, and web browsing. Can be supported using the Data Session SCF.

- **Generic Messaging Services**

  Supports the delivery of voice mail, email, fax mail, pages, and internet. Users may access, as well as be notified of, various message types (voice mail, email, fax mail, etc.), independent of the means of access (i.e., wireline or mobile phone, computer, or wireless data device). Can be supported using the Generic Messaging SCF.

- **Multimedia Services** (ie. video streaming, video conference) –

Multimedia Services allow multiple parties to interact using voice, video, and/or data. This allows customers to converse with each other while displaying visual information. It also allows for collaborative computing and groupware. Can be supported using the Multimedia Call Control SCF and/or Conference Call Control SCF.

- **Location Based Services**

  Location based services allow services to be delivered based on the user's location. Can be supported using the User Location SCF.

- **Context Aware Services**

  Services delivery dependent on user preference, or status. Can be supported using the Mobility, Terminal Capability, and/or Presence and Availability Management SCF.

In the next section, a business model for the Parlay service architecture is presented.

2.1.5 The OSA/Parlay Business Model

Figure 2.2 illustrates the OSA/Parlay business model. The OSA/Parlay architecture defines an incomplete business model which describes the relationship and interfaces between the service provider and network connectivity provider only. In the OSA/Parlay specification these roles are defined as the traditional network operator providing access to SCFs and the Enterprise Operator who provides 3rd party applications which are located in what is referred to as the Enterprise Domain.

The OSA/Parlay architecture pays little attention to the consumer domain. The OSA/Parlay specifications only detail the interaction between the 3rd party applications, the SCFs, and Enterprise Operators [11]. In the OSA/Parlay model, the enterprise operators act in the role of subscriber of services (SCFs) and the 3rd party applications act in the role of users or consumers of services. The framework itself acts in the role of retailer of services. However, the interfaces between an enterprise operator and the 3rd party applications in its domain are outside the scope of the Parlay API [11]. This gives the enterprise operators the capability to dynamically create, modify and delete the client applications and service contracts belonging to its domain. The next section elaborates on Parlay's limitations with regard to the lack of a consumer domain.

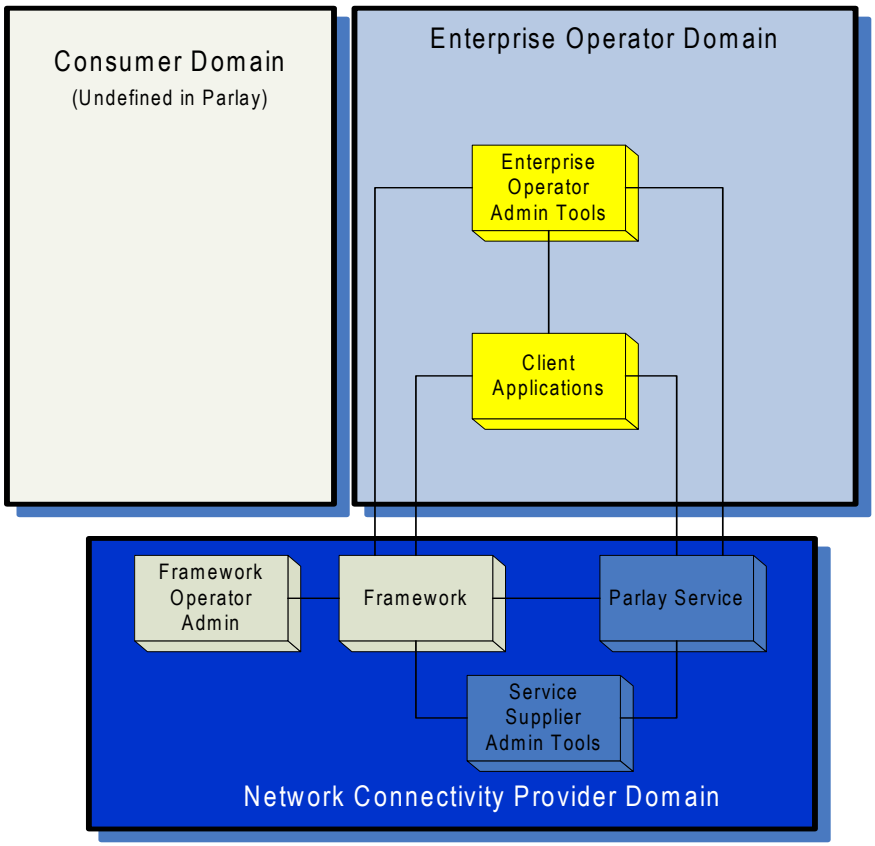

Figure 2.2: The OSA/Parlay Business Model

2.1.6 Limitations of the OSA/Parlay Service Architecture

OSA/Parlay defines an API between two business domains, a service provider (otherwise known as the Enterprise Operator domain in OSA/Parlay) and a network connectivity provider. OSA/Parlay does not however identify an API between a consumer and the service provider domain. This means that 3rd party applications housed in the service provider domain have no defined interfaces to manage service delivery to the consumer domain. The OSA/Parlay model does not specify how applications manage user access, authentication, and usage. In addition, no explicit model for user subscription and profile management is given. In the next section we discuss how the lack of a standard consumer to provider API in the OSA/Parlay model hinders application deployment and efficient service delivery to the consumer domain.

2.1.7 Current Problems Deploying OSA/Parlay Applications

The lack of a standardized set of APIs to support end user service access and usage management presents a number of significant concerns for service providers. For most service providers, the lack of a non-standardized API set impedes efforts to decrease application creation and deployment time.
Without the necessity to conform to a standard platform, most of the systems used by service providers are created independently. This means that varying systems and access methods are implemented, creating independent "proprietary" implementations in the service provider domain [28, 29]. Currently, this is the case with OSA/Parlay applications as no standard platform incorporating common service functionality such as Authentication, Access, Subscription exists to support Service Provider to Consumer domain service delivery and management.
Furthermore, it may also be the case that even internally, different development teams for a single service provider may create applications with their own unique systems (ie, authentication, subscription, etc) and each with its own unique method of access or use. This places a heavy burden on both maintenance of existing services and the development of new services for the following reasons [29]:

- **Tight Coupling** -- This creates a situation where there is a very tight coupling between the applications developed by different development teams and/or application providers and the systems they use to deliver services to users. This means that similar services may be delivered in a non-uniform manner to users, each with a different way of performing the same task.
- **Low Reusability** -- Application developers must work with a wide variety of systems, each with its own unique access methods and behaviors. This makes it difficult to build a new application in a standardized and repeatable way, and can often make the potential introduction of a new service prohibitive from a cost and risk perspective.
- **Limited Service Personalization** -- For example this method of proprietary bundling creates a situation where a user cannot personalize access, or service usage for her set of subscribed services, because "proprietary" access and usage mechanisms are in place for the same or different service provider offerings. Service personalization is therefore limited due to a lack of interworking service delivery systems. However, to build services that maximize growth and profitability of the subscriber base, more needs to be done in terms of understanding and improving individual user experiences through increasing personalization and diversification, and making the services respond to user needs in a dynamic, on demand fashion [28].
- **Multi-way Service Provider partnerships** for more complex services are difficult to support [28]. The lack of a standard platform makes it complex for proprietary systems to interwork with each other.

All of these problems are barriers to offering new services profitably, and often the business case for many new services cannot be justified.

2.1.8 Implementing an End User Interface as a Solution

In order for service providers to address the highlighted issues, it is essential that an end user or consumer domain interface for service access and usage management be introduced. Rapid service creation and deployment as well as uniform service delivery of OSA/Parlay applications can be facilitated by the development of an end user interface with the appropriate functionality. Through the use of reusable service independent logic, an end user interface can allow service providers to rapidly introduce new OSA/Parlay services.

It is therefore proposed that an end user interface be introduced between the OSA/Parlay Enterprise domain (ie. the Service Provider Domain) and the Consumer domain. The main aim of the consumer domain interface would be:

- *The provision of a standardized set of APIs to support end user access and usage of OSA/Parlay applications, as well as the management of end user subscriptions and profiles in order to provide authenticated usage, service customization, and context aware service delivery.*

Introducing a standardized way of creating and deploying new services allows for a normalization of the application development methods and toolsets such that efficient gains may be achieved leading to significant increases in efficiency. New services can be created in a greatly reduced timeframe because a standard agreed mechanism for accessing the common service logic can be used. This means common skill sets, toolsets and processes can be brought to bear on new developments. As projects are completed and deployed, new projects become lower risk and more predictable. Development teams (both inside and outside the organization) can work to a common, agreed upon set of interfaces to the core systems that underpin a new service [29].

2.2 The TINA Service Architecture

In order to define an end user interface for OSA/Parlay, we examine the TINA service architecture. TINA provides a well defined model for the construction, deployment, and operation of NGN services. We examine the TINA business model to determine whether it can be used in providing a more complete OSA/Parlay business model. The section also examines TINA's computational object model to determine whether the functionality it provides in delivering TINA services to the consumer domain can be adapted to deliver

OSA/Parlay services. The session concept is then introduced as a means of grouping common consumer-provider interactions. We discuss its relevance to OSA/Parlay applications and consider how it can be used to assist consumer-provider interactions in Parlay. The section concludes with a discussion of Personal mobility followed by a discussion of the overall contribution TINA can make to the definition of a Consumer to Provider API set for the OSA/Parlay service architecture.

2.2.1 The TINA Business Model

The TINA-C service architecture presents a set of business roles and responsibilities which provide a high-level description of the business situation in which telecommunications services are provided. [10] describes the scope of the service architecture with respect to the initial set of business roles and business relationships. The following terminology is used in the description of the business model [12]:

- A *stakeholder* is a representative of an organization or person responsible for a portion of the TINA system.
- An *administrative domain* is a portion of the TINA system owned by a single stakeholder. Several business roles may be aggregated by one administrative domain.
- A *business role* is a role in an administrative domain performed by a stakeholder. A stakeholder may perform several business roles in a single administrative domain. For example a telecom operator may act as a retailer, while providing services to consumers.
- A *reference point* (RP) classifies the relationship between business roles.

TINA defines a set of business roles. In comparison, OSA/Parlay defines two business roles, an enterprise operator, and a connectivity provider. The first step outlined in the previous chapter in order to create a consumer interface for Parlay is to *"define a complete system of administrative domains and business roles within the OSA/Parlay application layer"*. Thus, the TINA defined business roles may be useful in accomplishing this goal. Below is a summary of the defined business roles:

- The **Consumer** role is performed by a stakeholder who consumes services provided in a TINA system. A consumer may be an individual, a private household, small or large business, and all kinds of companies. The broker and retailer roles provide services for consumers to use [4]. This definition of a consumer role can also be used within the Parlay application layer to represent the consumers.

- The **Broker** role provides stakeholders with information that enables them to find other stakeholders (administrative domains) and services in the TINA system.
- The **Retailer** role provides stakeholders in the consumer role with access to services. This is the key role in the TINA services environment. The major responsibility of the retailer is to provide the unified management of services, in terms of subscription and accounting. Retailers act as intermediaries for service and connectivity providers in order to present a single point of contact to the consumer. However, it is not uncommon for the retailer and service provider role to be managed by a single stakeholder. In telecommunications, often telecom operators bundle their own services such as video/voice conferencing, and at the same time provide retailer services by managing user subscription, and access to services.

  In OSA/Parlay, an enterprise operator would interact with a retailer to provide services to a consumer domain. We suggest that the TINA retailer be incorporated into the OSA/Parlay business model to provide the functionality required by an enterprise operator to interact with the consumer domain to provide services.
- The **Third party service provider** aims to support retailers or other third party providers with services meant for use by consumers. An example of a 3rd party service provider could be a music provider who does not sell music directly to consumers, but instead offers its products through a music retail store. This role is comparable to OSA/Parlay's enterprise operator.
- The **Connectivity provider** manages the underlying transport network. The connectivity services are not directly offered to consumers, but have to go through retailing. In Parlay, the framework and SCFs would fulfil this role.

Figure 2.3 illustrates the relationship between the different administrative domains, while at the same time highlighting the reference points between domains. Note that any given company may be a combination of any or all of these five domains. The consumer, retailer, and third party service provider business roles are mostly within the scope of the TINA-C service architecture. The business relationships of these roles with the connectivity provider are within the scope of the network resource architecture [13]. Although TINA-C assumes a one to one mapping of business roles such that each reference point shown in figure 2.3 should have its own documentation, the only RP which has been completely defined and standardized by TINA-C is the Ret-RP between Customer and Retailer.

We therefore focus on the Ret RP and the consumer and retailer business roles. These roles provide the necessary high level functionality required by OSA/Parlay application providers to deliver services to end users. Using the Ret RP, TINA defines the following high level requirements for the *consumer* business role [24]:

1. obtaining location of retailers, service providers and other consumers.
2. initiating service relationships that include service providers and other consumers.
3. indicating availability to the service provider (for receiving invitations).

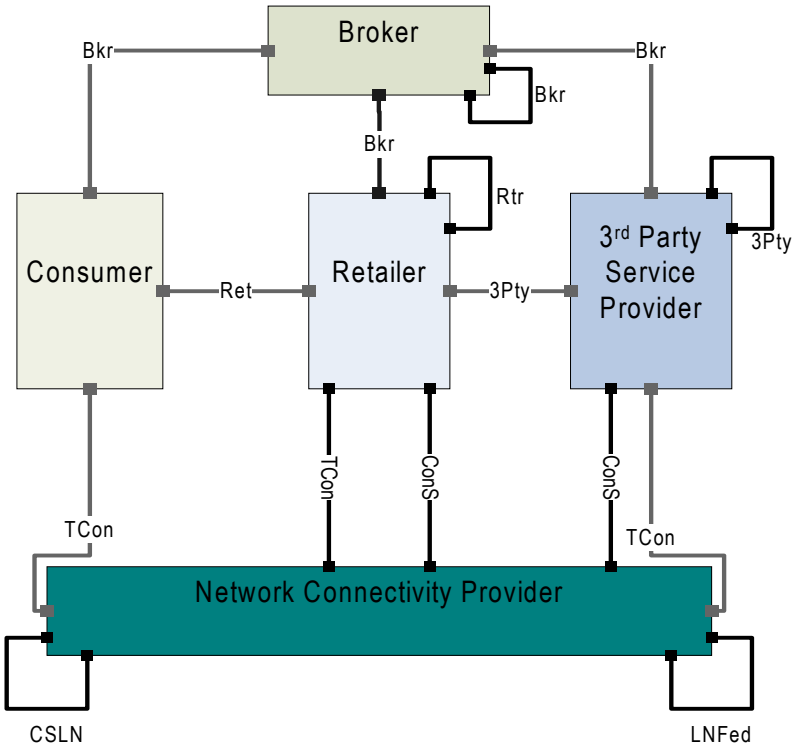

Figure 2.3: The TINA-C Business Model

And for the *service provider* role, the following high level requirements are defined:

4. manage (de)registration to obtain various services by consumers.
5. authorization prior to usage.
6. maintenance of session-level user service profiles and treatment policies.
7. session management, communication to establish and maintain the association list of parties and resources that partake in a session with session owners and session policy information for the purpose of establishing access to the session.

In this research, the first step in solving the main problem is to define a relationship between the consumer and service provider domains within the Parlay architecture. TINA's Ret-RP provides a well defined set of APIs to support the interaction between a retailer and a set of consumers. It can be used to provide the consumer to service provider API set required in the implementation of an end user interface for Parlay. It is therefore suggested that TINA's Ret RP to be used to define the consumer to service provider API set. The next section describes the computational model used to support the TINA Ret RP.

2.2.2 TINA Service Components and Interfaces

In order to realize the relationship specified by the Ret RP, TINA provides a set of service components that can be used in service creation and management. The service components provide functionality which can be manipulated to support service delivery between a consumer and provider domain. These components are distributed across different network elements and administrative domains. They provide access, service and communication session related functionality.

The components are separated according to their business administrative domain and session type. Access related components support interfaces which provide universal access to services. Usage related components support interfaces which allow users to use a service. Figure.2.4 illustrates the possible grouping of the service components within the end user interface in relation to Parlay's enterprise operator/3rd party application provider domain. Their location in terms of the retailer, service provider, and consumer domains is also shown. It is important to note that figure 2.4 utilizes the TINA business model to illustrate the possible arrangement of a hybrid OSA/Parlay-TINA service architecture. The main aim here is to illustrate the positioning of the service components within the TINA system (ie. in relation to the consumer, retailer and 3rd party service provider domains). However, Section 2.5.3 provides a formal description of the proposed business model, as well an illustration of the system concept based on that model. It must also be noted that only the most relevant service components to this research are mentioned. The list of components and supported capabilities presented here is by no means exhaustive. However, the TINA-C Service Architecture specification [10], presents a greater number of service components, and each in greater detail. The following sections present detailed descriptions of the TINA service components and their purpose.

2.2.2.1 Access Session User Application (asUAP)

An API to control end user Access and Authentication to OSA/Parlay applications is not specified within the Parlay standard. Furthermore, end users have no mechanism through which they may discover available service from a service provider. Authorized access to a service provider and their services is necessary to allow for the setup of a secure service usage context between a consumer and service provider. One of the main objectives in this research is to provide this functionality.

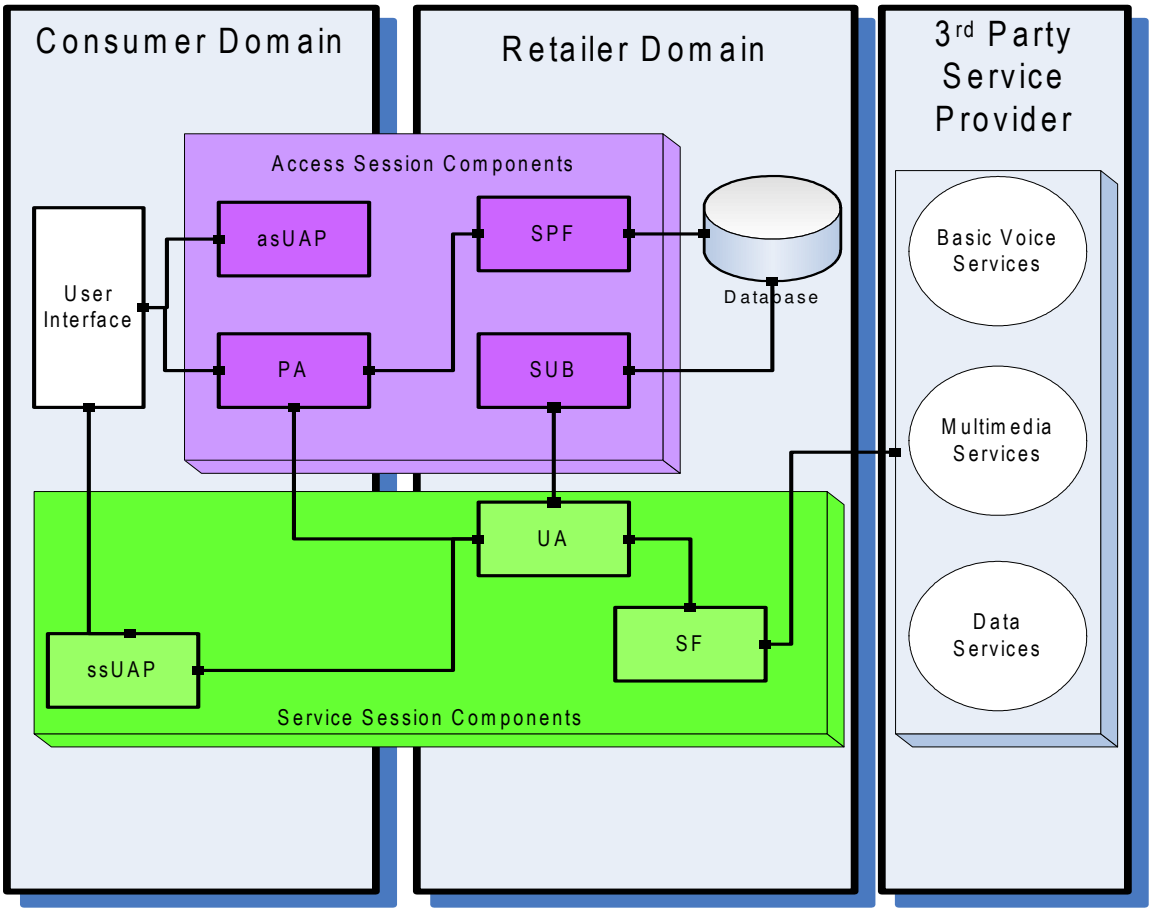

Figure 2.4: The service components in the TINA service architecture

In TINA, an access session related User Application is located in the consumer domain and is the first point of contact for a user. It supports access session setup and authentication. The asUAP is defined to model a variety of applications in the User domain. It represents one or more of these applications and programs and can be used by human users, and/or other applications in the user domain. A UAP can be either or both an access session related (asUAP) and service session related (ssUAP) service component. The asUAP provides capabilities for:

- the user to discover a service provider.
- the initiation of authentication of an individual user and/or a consumer domain.
- the user to discover available service offerings from a provider.
- the user to request the setup of a usage context.

- the user to request service and user information.

The asUAP collaborates with the PA to provide an Access and Authentication management system for end users within the consumer domain. In conjunction with the SPF and the UA, these components provide a complete Access and Authentication management system within the TINA service layer. The main function of the asUAP is as a 'first contact' to the user to support the initiation of access and authentication requests. The PA is discussed next.

2.2.2.2 Provider Agent (PA)

The Provider Agent (PA) is defined as the user's end-point of an access session within the consumer domain. The PA supports a user accessing its UA and making use of services. The following capabilities are supported by the PA:

- convey request for discovering a service provider.
- set-up a trusted relationship between the user and the provider through authentication (an access session between the consumer and service provider domain).
- convey request for discovering available service offerings (either anonymously or as a recognized user).
- support usage context setup.
- convey request for service and user information.

2.2.2.3 User Agent

A User Agent (UA) represents a user in the provider's domain and supports the user in all his/her interactions with the service provider. It is the provider domain's end-point of an access session with the user. The UA supports the following capabilities:

- support the setup of a trusted relationship between the user and the provider (an access session). The provider may or may not know the identity of the user. (In the case of anonymous users 'Trust' is not guaranteed by identifying the user, but may be ensured by, for example, pre-payment.).
- convey request for discovering available service offerings (either anonymously or as a recognized user) to the Subscription manager (SUB).

- setup a usage context for a user.
- convey request for service and user information to the SUB.
- manage the user's preferences (choices or constraints) on service access and service execution. (These would be determined during access session setup).
- provide access to a user's user profile. (The user profile is a dynamic object which contains all information that is used directly within the access session for authorization decisions, constraints and customization of the access session. It is defined at the start of the access session and terminated at the end of the access session.).
- resolve the service execution environment for the user through the user profile, allowing him/her to use services from many different types of terminals. This requires resource configuration information of the user system (which includes terminals and their access points being used by or available for the user; access to this information is restricted by the user.) This includes support for personal mobility.
- manage invitations received by the user according to policy. This includes support for personal mobility.
- accept invitations from users to join a service session.
- deliver invitations to a terminal, previously registered by the user.
- allow the anonymous user to register as a user of the provider (i.e., set-up a contract with the provider for longer than a single access session).
- convey user requests during subscription management to the SUB.

2.2.2.4 Service Provider Framework (SPF)

No standardized set of Parlay interfaces have been defined to guide the interaction between the application server and end user applications seeking to access and use 3rd party applications. The addition of the SPF is meant to support these activities, however, an API to access the application server remains undefined. It is therefore suggested that a similar point of access to the application server as the Parlay gateway's framework API be implemented. In this section, we introduce the Service Provider Framework (SPF) and define its functionality as required.

The Service Provider Framework is modelled after TINA's IA service component and OSA/Parlay's Framework and is intended to support the access and usage of 3rd party application services by end users implementing the TINA based end user interface. Its main functions are:

- authenticate the requesting domain and set up a trusted relationship between the domains (an access session) by interacting with the PA.
- establish an access session, but allowing the requesting domain to remain anonymous when the user is unknown.

Besides access and authentication management, the SPF can be extended to support the deployment, configuration, activation, deactivation and withdrawal of service instances. It is important to note however that the only the SPF's access and authentication management functionality is implemented in this project.

2.2.2.5 Subscription Manager (SUB)

One of the objectives outlined in Chapter 1 was the provision of a Subscription and Profile management system for Parlay. This was due to the fact that Parlay offers no API support for subscriber management. The standard does not specify how a service provider/enterprise operator manages the online or offline subscription of consumers to services. In order to authorize access to the service provider domain and the services offered therein, as well as to set, monitor and enforce limits on the use of those services based on some policy, it is necessary that a service provider be able to differentiate between users and their assigned rights and privileges in terms of service usage. TINA defines a subscription component which can be used for the management of subscribers, subscriptions, and users for a whole set of services provided by a service provider.
The Subscription Management Component (SUB) can be considered as the provider domain's control point of subscriber, user and subscription lifecycle. The SUB, working together with the UA, provides a complete system for user and service profile management within the provider domain. The main functionality offered by this component is:

- creation, modification, deletion and query of subscribers.
- creation, modification, deletion and query of subscriber related information (associated end users, end user groups, etc.).
- creation, modification, deletion and query of service contracts (definition of service profiles).
- retrieval of the list of services, either the ones available in the provider domain or the subscribed ones.
- retrieval of the service profile for a specific user (or terminal or NAP).

To achieve the outlined objective of providing Subscription and Profile management functionality to the Parlay service architecture, it is important that the SUB component be incorporated into the consumer interface.

2.2.2.6 Service Session User Application (ssUAP)

The consumer's initial point of contact during service usage is the Service Session User Application (ssUAP). The ssUAP supports all user interactions with the service provider outside of an access session. It represents a variety of applications in the User domain that interact with OSA/Parlay services and support the service session. The service session UAP is service specific and logically supports a combination of session controls;

- initiating/terminating the session.
- inviting other participants to join.
- joining an existing service session.

The main reasons why we use the ssUAP instead of the asUAP in service session related interactions are:

- Separation of concerns (access session vs. service session).
- Service specific downloads -- the ssUAP may be required to download service specific data (ie. codecs) to run a service.

2.2.2.7 Service Factory (SF)

A Service Factory (SF) is a service-specific object, which manages the lifecycle of service session computational objects (i.e. App Call Manager, App Call Object and App Call Leg) for an OSA/Parlay service type. The SF is critical in achieving the outlined objective of initiating and terminating Parlay applications. The SF supports capabilities to create and delete these objects. A request to create a service session of a particular service type (ie. generic, multiparty, multimedia, conference) results in the creation of one or more object instances. The SF can create and initialize the instances according to rules imposed by their implementation under the OSA/Parlay specification. Requests to create new service objects are typically made by an OSA/Parlay Application (App).

2.2.2.8 OSA/Parlay Applications (App)

Within the system of TINA service components, OSA/Parlay applications are represented by the App service component. This object encapsulates the service logic required for a functioning application. Depending on the type of call control functionality it requires, an

App has a different set of call managers, call objects, and call legs. The App supports the capability to:

- Request the instantiation of OSA/Parlay call objects.
- Request the termination of OSA/Parlay call objects.
- Request connection setup from the OSA/Parlay gateway.
- Request connection teardown from the gateway.

This section presented a set of TINA service components used to encapsulate the functionality provided by the TINA's Ret Ref Pt. The Ret Ref Pt defines an API set which can be used to manage the provision of services between a retailer and consumer domain. It is proposed that the service components presented be used to implement an end user interface utilizing the Ret RP APIs to support consumer-provider service provision in OSA/Parlay. Figure 2.4 illustrated the proposed arrangement and the roles of each component have already been described. The functionality provided by the service components can be used to achieve the objectives outlined in Chapter 1. The asUAP, PA, SPF, and UA, provide the necessary functionality to implement an Access and Authentication management system for the end user interface. The service session or access session UAPs, UA, and SUB provide the necessary functionality to implement a Subscription and Profile management system. The initiation and termination of Parlay applications can be facilitated by the ssUAP, UA, and SF. The next section presents the TINA Ret RP APIs.

2.2.2.9 TINA Interfaces

To provide the functionality specified for each service component, the Ret RP specifies a set of interfaces. These interfaces are generally grouped according to their session type. The session concept is introduced in the following section. In this section we present a general overview of the Ret RP APIs used in this research, a more detailed description is provided in the relevant design chapters. The following interfaces are defined for the Access Session:

- i_Initial

  Allows a consumer to contact a provider as well as to request authentication.
- i_Authenticate

  Supports user and/or domain authentication.
- i_Access

Provides consumers with access to services once they have been authenticated. The user may view available services and start service sessions using this interface.

The following interfaces are defined for Primary Service Session:

- i_Access

  In terms of service management, this interface can be used to initiate a service session.

- i_SSManage

  Supports the creation and deletion of service related objects.

- i_Invitation

  Supports user invitation scenarios.

- The Basic Feature Set of Interfaces

  A limited version of this feature set is used in this project. The functionality utilized allows a consumer to terminate a service session.

- The Multiparty Feature Set of Interfaces

  Allows consumers to initiate and terminate multiparty service sessions.

- The MultipartyInd Feature Set of Interfaces

  Supports the initiation and termination of multiparty service sessions.

The following interfaces are defined for Ancillary Service Session:

- i_Subscribe

  Allows consumers to subscribe and unsubscribe to services.

- i_SubscriberInfoMgmt

  Allows subscribers to manage their information.

- i_ServiceContractInfoMgmt

  Allows subscribers to create a service contract and to manage information related to it, such as service templates and service profiles.

- i_SubscriberInfoQuery

  Allows a subscriber to query information about end users currently associated with the subscription.

The following section introduces the session concept.

2.2.3 The TINA Session Concept

One of the core concepts arising from the TINA service architecture is the definition of sessions. Sessions can be useful as a means for grouping specific activities between end users of OSA/Parlay services during a specific period of time. Three types of sessions are defined, the access, service and communication sessions. The interactions required to discover and request services are captured under access. On the other hand, those that control service behavior or deliver stream content are captured under usage. Stream content delivery supported such as QoS and connection setup is provided by the communication session [4].

2.2.3.1 Access Session

In the TINA architecture, all interactions between a user and a provider are executed within the context of a session. The architecture distinguishes between an access session and a service session. The access session is used to initiate a secure dialogue between two domains through user authentication. Authentication takes place using agreed upon algorithms between the authenticating domains (consumer and provider). The access session also allows for the setup of user context information such as terminal configuration (primarily for service usage management), as well as the discovery of provider service offerings. After the access session is successfully completed, the user can start a service session in which he can select one or more services to use. The access session maintains state about the user's association to the provider and about his involvement in a service session. A user may be involved in many service sessions at the same time, and an access session maintains state about his involvement.
The asUAP, PA, SPF, and UA enable the realization of an access session within the TINA system. In this research, we intend to explore their suitability in providing access session functionality within an OSA/Parlay system.

2.2.3.2 Service Session

The service session can be used by a consumer to interact with active OSA/Parlay services. It represents a single activation of a service and encapsulates information and functionality required to activate, control and manage services. A service session is decomposed into a usage service session, which provides a confined view for each user

of the service session, and a provider service session, which holds a universal view. The usage service session represents the across domain relationship between two domains. The provider service session contains service capabilities common to multiple members of the service session, and represents the core service logic necessary to execute service requests while maintaining the session for multiple participating domains [4].
TINA differentiates between primary and ancillary service sessions. In this context, a service invoked in a primary service session can be characterized by a multimedia conference, and an ancillary service, an online subscription. Primary service sessions will generally require that Parlay applications request connection setup from the gateway. In such instances, the SCFs can be used to provide generic or multiparty multimedia connectivity. Users in a service session have the necessary information and capabilities to negotiate Quality of Service (QoS), security contexts, user contexts and the use of service and communications resources during service provision [14]. Ancillary services are separated architecturally because they are an obvious source of competitive differentiation among providers. By categorizing them as services, it becomes possible for a service provider to manage their deployment and provision to users in a manner consistent with that of primary services [23].
TINA also defines number of feature sets (FS) and interfaces which can be used to provide generic service session functionality. The complexity of the service offered by a service provider determines the feature sets that the domain must support. For example, within a FS, interfaces providing single party or multi party call capabilities are defined. Parlay application requiring multiparty functionality within a service session must implement the TINA multi party feature set. Feature sets are discussed in more detail in Chapter 3.
The ssUAP, UA, SF, and SUB enable the realization of a service session within the TINA system. In this research, we intend to explore their suitability in providing service session functionality within a Parlay system.

2.2.3.3 Communication session

A communication session provides a service-oriented view on the end-to-end stream connections in the transport network between the participants of a service session. Typically, the service session specifies a stream connection to be set up between parties in terms of QoS (quality of service) parameters and abstract medium descriptions. The communication session is not relevant to this research as connectivity within our model is provided by the OSA/Parlay gateway. End to end stream connection setup is handled by the SCFs within the Parlay model. Therefore, we only explore the feasibility of integrating TINA's access and service session model into the OSA/Parlay service architecture.

Figure 2.5 illustrates the lifetime dependencies between sessions for which the following general principles can be applied [4]:

- A service session cannot exist without the access session of the party holding the ownership of the service session (ownership can be transferred between session parties).
- An access session can encompass many service sessions.
- A service session can encompass many communication sessions.
- A communication session cannot exist without a service session.
- A communication session can only be controlled by a single service session to avoid control conflicts.

The session concept is important because it separates the different concerns of service access and usage, and promotes the distribution of service functionality [20]. The separation of the provider and usage service session supports the distribution of service logic whereas, the separation of access session and service session allows the access and service management methods to vary based on changing user context.

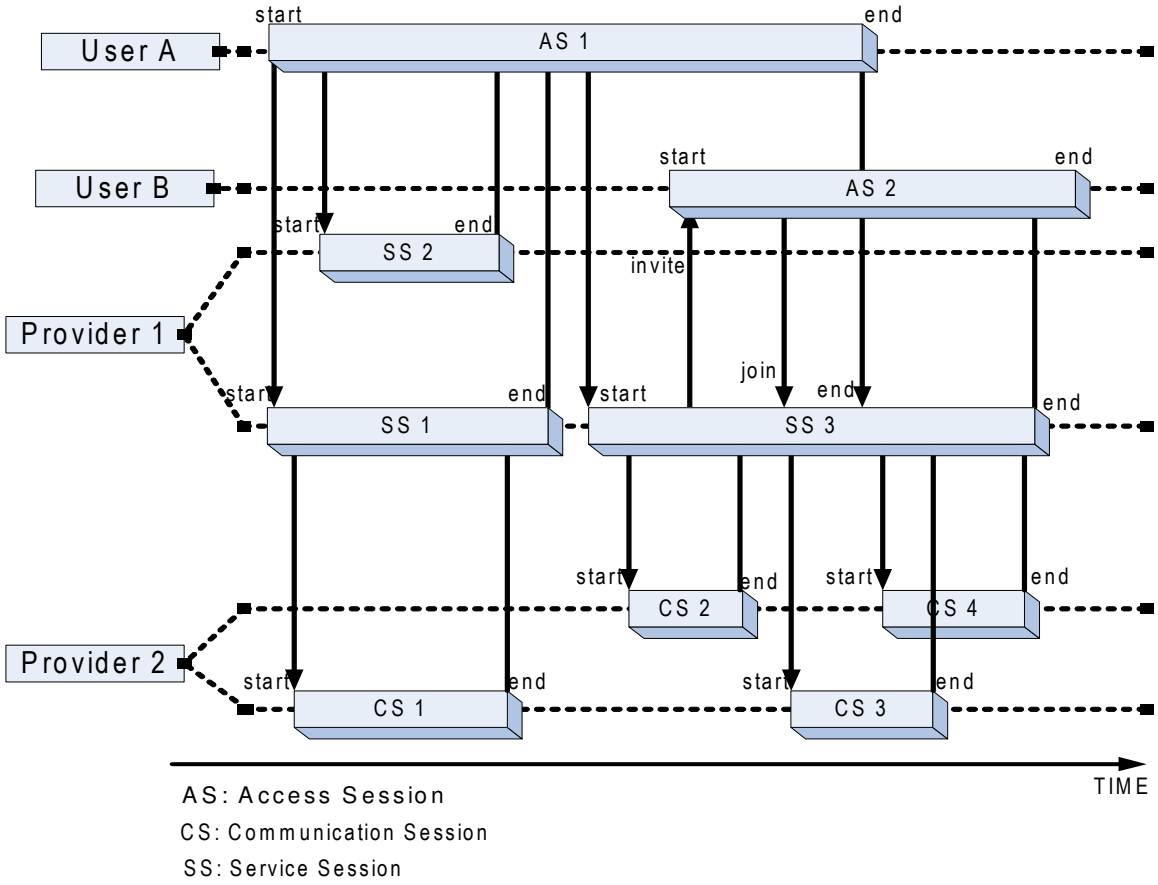

Figure 2.5: Lifetime dependencies among sessions. Redrawn from [4]

Service delivery requirements can therefore be tailored to meet the changing needs of a user. For example, usage context such as location, device type, or date and time, may influence service access and usage aspects such as service discovery or registration, authentication, and QoS [21].

2.3 TINA and Context Awareness

The TINA Service architecture supports context aware service delivery by defining data structures and types that are user context related. Below is a listing of the different types of context and the TINA data structures that are used as enablers:

Network Characteristics: TINA provides no explicit support for service delivery based on network characteristics [14].

Device Characteristics: TINA defines terminal related data structures *t_TerminalConfig* and *t_TerminalInfo* that enable service delivery based on device characteristics.

The *t_TerminalConfig* is a structure containing the terminal id and type, the network access point id and type, and a list of terminal properties, *t_TerminalInfo* and *t_ApplicationInfoList*.

The *t_ApplicationInfoList* provides a list of the applications that are available on the terminal.

The *t_TerminalInfo* gives details on the type of terminal, the operating system type and version, the network cards available on the terminal, the maximum number of network connections which can be supported by the terminal, as well as memorySize (amount of RAM in megabytes), and the disk capacity in megabytes.

It is important to note however that TINA has no mechanisms to convey and manage (on-line) information about current terminal capabilities from the terminal to an application without querying the consumer domain [4]. TINA can only retrieve information about terminal capabilities directly from the consumer. Unlike OSA/Parlay, no functionality to retrieve terminal capabilities from the network exists.

User Context: TINA defines *t_UserCtxt*, which is a data structure which is used to pass user context to the retailer, as soon as an access session has been established.

- **User Preferences**

  TINA subscribers can define their preferences through policies selected during the negotiation of a *service contract* (which describes the terms of provision of a service). Using a *subscription service profile*, subscribers are able to tailor a service's behavioural characteristics to their specific requirements and needs based on a service specific template [10]. End users are granted with

customization capabilities through the definition of a *user service profile*, constrained by the subscriber's preferences.

- **User State**

  The *t_UserCtxt* data structure informs the service provider about consumer domain information such as available interfaces during the access session.

2.4 TINA and Personal Mobility

Personal mobility as defined in Chapter 1, is important in providing services to end users at any terminal. The UPT definition **Error! Reference source not found.** requires that users be able to access services from any terminal and the network to be able to support the provision of those services to any terminal. Currently, the OSA/Parlay architecture has no standard API to enable personal mobility. The architecture lacks the capability.
The TINA service architecture on the other hand provides inherent support for personal mobility. However, in order to enable personal mobility within the OSA/Parlay network, TINA specifies the following requirements [4]:

- Users should be able to access the system from any terminal, subject to restrictions imposed by the service provider and by the service subscriber. To "access the system" includes: to initiate access sessions, to initiate service sessions, and to receive invitations to service sessions at a specified terminal.
- Users should have the ability to register so that service invitations are sent to a terminal specified in the registration. That is, the registration will result in invitations to join a service arriving at the terminal specified.
- Conditions may be applied to this registration, for instance, a user may register at different terminals for different services, different callers, different times of day, etc. The architecture should allow the support of this conditional registration, but it is a service provider issue to define and implement the actual conditions that are supported.
- Users may respond to a service invitation on a different terminal than the one specified for getting the invitation.
- The user should be able to "access the system" according to the service provider's and user's own preferences, as much as possible, independent of the terminal used. The delivery of a service is subject to terminal and network capabilities. For example, a user may not register a POTS telephone for a video service as it does not have the capability to receive video streams.

By adhering to these requirements when implementing the end user interface, personal mobility can be integrated into the OSA/Parlay service architecture.

2.5 Overall Contributions of TINA to OSA/Parlay

TINA defines an architecture for telecommunication applications. A major limitation of the TINA-C effort is that it mainly takes into account advances in distributed computing technologies, aiming only at supporting highly intelligent terminals in the NGN without taking account of backward interoperability with the legacy service platforms [14,17,19]. Even though TINA ideas are not sufficient for all kinds of services, the architecture still provides a comprehensive set of concepts and principles that may be useful in the design and implementation of an end user interface for OSA/Parlay applications.

2.5.1 What support can a TINA Derived Consumer Interface give to Parlay Applications

The proposed end user interface can provide OSA/Parlay applications with the following:

1. A logical separation of roles in the application and SCF layer to support the provision and deployment of OSA/Parlay services. It provides a model where Consumer, Service Provider, and Connectivity Provider are logically separated into different administrative domains, and the responsibilities of each domain are defined. The OSA/Parlay service architecture does not define any responsibilities for any roles other than the service provider and the connectivity provider.
2. The specification of a consumer provider relationship to govern service access and usage within the Parlay system. The TINA Ret Ref. Pt provides a highly defined description of the responsibilities of the service provider and consumer domain within an interaction.
3. An integration of the session concept into the OSA/Parlay service architecture. The session concept replaces the traditional call and connection concepts of telecommunications, and is more suited to multimedia and multi-party services [12]. The session model clarifies the purpose of the interactions and interfaces between a user and a provider.
4. The specification of a generic service independent API set (TINA Ret RP) between the consumer and service provider domains to support the provision and deployment of OSA/Parlay services.
5. The definition of a component specification which simplifies service creation by encapsulating the functionality provided by the Ret RP API into reusable and interoperable service objects. This separates service independent logic from service specific logic. Interfaces for Authorization, Subscription, Profile, Usage, and Access management are detached from service specific logic and placed onto

a single service delivery platform for use by all services in the Parlay enterprise operator domain. This decreases the development time for OSA/Parlay applications because the application provider may reuse the standard service components and need only develop the application specific logic when creating new services.

6. The definition of an API and component specification to support Access and Authentication management.
7. The definition of an API and component specification to support Subscription management.
8. The definition of an API and component specification to support Service Usage management, specifically the initiation and termination of OSA/Parlay services by end users.
9. Support for context awareness through the definition of the UA and SUB. The UA has the capability to setup a usage context, as well as make service delivery decisions based on a user profile which amalgamates a user's preferences, and usage context. The SUB provides facilities for a user to register their preferences.
10. The provision of inherent mechanisms to support personal mobility.

Other benefits of a standard OSA/Parlay consumer interface include:

- Standardized Service Access and Usage.
- Elimination of Duplicated Functionality in the Service Provider Domain.
- Reduced Software Cost and Maintenance.

2.5.2 What can it allow users to do?

The proposed end user interface would be made up of a set of TINA APIs and service components designed to provide essential functionality not only to OSA/Parlay applications but to the end users in the consumer domain. Access session and Service session functionality is separated to simplify the interaction between users and service providers. Using TINA access session scenarios as a guide we propose that the end user interface's access session APIs would allow users in an OSA/Parlay environment to:

- Locate and contact a service provider.
- Login and logout of the service provider domain.
- Setup the default context required for control of the service session.
- Discover service offerings.

- Terminate an access session.
- Terminate active service sessions.
- Join a service session from the access session.

The Service Session APIs would allow users to:

- Initiate and terminate a service.
- Invite a user to join a service session.
- Join an existing service session with an invitation.

The Supplementary Service Session APIs would enable users to perform:

- Subscription Management (subscribe, modify, and release services offered by providers).
- User and Service Profile Management (create, modify, and delete profiles).
- Query subscriber, service, and profile information.

2.5.3 Proposed Service Architecture

This chapter reviewed the OSA/Parlay service architecture and summarized the capabilities offered by its SCFs as well as the types of services it can support. We also took a look at the Parlay business model where two roles, an enterprise operator and a connectivity provider, were identified. It was then noted that the Parlay standard does not define a consumer role within the business model. The problems caused by the lack of a consumer interface were then summarized. We then reviewed the TINA service architecture to determine whether it could provide Parlay with the necessary support to define a consumer domain and the functionality required for its realization within the Parlay architecture. The main results of the TINA review was the definition of a business model, an consumer/provider API set, a component model, and a session model which could all be used to support the definition and implementation of a consumer interface for Parlay applications. Support for context awareness and personal mobility was also discovered. We then illustrated a system where TINA's service components and Parlay applications were combined to form a hybrid service architecture which incorporated TINA's consumer and retailer domains into the Parlay business model.

In this section we formally propose a service architecture to support the implementation of Parlay's consumer interface. Figure 2.6 illustrates the logical separation of roles. The main difference from the figure.2.4 is the addition of the retailer domain into the enterprise operator domain. This is meant to more closely align our business model with that provided by the Parlay standard. In the remainder of this report, the retailer,

enterprise operator, and service provider terms will be used interchangeably, except when referring to non-retailer elements or API such as the Parlay applications.

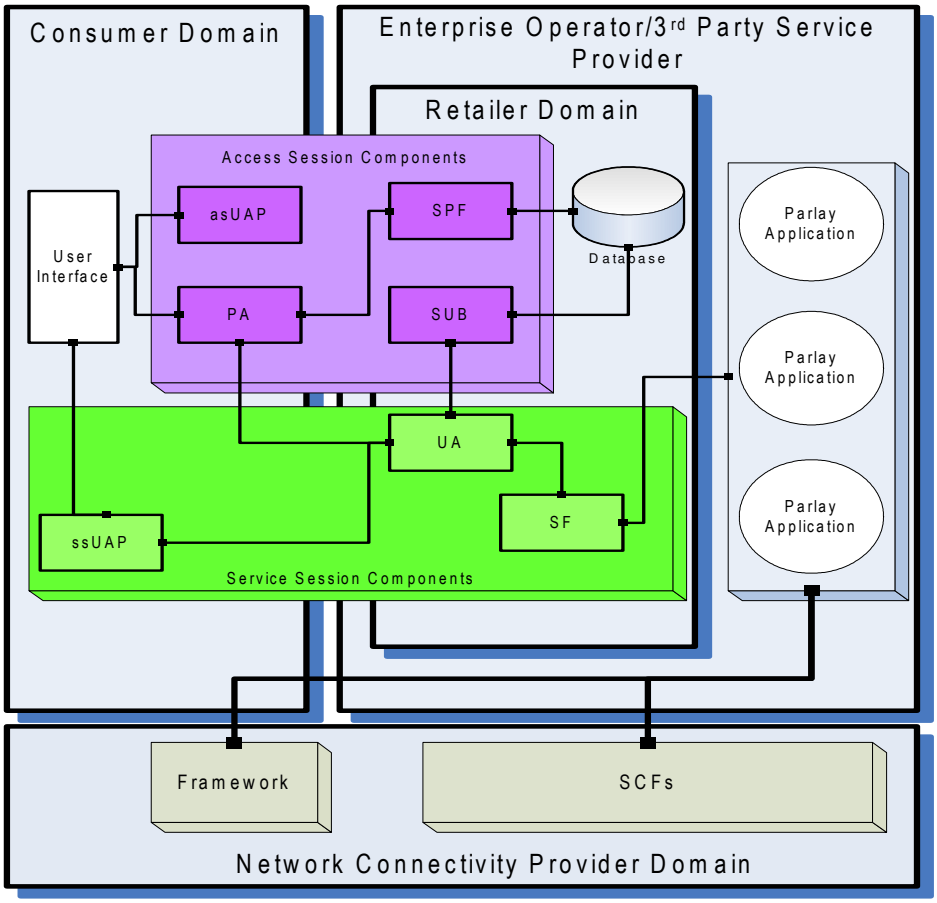

Figure 2.6: The Hybrid OSA/Parlay-TINA Service Architecture

The consumer domain consists of an asUAP, ssUAP, and a PA which are all collocated within the user's terminal. This means that the architecture only caters for consumer devices which are DPE enabled.

The next chapter presents the requirements for the design and implementation of the consumer interface. The chapter is divided up based on the functionality specified in the research objectives.

3 REQUIREMENTS SPECIFICATION FOR THE OSA/PARLAY END USER INTERFACE

This chapter presents a high level requirements analysis of the OSA/Parlay end user interface. Requirements for the design of Access and Authentication, Subscription and Profile management, and Service Usage management APIs are presented. These APIs are based on interfaces and operations defined in the TINA Retailer Reference point specification [23]. We also take a look at context aware service delivery and discuss what is required to integrate them into the consumer domain interface. The requirements for the integration of personal mobility into the consumer interface were discussed in the previous chapter (see Section 2.4).

3.1 Requirements for Access and Authentication Management

The access and authentication APIs cover the interactions required for the consumer and service provider domains to establish a service usage context. To establish a service usage context, the service provider requires the consumer to be authenticated and thereafter to maintain information about their context. The access and authentication APIs should provide [4]:

- **Location transparent discovery** of a particular service provider and their services.
- **Customized access** to OSA/Parlay services, the interaction between the service provider and consumer is customized by previously arranged role specific profiles, taking into account the end user's preferences or terminal capabilities.
- **Mobility**, ubiquitous access to the system of OSA/Parlay services, irrespective of the terminal being used and the point of attachment to the network.
- **Secure access**, the means to create a secure binding between the consumer and service provider.

Chapter 4 presents a design of the Access and Authentication APIs based on the above design goals. A detailed description of how each requirement is fulfilled is given.

3.2 Subscription and Profile Management Requirements

In order for service providers to offer services to end users, it is essential that they have sufficient information to handle end users, subscribers, and the subscription life cycle. The Subscription and Profile Management API is expected to fulfil the following high level requirements [4]:

Management of subscriber related information

- Addition or removal of subscribers.
- Query and modification of subscriber related information.
- Assigning rights and/or privileges to subscriber related entities (e.g. end users, terminals, or Network Access Points (NAPs)).
- Removing assigned rights and/or privileges to subscriber related entities.

Management of service related information

- Creating, modifying, and deleting service profiles. This includes addition and/or removal of subscribed services from a service profile.
- Retrieving service profile and service contract information.
- Query and modification of service related information.

This section presents a high level view of the basic requirements of the Subscription and Profile Management API. Chapter 5 presents a detailed description of their design.

3.3 Service Usage Management Requirements

The service usage management API's cover the interaction between the consumer and service provider domains during the use of a service session (based on policies agreed upon in the service contract). In the context of a service session, different services offer end users with differing functionality specific to the type of service. Due to a desire to not restrict the functionality offered by services, the service usage management APIs do not cover these types of service specific functions. The focus here is on providing generic service independent functionality to support consistent service usage management for all types of Parlay services.

3.3.1 Service Usage Management Feature Sets

The TINA service architecture supports service usage management by providing generic service session control methods that can be used in both single and multi-party service interactions. The generic service session control methods are categorized according to feature sets. Services requiring some specified service session functionality must implement interfaces within a corresponding feature set. This section describes the feature sets used in this research to implement the generic service session functionality used to support OSA/Parlay application usage.

The basic feature set (BasicFS) is used by all applications and provides the generic service session functionality required to initiate and control a single party session. The BasicFS is suitable for single-party applications like e-mail.

The MultipartyFS allows multiparty support for service sessions. It supports requests for multiparty control actions such as inviting a user to a service session.

The MultipartyFS supports operations to invite others to join, as well as to end a multiparty service session [23]. It supports the session providing information on events that have happened to other participants, eg. another party is leaving the session; and the session indicates to all other components that a user is about to exit (see Section 3.2.1.). In the following sections interfaces supporting multiparty operations are discussed.

The MultipartyIndFS allows indications to be sent to users in a multiparty session when an action is about to be taken shortly. For example, an indication would be sent to all users in a session when the session is about to be ended. By implementing the MultiPartyFS and MultipartyIndFS, multiparty applications such as call conferencing can be supported by the consumer interface.

Using the TINA Feature sets, the service usage management APIs will be required to provide the following service session functionality:

- Initiate and terminate single party service sessions.
- Initiate and terminate multiparty service sessions.

Chapter 6 presents a design of the Service Usage Management API.. The chapter is split into two parts, the first describes the initiation and termination of single party service sessions, and the second, multiparty service sessions. A detailed description of the implemented Feature Sets and the interfaces used to support them is given.

3.4 Context Aware Service Delivery Requirements

In order to integrate context into the delivery of OSA/Parlay services, the following requirements should be met:

- Service delivery should be adaptable based on terminal capability.
- Service delivery should be adaptable based on user preference.
- Service delivery should be adaptable based on user state.

In Chapter 6 (Section 6.3) the implementation of context awareness in consumer interface is presented.

3.5 Summary

In this chapter, we detailed the requirements for the implementation of the consumer interface. The first section outlined the requirements of the end user interface with respect to the enablement of access and authentication methods for OSA/Parlay service providers. In the second and third section, requirements for the enablement of service usage management, and subscription and profile management methods respectively were outlined. The chapter concluded with a specification of the requirements for incorporating context awareness into the end user interface.

4 DESIGN OF THE ACCESS AND AUTHENTICATION API

This chapter presents the design of an access and authentication management API for Parlay's consumer interface using TINA interfaces. The access session scenarios are supported by the enterprise operator to allow a consumer to request the establishment of an access session. They are the initial point of contact for the consumer, and allow him to authenticate himself and the enterprise operator as well as to establish the access session. The access session also allows the consumer to discover services, initiate usage of those services, and register the consumer's context with the operator. The following section provides a detailed description of the TINA Ret RP access session scenarios using sequence diagrams and an explanation of the class structures and methods used to implement the scenarios.

4.1 Access and Authentication Interfaces

To provide consumers with access to the service provider domain, the end user interface require APIs that provide the capability to:

- Transparently locate a specified service provider and their services.
- Establish secure access, the means to create a secure service usage context between the consumer and service provider.
- Provide customized access to OSA/Parlay services, the interaction between the service provider and consumer is customized by previously arranged role specific profiles, taking into account the end user's preferences or terminal capabilities. (Note that determining the profile is not part of access session functionality, but done in a specific service session).
- Provide Personal Mobility, ubiquitous access to the system of OSA/Parlay services, irrespective of the terminal being used and the point of attachment to the network.

To meet these requirements, we use the retailer reference point specification [23], which defines the required interfaces between the service provider and the consumer premises, as well as essential operations and parameters useful in these interactions. Only those retailer reference point interfaces that provided the required functionality between the consumer and service provider domains were utilized in this research.

The naming convention used in the research for interfaces is as follows; the same interfaces are usually used by both the consumer and service provider domains, however, when an interface is implemented by the service provider, the word 'Provider' is added to the interface name. For example, the ***i_Initial*** interface is known as the ***i_ProviderInitial*** interface when implemented by the service provider.

4.1.1 The i_Initial interface

The ***i_Initial*** interface is a consumer's initial interaction point with the enterprise operator. The **contactProvider()** method allows the consumer to transparently locate a specified provider using a naming service offered by the DPE. The user is required to input the name of the provider, after which a reference to the service provider's ***i_ProviderInitial*** interface is returned. Upon receipt of a reference to the ***i_ProviderInitial*** interface, a user may request for an access session to be setup using this interface.

The **requestAccess()** method allows the consumer to identify himself to the provider, and establish an access session. This method takes as input parameters a *userId*, and a set of *userProperties* which include the user's password. Depending on whether the user has already setup a secure context for service usage with the service provider (ie. through authentication), the provider returns a reference to an authentication interface (***i_ProviderAuthenticate***), or an access session identifier, *asId* and a reference to the ***i_ProviderAccess*** interface (the *asId* is generated using the **setupAccessSession()** method which is described shortly).

If the consumer has not already been authenticated, then a reference to the ***i_ProviderAuthenticate*** interface, which may be used to authenticate the user to the provider, is returned. After authentication, the **requestAccess()** method can be invoked again to retrieve the reference to the ***i_ProviderAccess*** interface.

Once a user has been authenticated, the **setupAccessSession()** method can be invoked. This method allows the service provider to generate the access session identifier, *asId* and return a reference to the ***i_ProviderAccess*** method, which is used to list and start OSA/Parlay services.

In order to provide ubiquitous access to the system of OSA/Parlay services, irrespective of the terminal being used and the point of attachment to the network, the following approach is used [4]:

- The user provides the PA with his/her user name and retailer name.

- The PA contacts a (DPE) naming service, which is able to find the interface reference of the SPF.
- The PA requests to the SPF that an access session be set-up with the UA, providing the user name to the SPF. Security procedures are executed between PA and SPF. The SPF returns a reference to the UA.
- The access session is then established. The PA can perform further requests (e.g., to start a service) to the UA.

4.1.2 The i_Authenticate interface

The ***i_ProviderAuthenticate*** interface allows the consumer and service provider to create a secure service usage context through authentication. It provides a generic mechanism for authentication which can be used to support a number of authentication protocols [23]. The methods provided by the ***i_ProviderAuthenticate*** interface may be used for mutual authentication where both the consumer and provider domains are authenticated, or one sided authentication where only the cosumer or provider domain is authenticated. It is not necessary for the authentication protocol to identify the individual consumer.

The **getAuthenticationMethods()** method provides a generic request mechanism through which the consumer domain can ask the service provider for a list of supported authentication methods. This method takes as an input parameter the *desiredProperties,* which is a list containing the properties that the consumer wishes the requested authentication method to support, and returns *authMethods*, which is a list of authentication methods which match the *desiredProperties*, and which the service provider supports. If the *authMethods* list returned is empty then the provider does not support any methods matching *desiredProperties*, or the provider does not wish to allow the consumer to authenticate using a method with the *desiredproperties*. Once the *authMethods* list is returned, the consumer domain selects a method to use during authentication.

Once a suitable authentication method has been agreed between the consumer and the provider, the consumer invokes the **authenticate()** method to transport authentication data (including the selected authentication method) to the provider. This method takes as input parameters an authentication method, *authMethod,* which is used to identify the authentication method proposed by the consumer, and *authenData*, which contains consumer attributes to be authenticated. The consumer may also request specific privileges from the provider using *privAttribReq* parameter.

The provider returns the following parameters to the user: *privAttrib* is returned in response to the request for specific privileges. If the provider wishes to authenticate the consumer further, then *continuationData* is returned with challenge data for the consumer. The user may examine their authentication status using *authStatus*. The

consumer can be in one of three states represented by *authStatus* during authentication, authSuccess, authFailure, authContinue, or authExpired (in which case the provider did not receive a response within a certain timeframe). If the consumer's status is authSuccess, then authentication has completed successfully and the user may re-invoke the **requestAccess()** method to setup an access session. If the consumer's status is authFailure, then authentication has completed unsuccessfully, and the consumer cannot establish an access session.

In the case where the consumer's status is authContinue, the **continueAuthentication()** method may be invoked. It allows the consumer to continue an authentication protocol, started using **authenticate()**, and pass authentication data to the retailer. This method returns the same output parameters as **authenticate()**, however the only input parameter passed to the provider is *authenData*.

4.1.3 The i_Access interface

The ***i_Access*** interface provides consumers with controlled and customized access to OSA/Parlay services. It provides a collection of methods through which a consumer may retrieve service, and user information as well as access session information. The user may also start a service using this interface.

Customized access to OSA/Parlay services is provided in part by the **setUserCtxt()** which allows the consumer to inform the enterprise operator about interfaces in the consumer domain, and other consumer domain information. (e.g. user applications available in the consumer domain, operating system used, etc). The method passes *userCtxt* which is a structure containing domain configuration information and interfaces. The service provider uses this method to gather context information in order to provide context awareness in service delivery. The **setUserCtxt()** method denotes the end of access session setup. If this operation is not called successfully, subsequent methods may fail. It is important to note that the information provided by the user in the **setUserCtxt()** method is not used independently to provide context aware service delivery but is rather used in conjunction with other context information provided in the user profile.

Controlled access to services is provided by the **listServices()**, and **startService()** methods. **startService()** is used to start a service session. This method is discussed in the service usage management chapter (see Section 6.1.1). The **listServices()** method allows an end user (known or anonymous) to request a list of the whole range of available services offered by a service provider. The method takes as input parameters *desiredProperties,* which is used to specify the type of services the user would like to view, and a *howMany* parameter which is used to limit the number of services which are returned by the provider. The only output parameter returned is a *serviceList*. This is used to return a list of retailer services matching the *desiredProperties* parameter.

The ***i_ProviderAccess*** interface also provides consumers with the ability to terminate an access session. **endAccessSession()** allows the consumer to end the access session. If any

service sessions exist, the consumer will be requested to terminate them first before ending the access session. The only input parameter is the access session Id, *asId*, no output parameters are returned.

Appendix A.1 provides a summary of all the interfaces, operations, and attributes presented in this chapter.

4.2 Proposed Implementation Scenarios

The following section provides detailed descriptions of the main use cases in the access and authentication process. These include *contact a service provider, login to a service provider, and logout from a service provider* as presented in figure 4.1. Each use case presented in this section is followed by a Message Sequence Chart (MSC) which shows the interaction between the service components in order to realise the use case.

The following roles are identified during an access session [4]:

- **End user** requests access to services and uses service content. Usage charges may be at no cost, or its cost may be assigned to a subscription contract (a portion of the overall charge or the entire charge according to the agreed contract). Only an end user can be invited to participate in a service. An end user can be a person, a terminal or a network access point (NAP).
- **Subscriber** administers subscription contracts on behalf of the consumer domain. Their responsibilities include: contract negotiation, arranging end users, and setting subscriber constraints on the service to limit end user capabilities. Subscribers may be responsible for payment of charges incurred by end users. A subscriber cannot administer anonymous users. A physical entity, such as a person, may be both an end user and a subscriber. The subscriber does not feature in this chapter but is included for completeness.
- **Anonymous user** is unknown or unrecognized by the access provider. An anonymous access user may initiate and utilize services that are free or where payment guarantees are supplied on-line. An anonymous user cannot assign usage to a subscriber. By invoking ancillary services such as subscription, an anonymous user may become a subscriber and/or end user. This requires permanent records in the provider domain and a capability to identify the user. This identity would allow the user to act as end user or subscriber. Anonymous users cannot be invited to service sessions because they do not have resolvable names.
- **Service provider** offers primary and ancillary services and also supplies service content to end users. The service provider retains knowledge about end users and subscribers associated with the consumer domain, such as subscriptions, user profiles and constraints. A service provider enforces certain policies in the access

session which are derived from agreements set in the subscriber contract. Policies are held in the user profile.

In the use case shown below, the consumer role aggregates the anonymous and end users.

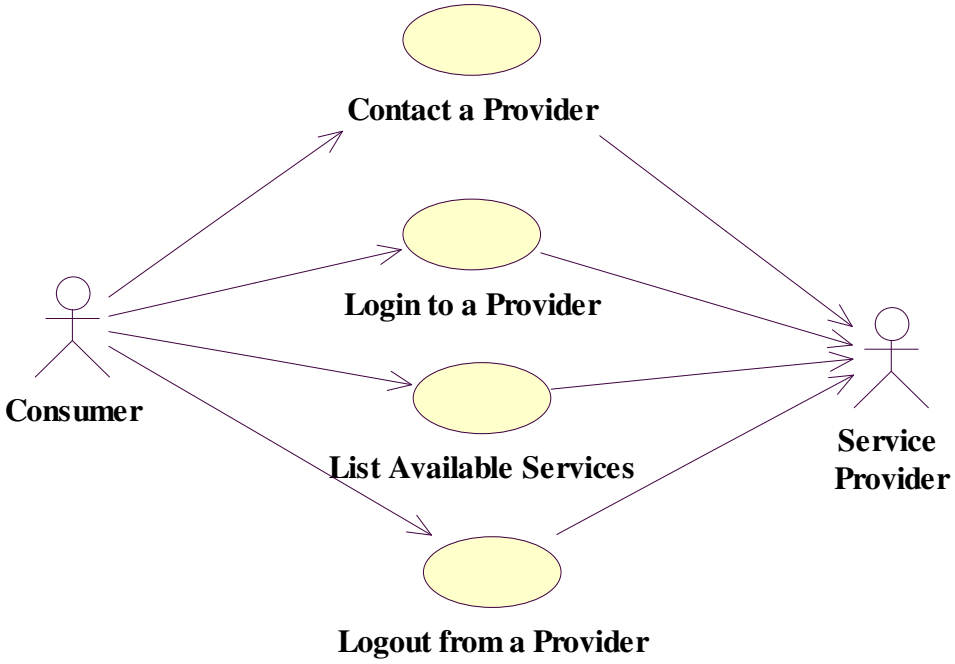

Figure 4.1: Access and Authentication Management Use Case Scenarios

To illustrate the activities described within the use case, we define message sequence diagrams. The MSC diagram shows the object interaction between service components in a chronological order. The sequence diagrams describe the service components, their interfaces, and methods. The MSC diagram for contacting a service provider is shown in Section 4.2.1.1, the login and logout from a service provider diagrams are shown in Section 4.2.1.2 and 4.2.1.3 respectively.

4.2.1 Contact a Provider

Once a user has subscribed to services offered by a particular service provider, to access their subscribed services, the subscriber must first contact the service provider. An anonymous user may also contact a service provider. However, the anonymous user may only use services which do not require subscriber access. In this use case the user may be

a subscriber or an anonymous user. The following scenario describes the steps required to contact a service provider.

Description:

A user requests to access the service provider domain by specifying an appropriate provider name (*providerId)*. On completion, the specified service provider has been contacted and the user can login using a username and password in order to access the provider's services.

Pre-Condition:

The asUAP and the PA must be present in the user's domain.

The UI must have a reference to the asUAP's ***i_Initial*** interface.

The asUAP must have a reference to the PA's ***i_Initial*** interface.

The SPF must have a reference available to its ***i_ProviderInitial*** interface.

Post-Condition:

The PA has an access interface reference to the SPF's ***i_ProviderInitial*** interface. An access session has not been setup by the user.

4.2.1.1 Message Sequence Diagram

This example shows the user making contact with a provider. This scenario supports user mobility by allowing the user to contact a specific provider from any terminal.

1. The user starts an access session related UAP by invoking the **contactProvider()** method on the ***i_Access*** interface. He supplies the provider name he wishes to contact.

2. The asUAP requests the PA to contact the provider, giving the provider name.

3. The PA gains a reference to an ***i_ProviderInitial*** interface of a provider specific Service Provider Framework (SPF) using a location or naming service. The PA returns success to asUAP. The asUAP notifies the user through the User Interface (UI).

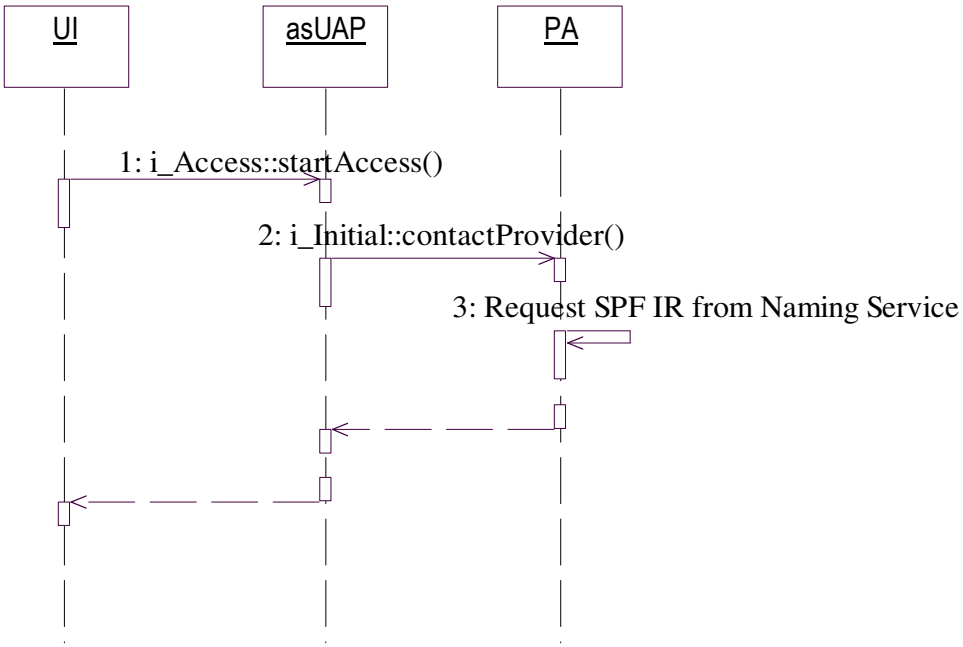

Figure 4.2: Contact a Provider Sequence Diagram

4.2.2 Login to a Provider

Once a subscriber or anonymous user has contacted a provider, they can only access services through logging into the service provider's domain. An unknown or invalid username and password automatically logs users into the service provider domain as anonymous users. In that case no user authentication is necessary, however, domain authentication must still be performed. The case below describes a scenario where the user has been registered with the service provider.

Description:

A user logs into the provider domain by specifying a valid user name and password. The user authenticates him/herself to the SPF. On completion, the service provider can correctly identify the user by a user name, and the UA has an up to date user profile containing information about the current user's domain characteristics (ie. Interfaces, user applications, operating systems, etc).

Pre-Condition:

The asUAP, PA, SPF, and UA are present in their respective domains.

Each component has made available the required access session interface references.

The asUAP has a list of authentication methods which are supported by the SPF.

Post-Condition:

The UA has generated the access session identifier, *asId*, and an access session between the PA and UA has been setup. The UA is personalized to the user and has knowledge of interfaces of the PA.

4.2.2.1 Message Sequence Diagram

1. The user uses the User Interface to request to log in to the provider. The User Interface requires the user's user name (*userId*) and password (*userProperties*). These are passed as input parameters to the ***requestAccess()*** method which is invoked on the asUAP.

2. The asUAP invokes the **requestAccess()** method on the PA. This method takes as input parameters the user name and password, and returns as output parameters, references to the PA's ***i_Access*** and ***i_ProviderAuthenticate*** interfaces, and an access session identifier, *asId*.

If the user is required to authenticate themselves, a reference to the PA's ***i_Authenticate*** interface will be returned, otherwise, the ***i_Access*** interface reference is returned and the user is allowed to setup an access session. In the latter case, the access session identifier, *asId* is also returned.

3. The PA invokes **requestAccess()** on the provider specific SPF to establish an access session that allows the user access to the provider's services. This method has the same input and output parameters as the previous invocation on the PA, except the interface references returned are to the SPF's ***i_ProviderAccess***, and ***i_ProviderAuthenticate*** interfaces. At this point if the user has not yet been authenticated, the provider will not allow an access session to be set-up, until the user is authenticated.

If the user has already been authenticated, then a reference to the SPF's ***i_ProviderAccess*** interface is returned, otherwise the ***i_ProviderAuthenticate*** interface is returned and the user is informed that authentication is necessary before an access session to the provider can be set-up.

4. The ***i_ProviderAuthenticate*** interface provides a generic set of operations that can be used to 'transport' authentication information (ie.authentication algorithms) between the authenticating domains.

In order to use these methods for authentication, both domains must know the authentication method they will use to generate the authentication data. The asUAP requests the PA to find out the authentication methods the provider supports. This method takes as an input parameter the *desiredProperties,* which is a list containing the properties that the consumer wishes the authentication method to support, and returns *authMethods*, which is a list of authentication methods which match the *desiredProperties*, and which the provider supports.

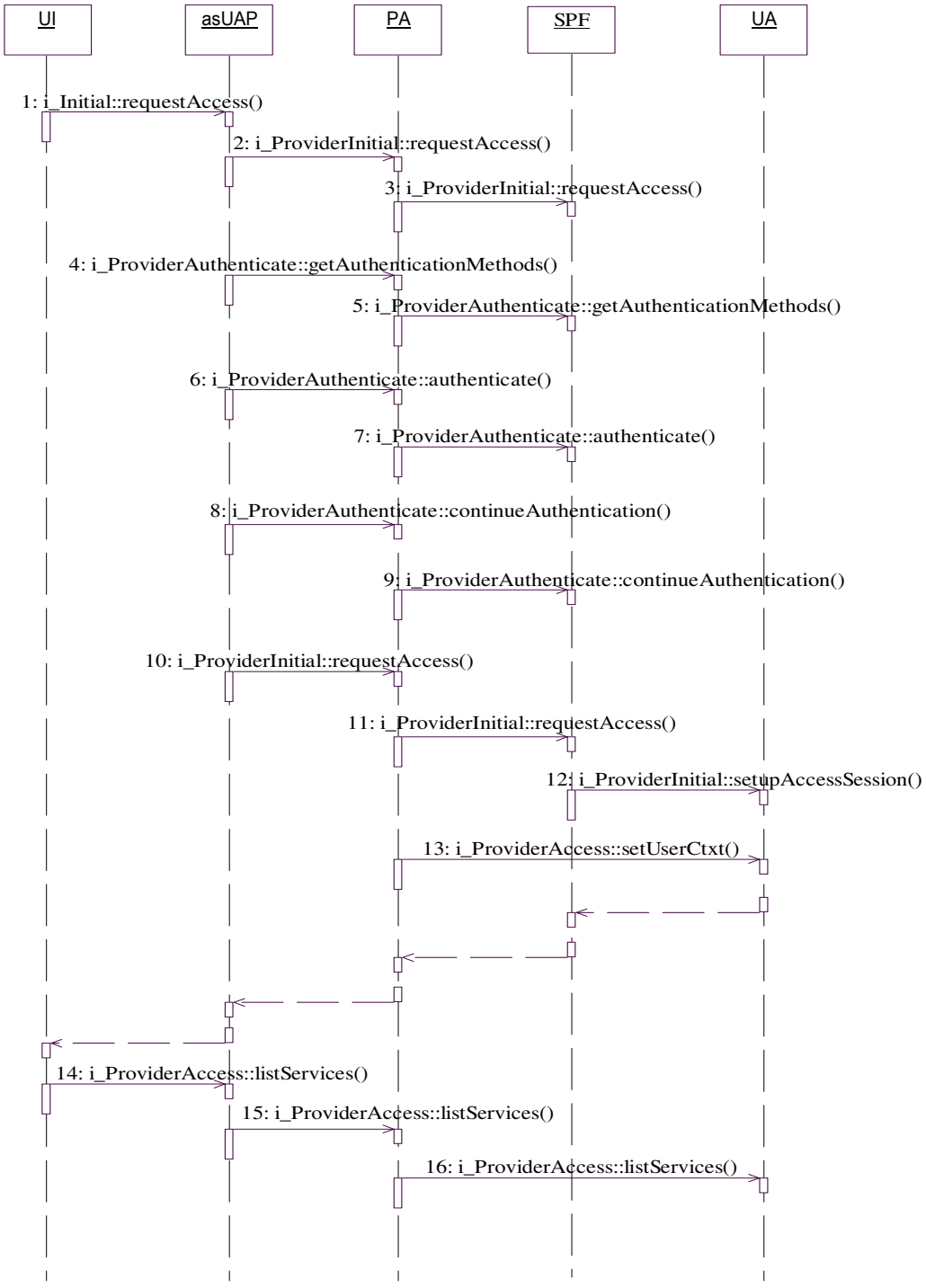

Figure 4.3: Login to a Provider Sequence Diagram

5. The PA request a list of the authentication methods the provider supports. The list of authentication methods is forwarded to the asUAP.

6. The asUAP uses the ***authenticate()*** operation to select an authentication method and pass authentication data to the PA. The asUAP may also request specific privileges on behalf of the user from the provider.

7. The PA passes the selected authentication method and authentication data to the SPF. The SPF then processes the authentication data. If further authentication is required then the SPF informs the asUAP using the *authStatus* parameter, and requests it to respond to some *challengeData*. Authentication may then proceed using the ***continueAuthentication()*** method.

8. The asUAP responds to the *challengeData*, processing.

9. The PA forwards the results to the SPF. If the *authStatus* is returned as authContinue, then the PA and asUAP must continue to process the *challengeData* returned by the **continueAuthentication()** method. This loop continues until the authentication status is successful (authSuccess), or a authFailure, or a authExpired is returned. The former means the user has failed to authenticate. The latter means that responses must be generated within a certain timeframe, and the PA did not respond quickly enough. They may attempt to authenticate again, restarting the authentication process by calling **authenticate()**.

Upon successful authentication of the user by the SPF, the SPF returns a reference to its ***i_ProviderAccess*** interface when the **requestAccess()** method is invoked. This interface will be used in subsequent access session interactions with the SPF. Similarly, the PA also returns a reference to its ***i_Access*** interface.

10. Now that a secure context between user and provider has been established, the asUAP again requests that the access session is established. This operation returns the PA's ***i_Access*** Interface Reference to the asUAP.

11. The PA invokes **requestAccess()** on the provider specific SPF to establish an access session that allows the user access to the provider's services.

Since the user has already been authenticated, a reference to the SPF's ***i_ProviderAccess*** interface is returned, as well as an access session Id, *asId,* generated by the UA, which is used to identify the access session.

12. The SPF contacts the UA and requests the setup of an access session with the authenticated user. The SPF provides some user specific information (ie.*userId*) to the UA. The UA generates a unique access session identifier *asId* and returns it to the SPF.

13. The PA completes access session setup by invoking the **setUserCtxt()** operation on the ***i_ProviderAccess*** interface. This gives the user's UA some information about the user's domain, such as interface references, and device information.

The user's UA acknowledges the receipt of this user's domain information. Once the PA receives the UAs confirmation, the access session can be officially considered to be established.

The PA informs the asUAP of the successful establishment of the access session. The asUAP in turn will inform the user.

14. The user requests a list of available services from the service provider. He specifies a list of desired properties.

15. The asUAP forwards the request.

16. The PA requests a list of services from the UA on behalf of the user. A list of services matching the user's desired properties is returned.

(It is assumed that the UA has an up to date list of available services from the SUB.)

4.2.3 Logout from a Provider

Description:

A user logs out of the provider domain. All access and/or service sessions are terminated upon completion.

Pre-Condition:

The asUAP, PA, and UA are available in their respective domains.

The user has an access session setup between the PA and UA.

Post-Condition:

All access and service sessions are terminated. All access and service session identifiers are invalid.

4.2.3.1 Message Sequence Diagram

1. The user uses the User Interface to request a list of active service sessions. The User Interface requires the user's user name (*userId*) and password (*userProperties*).

2. The asUAP forwards the request by invoking the **listServiceSessions()** method on the PA.

3. The PA requests a list of service sessions from the service provider. The UA returns a list of service sessions, *ssIdList*.

Note: The user may terminate the service sessions at this point. An example of this can be seen in Chapter 6, Sections 6.1.6.2 and 6.2.4.3.

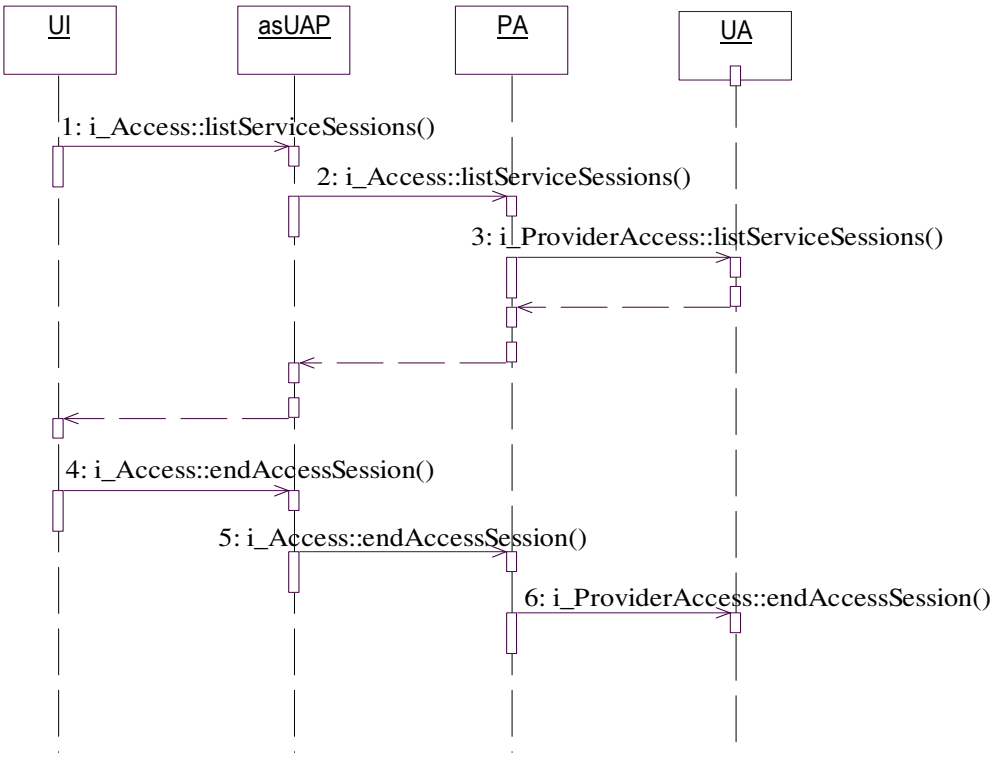

Figure 4.4: Logout from a Provider

4. The user requests to end the access session. The user inputs an access session Id, *asId*.

5. The asUAP forwards the request.

6. The PA forwards the end access session request to the UA. Any active service sessions are terminated forcibly at this point.

The UA considers the access session finished (deletes associated interfaces and user context) and returns an acknowledgment to the PA.
The PA deletes the interfaces associated to the access session and forwards the acknowledgment to the UAP, which in turn informs the user of the completion of the logout process.

4.3 Chapter Summary

The Access and Authentication management API were presented in this chapter with the assistance of interface descriptions, use case diagrams, and sequence diagrams. The functionality provided by the API was discussed with reference to the requirements specified in the previous chapter. Three TINA Ret RP interfaces were introduced, the ***i_Initial***, ***i_Authenticate***, and ***i_Access*** interfaces. Each of these interfaces played a role

in supplying the Access and Authentication functionality required by Parlay's end user interface. The ***i_Initial*** interface provided methods that allowed for the transparent location of a service provider and their offered services. The PA implements the method which allows a specified service provider to be located using a naming service offered by the DPE. The ***i_Authenticate*** interface supports the establishment of a secure service usage context between the consumer and service provider. It provides generic methods which can be used for authentication. Controlled and customized access was provided using the ***i_Access*** interface. ***i_Access*** offers a method which allows a user to offer consumer domain information such as current user applications, operating systems used. This information is used by the UA to setup a usage context for the specific user. Service delivery to a specific user is then based on the usage context. ***i_Access*** also allows a user to discover services. The provision of ubiquitous access to the range of Parlay services from any terminal is insured by the presence of a PA on the specified terminal.

5 DESIGN OF THE SUBSCRIPTION AND PROFILE MANAGEMENT API

This chapter details the design of a subscription and profile management API for Parlay's end user interface using TINA concepts, information structures and interfaces. The subscription and profile management APIs provide the Parlay enterprise operator with the capability to manage the set of consumers subscribed to their services. Using the subscription API users can subscribe, modify and release services offered by providers. Using the profile management API users can manage user, device, and service information. The profile management feature supports context aware service delivery by allowing users to customize their services. This chapter presents a detailed view of the implementation of the subscription and profile management APIs used in this research. The following section presents an information model which can be used to model user, subscriber, and subscription related entities. Thereafter, detailed descriptions of the subscription and profile management interfaces are presented. A use case view follows and the chapter concludes with a summary.

5.1 Subscription and Profile Management Information

The subscription information model is used by service providers to support the handling of end users, subscribers and subscriptions in a service provider domain. The model is used to represent the entities and relationships required in subscription management. The model includes the following main entities [10]:

- *Subscriber*:

  It represents an entity (a person or organization) that signs a contract with the retailer for the provision of a (set of) service(s). This class contains all the information related to the subscriber that is independent from the services it is subscribed to. It includes subscriber identification, address, contact points, etc.

- *End user/Entity*:

  It represents a user of a set of service(s). An end user is assigned privileges and access to service content by a subscriber. End users cannot subscribe to new

services. Only an end user can participate in a service. An end user is usually a person, however, it can also be a terminal or a network access point (NAP). The terms end user and entity are used interchangeably to describe a user, terminal, or NAP.

- *Anonymous user*:

  It represents a user who is unknown to the service provider and is not yet subscribed to any services. An anonymous user may become a subscriber and/or end user.

- *Subscription Assignment Group (SAG)*:

  It represents a group of users, terminals or NAPs (Network Access Points) associated to, and defined by, a subscriber who share a common service profile (the SAG service profile).

- *Subscription Contract*:

  It represents the agreement for the provision of a service to a subscriber. It describes the terms of the agreement. This class does not describe service specific information but the conditions of the contract for the provision of the given service. This can include payment mode, bank account information, etc.

- *Service Description*:

  It represents the service specification produced by a standardization body, industrial forum or group of companies. It described a particular service type.

- *Service Template*:

  It describes the generic information and behavioral characteristics of a service instance (of a specific service type) as offered by a service provider. The template contains a service description from which a service provider may set configurable and non configurable service information.

- *Service Profile*:

  It represents the tailoring of a Service Template to the specific requirements and needs of a subscriber. The service profile contains a modified service template containing user configured fields. This represents the user's preference information.

- *SAG Service Profile*:

  It is a customization of a service profile for a SAG.

- *Service Contract*:

  It describes the terms of the provision of a specific service to a subscriber. It collects the service, and SAG service profiles, defining the agreed service characteristics, the generic ones for the provider and the specific ones for each SAG respectively.

- *User Profile*

  The user profile is a dynamic object which contains all information that is used directly within the access session for authorization decisions, constraints and customization of the access sessions (within access session bindings) and service sessions.

Using the above descriptions, the following can be said about subscription management:

A *subscriber* subscribes to one or more services by signing a *subscription contract*. Subscription contracts may be signed with more than one service provider. Each contract represents an agreement between a subscriber and a single service provider for the provision of one or more *services*. It constitutes the service independent part of the deal. This is where specifics such as payment mode, subscription period, and consumer bank account information are conveyed.

For each *service* in the subscription contract the subscriber signs a *service contract*. This constitutes the service specific part of the subscription, and allows the subscriber to set conditions on service delivery. The service contract details the terms of the provision of a specific service to a particular subscriber.

Each *service* has a *service description* which is used for the description of service types, and instances. Service types are described using parameters in the service description common to all services of that type , and service instances are describe by parameters specific to each instance. For example, a video calling service type could have a common service parameter, *'devices allowed to access service'*, which would restrict access devices to only DPE enabled devices and restrict access to devices such as gsm cellphones. This would be a non-configurable parameter applying to all users of the service from that provider. A service instance could have a parameter, *'max. no.of parties in session'*, which could be set by a subscriber to any preferred value within some provider imposed restrictions. A service description is contained in a *service template*. This would represent a configurable parameter and thus allow users to specify their preference.

The subscriber uses the service template to examine and set service specific information and behavioural characteristics before signing a service contract. Using the service template, the subscriber may look up and configure the service description, as well as other generic information. A service description is contained within a service template, and a service template is contained within a *service profile*.

Once a subscriber has customized a service profile by configuring the service description in a service template, they may assign the service profile to a set of *entities*, which include *users, terminals, or network access points (NAPs)* to create a *subscription service*

profile. An *entity* is the same as an *end user*. If a subscriber does not wish to grant the whole set of *entities* the same service characteristics or privileges, they may create groups for some entities called *subscription assignment groups* (SAG) and assign to each SAG a customized service template to create a *SAG service profile*. By default, the *subscription service profile* is assigned to a SAG including the whole set of entities. The SAG service profile is therefore just a customization of a service profile for a specific SAG.

A subscriber completes the subscription process by defining a *service contract*. A service contract aggregates the subscription, SAG and *user service profiles* to define a complete set of service characteristics and privileges for all entities.

An end user's privileges and capabilities during an access and service session are restricted by a *user profile*. The user profile aggregates three other objects that describe information specific to an *end user* [4]:

- **Usage Context**:

  Specifies a configuration of consumer domain interfaces, network access and terminal equipment information such as capabilities which defines the user's domain. A usage context details the configuration of the user domain in order to gain access and/or use services according to a service contract. For each access session there will be a specific usage context for the user (this may change depending on the user's current terminal, or NAP as well as a host of other consumer premises information such as current applications), and it will constrain the invocation of service sessions within that access session.
  (The usage context is set at the completion of access session setup using the **setUserCtxt()** method on the ***i_ProviderAccess*** interface.)

- **Service Profile**:

  Specifies whether or not an end user can invoke a service and the characteristics of the service when invoked. It contains information that specifies retailer imposed constraints and subscriber imposed constraints. If an end user has been given the right to customize their services then the end user's preferences are also included in the service profile (in the form of the user service profile). This information may be altered by the subscriber. The service profile may be constrained by a usage context since subscriber and/or provider constraints may include device information (ie. type, installed applications, etc), network access points, and/or consumer domain interfaces.

- **Session Description**:

  Specifies an existing service session which the user is either active in, or entitled to join or schedule. A session description is created when a user creates a service session (either joining an existing service session or after receiving an invitation). A session description may be constrained by the usage contexts and service profile. A user profile may have many session descriptions, one for each service session. In this research the session description is defined using TINA's

t_SessionInfo structure. This data structure is described in the TINA Ret Reference pt specification [23].

5.2 Subscription and Profile Management Interfaces

5.2.1 The i_Subscribe interface

The ***i_Subscribe*** interface allows the creation of a subscription contract for a subscriber. It defines operations for:

- discovering available services.
- subscribing and unsubscribing to services.
- initiating the agreement of a service contract.

In order to subscribe to services, a consumer must first discover the available service offerings from the provider. The **listServices()** method is used by a consumer to request all the services available from a provider. The consumer may represent an anonymous user wishing to become a subscriber, a subscriber wishing to subscribe to a new service, or an end user wishing to view available services. This method takes as input parameters *desiredProperties,* which is used to specify the type of services the user would like to view and *howMany*, which is used to limit the number of services returned by the provider. Only one output parameter, *serviceList*, is specified. This is used to return a list of retailer services matching the *desiredProperties* parameter.

To subscribe to new services, new user (anonymous user) and already existing users (subscriber) use the **subscribe()** method.

The **subscribe()** method creates a subscription for a new or existing customer. The anonymous user lists the services he/she wants to contract using *serviceList*, along with some initial *subscriberInfo* detailing his/her particulars (ie.name, address). An existing user specifies a *subscriberId* along with a *serviceList*. The method returns: a unique identifier for the subscriber, *subscriberId* (only in the case of an anonymous user) and interfaces for subscriber information management, ***i_SubscriberInfoMgmt***,. and service contract management, ***i_ServiceContractMgmt*** (in both cases). These two last items are included in the *interfaceList* structure. A customer may then use the reference to the subscriber information management interface to organize his/her entities and thereafter, the service contract management interface to initiate the agreement of a service contract with the provider. The subscription is only complete (ie. usage of a service may begin) once a service contract has been negotiated.

When a subscriber no longer wishes to use a service, he/she can use the **unsubscribe()** method. This procedure withdraws a subscription or a list of service contracts. The list of services the subscriber wants to unsubscribe, *serviceList*, and a *subscriberId* are input parameters. If this list is empty, that means the withdrawal of all the services, and thus the subscription. A list of services, *unsubscribedServices*, listing the services that have been unsubscribed is returned.

5.2.2 The i_SubscriberInfoMgmt Interface

This interface allows the management of the information related to a particular subscriber. This interface can be used either by the subscriber or by the enterprise operator on behalf of the subscriber. Using the UA, the service provider may for example retrieve user information in order to set service delivery context or to authorize a user request. It provides operations for:

- creating and deleting entities (end users, terminals, and NAPs).
- creating and deleting SAGs.
- assigning/removing entities to/from a SAG.
- listing entities and SAGs corresponding to that subscriber.
- querying and modifying the subscriber information.

5.2.2.1 Creating and deleting entities (end users, terminals, and NAPs).

A subscriber cannot use services, to use services a subscriber must assign usage rights to an end user. An end user is usually a person, however, it can also be a terminal or a network access point (NAP). The subscriber and user entity may or may not be the same person, however, a subscriber may assign usage rights to more than one entity.

The **createSAEs()** method is used to create a set of entities, where each entity is characterized by its identifier (*entityId*), *entityName*, and a set of *properties*. (The *entityId* is a union type which can be switched between *userId*, *terminalId*, and *napId*.) The subscriber specifies an *entityList*, and a *subscriberId*. Each entity in the *entityList* is mapped to the *subscriberId* and an *entityIdList*, containing the *entityId*‘s of all the created entities is returned.

The **deleteSAEs()** method deletes a set of entities. A *subscriberId* and an *entityIdList* are specified, and each entity is removed from all the SAGs it could be assigned to and then deleted.

5.2.2.2 Creating and deleting SAGs.

If a subscriber does not wish to grant the whole set of entities the same service characteristics or privileges, they may create groups for some entities called subscription assignment groups (SAG) and assign to each SAG a customized service template to create a SAG service profile. By default, the subscription service profile is assigned to a SAG including the whole set of entities. The SAG service profile is therefore just a customization of a subscription service profile for a specific SAG.

SAGs are used by subscribers to differentiate preferred service characteristics and privileges for entities. A subscriber creates a SAG and assigns a set of entities to it. When a SAG is no longer useful, the subscriber may delete it, and perhaps create a new, more functional one.

The **createSAGs()** method creates a set of SAGs, where each SAG is characterized by its identifier (*sagId*), a *sagDescription* (a textual description of the characteristics of the group), and an *entityIdList* (containing the list of entities composing it).

The subscriber inputs a *sagList*, and a *subscriberId*. The *sagList* must contain a non-empty *entityIdList* and *sagDescription* for each SAG. The *entityIdList* specifies the entities he/she wishes to include in a particular SAG. The *sagDescription* is used to remind the subscriber of the purpose of the SAG when assigning a service profile.

The consumer interface then maps each SAG in the *sagList* to the *subscriberId* to create *sagId*s for each SAG. A *sagIdList*, containing the *sagId*'s of all the created SAGs is returned to the subscriber.

The **deleteSAGs()** method deletes a set of SAGs. The entities belonging to that SAG are not deleted.

5.2.2.3 Assigning/removing entities to/from a SAG.

Once a subscriber has created a set of SAEs and included them in a SAG(s), he/she may wish to add or remove them. The **assignSAEs()** method assigns a list of entities to a SAG, and the **removeSAEs()** method removes an entity list from a SAG. To add a SAEs to a SAG, the subscriber provides a *subscriberId*, an *entityIdList*, and a *sagId* in **assignSAEs()**. The same parameters are specified when removing a SAEs from a SAG.

5.2.2.4 Listing entities and SAGs corresponding to that subscriber.

Once a subscriber has created a set of SAEs and SAGs, he/she may wish to view them. The **listSAEs()** method lists a set of entities assigned to a SAG, and the **listSAGs()** method returns a set of SAGs for that subscriber. To list SAEs, the subscriber provides a *subscriberId*, an *entityIdList*, and a *sagId* in **listSAEs()**. The same parameters except the entityIdList are specified when listing SAGs.

5.2.2.5 Querying and modifying the subscriber information.

The **getSubscriberInfo()** method can be used to return information about a specific subscriber. Subscriber information such as account number, name, address, monthly charge, payment record, credit information, date which its subscription expires on, the list of subscribed services and the list of defined SAGs can be retrieved. A *subscriberId* is required as input.

The **setSubscriberInfo()** operation modifies the information about a specific subscriber. Only name and address fields are modifiable. The rest are updated only by SUB as a result of other operations such as: createSAGs, subscribe, etc

A subscriber may wish to only view his/her list of subscribed services. This functionality is provided by the **listSubscribedServices()** method which requires a *subscriberId* to return the corresponding subscriber's list of services (*serviceList*).

5.2.3 The i_ServiceContractInfoMgmt interface

This interface is used to manage the information related to a service contract. This includes the subscription service profile, the SAG service profiles (service profiles for users belonging to a specific SAG), and the user service profiles. From the service provider's perspective, this interface provides access to the service profile which as mentioned earlier, along with the usage context and session description determines a user's privileges and capabilities during an access and service session. The negotiation of a service contract is also facilitated by this interface. It provides operations for:

- retrieving and setting the *service template*.
- creating, modifying, and deleting *service profiles.*

- retrieving *service profile* and *service contract* information. This information includes the associated service profiles.

5.2.3.1 Retrieving and setting the service template.

A subscriber uses the **getServiceTemplate()** method to return a *serviceTemplate*. A *serviceId* corresponding to the requested service template is specified as input. Using a *serviceTemplate*, the subscriber may look up and configure a *serviceDescription*, as well as other generic information.

setServiceTemplate() is used to set user configurable parameters in the service description part of the service template. The method takes as input a *serviceId,* and a user configured *serviceTemplate* and returns a *serviceProfileId*. The *serviceProfileId* corresponds to the service profile containing the modified service template. It can be used when assigning a service profile to a SAG. Each invocation of the **setServiceTemplate()** method results in the creation of a new service profile. If the *serviceId* given already has a service profile assigned, then a new service profile must be created. A service may be identified with more than one service profile.

5.2.3.2 Creating, activating, and deleting service profiles

Once a subscribe*r* has customized the service description in a service template, they may assign the template to a set of entities, which include users, terminals, or network access points (NAPs) to create a service profile. If a subscriber does not wish to grant the whole set of entities the same service characteristics or privileges, they may create subscription assignment groups for some entities and assign to each SAG a customized service template to create a SAG service profile.

Before assigning a set of service profile to a set of SAGs, a subscriber may wish to see the available list of service profiles. The **getServiceProfiles()** method takes as input a *subscriberId*, *serviceId*, and returns a *serviceProfileList*.

A subscriber may assign a service profile to a list of SAGs and SAEs by invoking the **assignServiceProfile()** method. The subscriber specifies a *serviceProfileId*, a *sagIdList*.. The service profile to be assigned is identified by a *serviceProfileId*, and the SAGs are identified by the *sagIdList*. All SAGs and SAEs assigned to this service profile will have the same set of characteristics and privileges as defined in the service template.

If a subscriber wishes to remove a service profile assignment to a list of SAGs, they must invoke the **removeServiceProfile()** method. It takes as input a *serviceProfileId*, and a *sagIdList*. The end user interface compares the *sagId*s against those registered for that

serviceProfileId and removes any common SAGs. A *sagIdList* of all the SAGs that could not be removed is returned.

Once a service profile is assigned to a list of SAGs, a subscriber may activate it. The **activateServiceProfiles()** method activates a set of service profiles specified by the subscriber in a *serviceProfileIdList*. The **deactivateServiceProfile()** method can be used to reverse the operation. It takes a *serviceProfileIdList* as input.

At some point the subscriber may wish to delete a set of service profiles. The **deleteServiceProfiles()** method deletes a set of service profiles and their associated SAGs. It takes as input a *serviceProfileIdList*.

5.2.3.3 Creating and retrieving service contract information.

A subscriber completes the subscription process by defining a service contract. A service contrac*t* aggregates the service profile, SAG service profile and user service profiles to define a complete set of service characteristics and privileges for all entities.

The subscriber may view service contract information before or after signing a service contract using the **getServiceContractInfo()** method. The subscriber inputs a *subscriberId*, and a *serviceContract* is returned. The returned *serviceContract* contains a *serviceId*, *accountNumber*, a *serviceProfileList*, a *sagServiceProfileList*, and other particulars (e.g. start time, authority limit, etc).

When a subscriber wishes to create a service contract, she may invoke the **defineServiceContract()** method. The subscriber specifies a *serviceContract*, and a *serviceProfileIdList* is returned. A successful invocation of this method indicates the completion of the subscription process.

Appendix A.2 provides a summary of all the interfaces, operations, and attributes presented in this chapter.

5.3 Proposed Implementation Scenarios

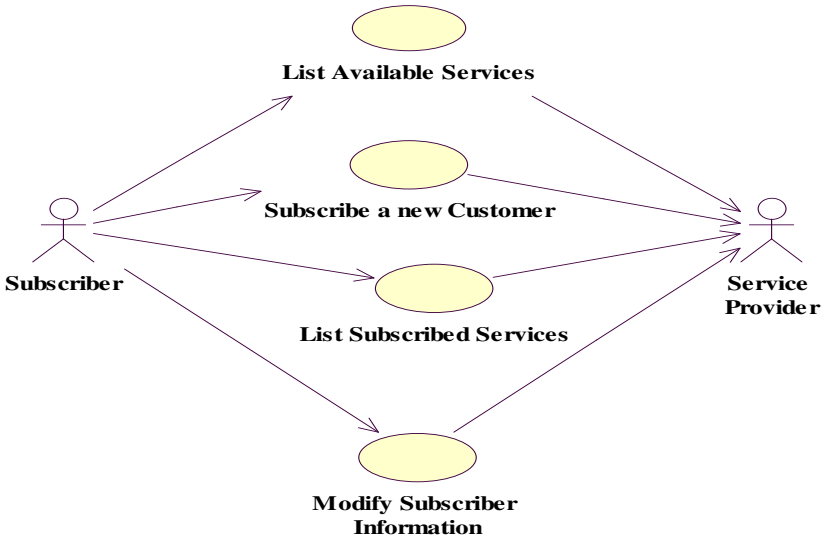

Figure 5.1: Subscription and Profile Management Use Case Scenarios

Figure 5.1 presents the use cases implemented in the subscription and profile management scenarios. The following sections describe these use cases, and use sequence diagrams to illustrate how they function.

5.3.1 Subscribe a New Customer (Becoming a Subscriber)

Description:

A new customer is subscribed to the provider. He/she contracts a number of services, and defines contract and subscriber information. Upon completion, the subscriber's service profile is activated.

Pre-Condition:

The customer is not previously subscribed to the provider.

Post-Condition:

The subscriber has an active service profile.

5.3.1.1 Message Sequence Diagram

1. The anonymous user requests to view the list of services the provider is offering.

2. The ssUAP passes the request on to the UA.

3. The UA requests the list of provider services on offer. At the end of this invocation the SUB returns the list, and thereafter the UA returns the list to the ssUAP, which returns the list to the UI therefore satisfying the user request.

4. The anonymous user requests that a new subscriber be created. The user request includes a list of the services the new customer would like to subscribe to along with subscriber information required by the provider in order to create the subscriber account (for instance, user identification and billing information).

The subscriber id is returned as a result as well as a list of service templates describing the properties of each offered service.

5. The ssUAP passes the request to the UA.

6. The UA requests the SUB to create a new subscriber with the given subscriber information and list of services. Upon completion, a new subscriber identifier is created.

7. The user requests a service template. He/she inputs a service identifier.

8. The ssUAP forwards the request to the UA.

9. The UA requests a service template from the SUB.

10. The user requests configurable service parameters in the service template to be set up.

11. The ssUAP forwards the request to the UA.

12. The UA requests the SUB to configure the service template. Upon completion, the SUB creates a service profile and returns the service profile identifier to the user.

The subscriber may repeat steps 7 – 9, to create new service profiles for the same service or configure service templates to create new service profiles for other services.

13. The user requests the creation of new subscription assignments for each of the entities (ie. Users or terminals) that could make use of the contracted service.

14. The ssUAP passes the user request onto the UA.

15. The user's request for new subscription assignments is passed on to the SUB. Upon completion, the list of assigned entity identifiers is returned. These identifiers are unique in the provider domain. The SAE list is stored in a database and then returned to the user.

16. The user requests the creation of new SAGs. The user specifies a SAG list where each SAG is characterized by its identifier, a textual description of the characteristics of the group, and the list of entities assigned to it.

17. The ssUAP passes the user request onto the UA.

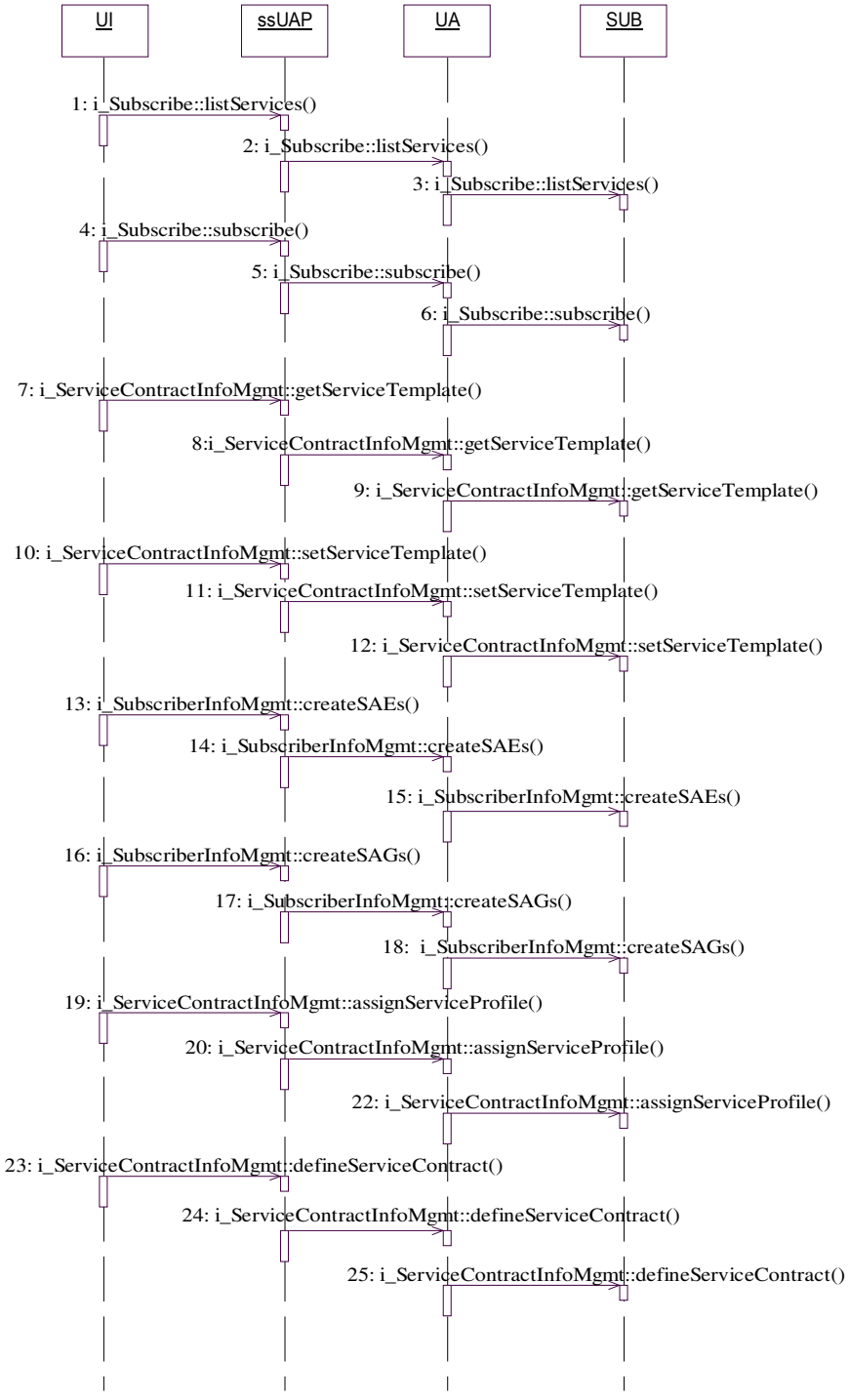

Figure 5.2: Subscribe a New Customer Sequence Diagram

18. The creation request is forwarded to the SUB. The SUB assigns the list of entities to the SAG and stores the assignment in a database. Upon completion, a list of SAG identifiers is returned to the subscriber. These identifiers are unique in the provider domain.

19. The subscriber requests the ssUAP to assign a service profile to a list of SAGs. The subscriber specifies a service profile identifier, and a list of SAG identifiers.

The subscriber may obtain service profiles by invoking the **listServiceProfiles()** method.

20. The ssUAP forwards the request to the UA.

21. The UA forwards the request to the SUB. The SUB assigns the SAGs, and contained SAEs to the service profile.

22. The User requests the ssUAP to define a new service contract. This contract includes contractual information, the set of service profiles for each requested service. The agreed Service Contract defines the conditions of the service provision for each of the service subscriptions.

23. The ssUAP passes the request to the UA. Upon completion, the returned service contract is saved by the UA in the subscriber's profile.

24. The UA passes the SUB the service contract information. A list of Service Profile identifiers for each subscribed service is returned and the subscriber is now registered to use services.

5.3.2 Modify Subscriber Information

Description:

This use case describes how the subscriber can modify their service profile. The subscriber modifies the service profile by adding and deleting information (SAEs and SAGs) from the service profile.

The main aspects that can be modified are SAGs, SAEs and assignment of SAEs to SAGs. This event trace shows first how new SAEs are defined and assigned to existing SAGs, then how existing SAEs are removed from a SAG and finally how some are deleted. The subscriber is already subscribed to the provider and has contracted some services. A number of SAGs and SAEs have been previously defined for the subscriber.

Pre-Condition:

The customer is already subscribed to the provider and has already contracted some services. A number of SAGs and SAEs have been previously defined for the subscriber.

Post-Condition.

None.

5.3.2.1 Message Sequence Diagram

1. The user requests the creation of new subscription assignments for each of the entities (ie. Users or terminals) that could make use of the current service by invoking the createSAEs() method on the ssUAP. Optionally, the user could include a service id to identify a chosen service when invoking the method while that service is not active.

2. The ssUAP passes the request to create new SAEs onto the UA. Upon completion, the UA updates the subscriber's profile by modifying the list of SAEs.

3. The user's request for new subscription assignments is passed on to the SUB. Upon completion, the list of assigned entity identifiers is the returned value. These identifiers are unique in the provider domain.

The list is passed back to the user interface.

Now the new SAEs are assigned to an existing SAG:

Note: SAEs can also be assigned to a SAG using the **createSAGs()** method.

4. The user asks for the list of SAGs defined for the subscriber.

5. The ssUAP passes the request onto the UA.

6. The UA requests a list of SAGs on behalf of the user. A subscriber id is passed to identify the subscriber.

A list of SAG identifiers is returned and passed back to the user interface.

7. The user requests that a list of SAEs be assigned to an existing SAG. An subscriber id, entity list, and a SAG id are input parameters for this method.

8. The ssUAP passes the request onto the UA. Upon successful completion the UA saves the new SAE/SAG mapping.

9. The user's request for the assignment of SAEs to a new SAG is passed on to the SUB. Upon completion a new SAE/SAG assignment is logged, however, no objects are returned.

Removal of SAEs from a SAG:

10. The user requests a list of SAEs assigned to a specific SAG. The SAG identifier is one of the input parameters.

11. The ssUAP passes the request on to the UA.

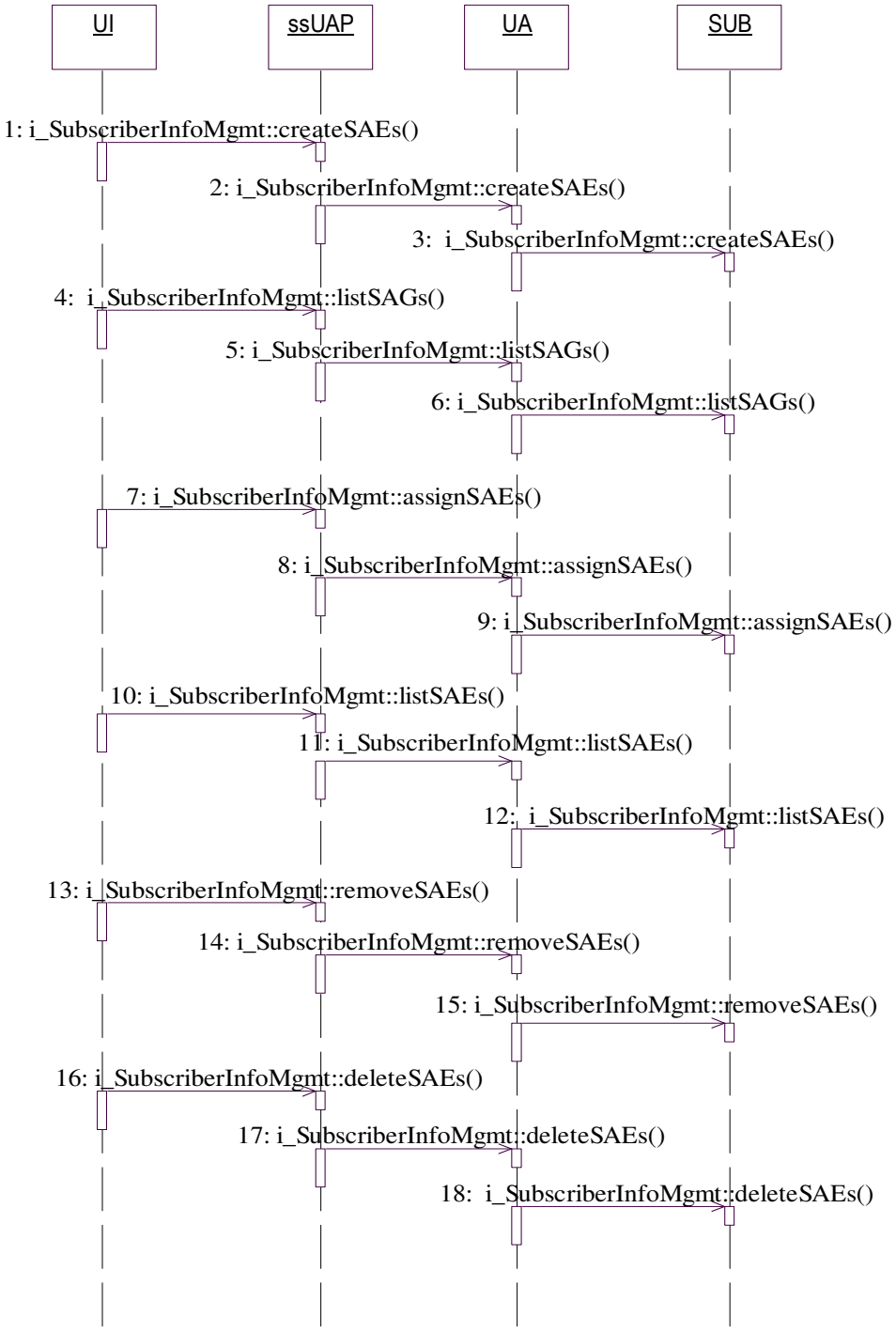

Figure 5.3: Modify Subscriber Information Sequence Diagram

12. The UA asks the Sub for a list of SAEs on behalf of the user. A list of entity identifiers is returned and passed back to the user interface.

13. The user requests the removal of a list of SAEs from a SAG. An entity list and a SAG id are input parameters to this method.

14. The ssUAP passes the request on to the UA. Upon completion the UA logs the removal of SAEs by updating the service profile.

15. The UA requests the removal of a list of SAEs from the SUB on behalf of the user. The Sub removes the entities from the SAG.

Deletion of SAEs:

16. The user requests the deletion of SAEs from any SAG they could be assigned to. No SAG identifier is required for this method.

17. The ssUAP forwards the request to the UA. Upon completion the UA logs the removal of SAEs in the subscriber's profile.

18. The UA forwards the request to the SUB. This causes the SAEs to be removed from any SAG they could be assigned to. The subscriber information is updated in the subscriber database.

5.4 Chapter Summary

The consumer interface's Subscription and Profile management APIs were presented in this chapter. First, an information model describing all the objects involved in the subscription lifecycle process was given. The interaction between consumers and subscription objects such as subscription assignment groups, service profiles, and/or service contracts, etc, was defined. Based on the defined interaction, the ***i_Subscribe***, ***i_SubscriberInfoMgmt***, ***i_ServiceContractInfoMgmt***, and ***i_SubscriberInfoQuery*** interfaces were defined. The ***i_Subscribe*** interface provided operations that supported the creation of subscription contracts for a subscriber. The ***i_SubscriberInfoMgmt*** interface was defined to support the management of information related to a particular subscriber. It provides operations to support the creation and deletion, assignment, removal, and listing of SAEs, and SAGs. The ***i_ServiceContractMgmt*** interface provided methods to support the management of information related to a subscriber's service contract with the service provider. The service contract includes the subscription service profile, SAG service profiles, and user service profiles. To retrieve subscription information for a particular end user, the i***_SubscriberInfoQuery*** interface is implemented. The chapter concluded with a description of the proposed implementation scenarios, where the interaction between end users and subscribers and the service provider through the consumer interface was illustrated through the "Subscribe a new customer", and "Modify subscriber information" scenarios. These scenarios were modelled using use cases and

the behaviour of service components implementing the interfaces was illustrated through sequence diagrams.

6 DESIGN OF THE SERVICE USAGE MANAGEMENT API

In this project report the service usage management API merges the TINA and Parlay service architectures into a hybrid Parlay/TINA application environment. In order to access network connectivity, applications must interact with the Parlay gateway, and to access the consumer domain, a TINA interfaces must be used. An interworking of the two service architectures is required and this chapter describes how this is achieved.

With service usage in Parlay's consumer interface there are two main scenarios. One is single party service usage which implements the TINA's BasicFS, and the other is multiparty service usage which implements TINA's MultipartyIndFS. The service usage management APIs are required to provide functionality which enables end users to:

- Initiate and terminate single party service sessions.
- Initiate and terminate multiparty service sessions.

6.1 Initiating and Terminating a Single Party Service Session

When a service session is started, it retrieves user profile information acquired during access session setup and subscription. The information is used to provide context awareness during service delivery as well as to customize the service for a user. A service session can only be instantiated via an access session. The service session then remains active while that access session is active, and when the access session is ended, active service sessions must be ended. Both multiparty and single party service sessions are initiated in the same way. The main difference is in the way they are terminated and the number of connections each is allowed to have. The following sections describe the interfaces and methods used in initiating a service session. An example of a generic messaging service using the Generic Messaging SCF is given to illustrate the initiation and termination of a single party service session.

6.1.1 The i_Access Interface

A user may only start a service session from within an access session. The user should already be authenticated, and therefore have a reference to the ***i_ProviderAccess*** interface. This interface provides consumers with secure and controlled access to OSA/Parlay interfaces and was introduced in Chapter 4.1.3. It provides a collection of methods through which a consumer may retrieve service, user and access session information. The user may also start and terminate a service using this interface.

To start a service session the user invokes the **startService()** method. The method takes as input parameters a *serviceId*, which is used to identify the service the consumer wishes to start, an *app* structure containing information on the application (ie. Application name, version, serial no., etc), and *uaProperties* which is used to pass user preference information before the service is started. The only output parameter returned is the *sessionInfo* structure which contains the service session id, *ssId*, generated by the UA, *purpose*, a string describing the purpose of the session, a *partyId*, which is used for multiparty services, *state* which describes the current state of the session, and *properties* which gives some properties of the current session. Along with the service profile, and the usage context, the *sessionInfo* data structure is also used to make authorization decisions for the user during the service session.

6.1.2 The i_SSManage Interface

The ***i_SSManage*** interface is an interface designed by the author to manage the lifecycle of service specific components. It is a combination specific methods from TINA's ***i_SSCreate*** and ***i_SSManage*** interfaces. It is only implemented by the SF. Generally, the UA uses this interface to request the creation or termination of a service and the App uses it to request the creation of service specific OSA/Parlay interfaces.

The **createSSession()** method allows the UA to request the creation of a new service session using the SF. To create a service the service factory requires that the UA provide a *serviceId*, *userId*, some information about the app (*app*), user preference information (*uaProperties*), and after the service is initiated the method returns a *sessionInfo* structure (which is described in Section 6.1.1). The UA generates a sessionId, *ssId* and passes it to the consumer domain through *sessionInfo*. The rest of the information in *sessionInfo* may be generated by the App and SF, however this is left to the service provider to decide.

The App uses the **createIpAppInterface()** is a method designed by the author to request the creation of service specific OSA/Parlay interfaces. The method takes a list of interfaces, *interfaces,* and a *serviceId*, as inputs and returns references to the created interfaces as outputs. The SF uses the *serviceId* to determine whether the interface creation requests by the App are valid (we assume that the service provider maintains a list of service Ids and allowed Parlay interfaces). Upon completion the Application may perform operations on the requested interfaces.

The **endSSession()** method allows the UA to request the deletion of a service session. The SF deletes references to any OSA/Parlay interfaces that were in use during the service session (the App is informed that the references are no longer valid using the **releaseIpAppResources()** method in Section 6.1.5).

6.1.3 The Basic Feature Set

In Chapter 3.2.1, we discussed Feature Sets. This section elaborates on the implementation of the BasicFS to support a single party service session. The BasicFS implements the ***i_BasicReq*** interface as a means to support user requests to terminate a single party service session. Using the **endSessionReq()** method an end user may request the termination of a service session. The *sessionId*, and *userId* are taken as inputs and no output parameters are returned. The *sessionId* identifies the session to be ended. The *userId* identifies the user requesting to end the session and is used to check the validity of an **endSessionReq()**. An **endSessionReq()** is used for both single and multiparty service sessions. In a multiparty service session, the **endSessionReq()** is followed by an **endSessionInd()** and **endSessionReq()** (see Section 6.2.1). After this request completes successfully, the session will end. (In a multiparty service session, the session may not end until other users are notified.)

6.1.4 OSA/Parlay Interfaces

When initiating and terminating single party service sessions, call control functionality is provided by the OSA/Parlay call control SCFs. Depending on the type of service in use, one of three types of call control mechanisms can be launched for a single party session.

6.1.5 The IpAppLogic interface

The ***IpAppLogic*** interface represents the OSA/Parlay application logic. In this project, the ***IpAppLogic*** interface provides an interworking between the TINA and OSA/Parlay specific logic. The Parlay specific logic can generally be associated with a pattern of three objects: the IpAppCM, IpAppCO, and IpAppCL. These are callback interfaces which correspond to the Call Manager(CM), Call Object (CO), and Call Leg (CL) connectivity provider interfaces respectively in the OSA/Parlay gateway. The callback interfaces provide a mechanism by which the gateway interfaces can return results and notifications to the Application (***IpAppLogic***). Application interfaces are instantiated by

the SF, which must co-ordinate their creation and deletion. App invokes operations on Gateway objects using the IpCM, IpCO and IpCL. The CM, Call, or Call Leg objects may return results and notifications to the Application via the respective callback interface. For example IpCO returns notifications via IpAppCO which forwards the notification to the Application. In this research the App component is a generic representation of all OSA/Parlay applications that may use the consumer interface. Therefore, no service specific logic is implemented however the examples that are given later show how service specific logic could be integrated.

The IpAppLogic interface utilizes both TINA and OSA/Parlay interfaces. The TINA interfaces provide support for interactions with the consumer interface while the OSA/Parlay interfaces provide support for interactions with the Parlay gateway. The interfaces required to interact with the Parlay gateway are specified in the Parlay standards, and thus are not described here. The following TINA methods are implemented for the ***IpAppLogic*** interface.

- **joinSessionReq()**
- **inviteUserReq()**
- **joinSessionWithInvitation()**
- **joinSessionInd()**
- **endSessionReq()**
- **endSessionExe()**

These methods are described in the coming sections. Two user defined methods are also implemented within the ***IpAppLogic*** interface. The **initiateApp()** method instructs the App to start a service session by initiating the creation of service specific interfaces. Once invoked the method takes as input the *sessionInfo* structure (see.Section 6.1.1), and returns a list of interfaces, *interfaceList* which contains the callback interfaces required to interact with the gateway. The list is dependent on the type of services offered (ie. context awareness, generic or multimedia services, etc) and the SCFs subscribed to by the Application.
Once an application is involved in a service session it will require the creation of callback interfaces to interact with the OSA/Parlay gateway. The App requests the creation of callback interfaces from the SF and references to created interfaces are returned using the **passReferences()** method.
Upon termination of a service session the SF may request the application to release all references to interfaces that were in use during that session using the **releaseIpAppResources()** method. This method takes a *sessionId* as input.

6.1.5.1 The Generic Messaging SCF

The Generic Messaging SCF (GMS) can be used by OSA/Parlay applications to provide voice mail and electronic mail messaging. The GMS is represented by the IpMessagingManager, IpMailbox, IpMailboxFolder and IpMessage interfaces. Applications that use the GMS must implement the IpAppMessagingManager interface to provide a callback mechanism. The GMS will be used to illustrate the initiation of a simple messaging service.

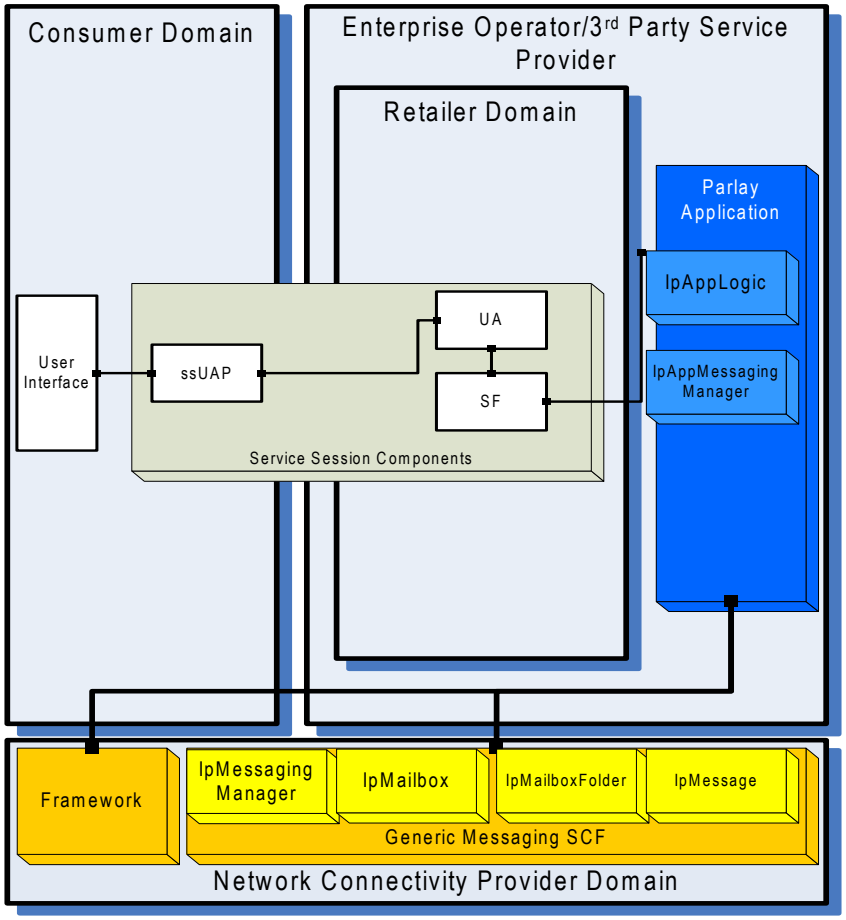

Figure 6.1: TINA's Service Session Components and Parlay's Generic Messaging SCF

Appendix A.3.1 provides a summary of all the interfaces discussed in Section 6.1.

Figure 6.1 illustrates the relationship between TINA's service session service components and the Parlay Generic Messaging SCF's IpCCM, IpCall, and their corresponding call back interfaces.

6.1.6 Proposed Implementation Scenarios

Initiating a service session using the consumer interface is the same for both single and multiparty users. Differences only arise during service usage when a user is to be invited to the service session. Figure 6.2 presents the use cases for the single party service usage management scenarios. These are described using sequence diagrams in the following sections.

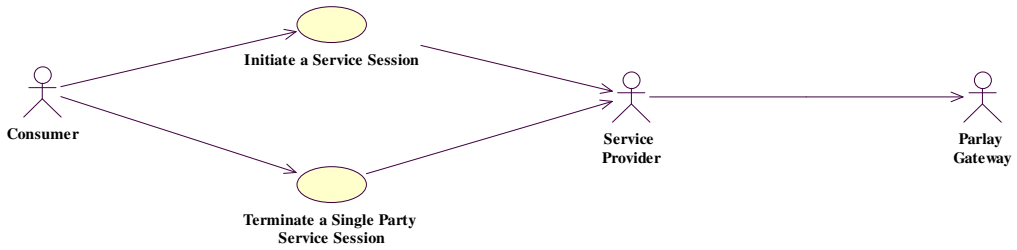

Figure 6.2: Single Party Service Usage Management Use Case Scenarios

6.1.6.1 Initiate a Service Session

Description:

A user starts a new service session. On completion, a service session is started and the user is ready to invite other users to join him/her in the session.

Pre-Condition:

The ssUAP, UA, SF, and App are present in their respective domains.

A list of available services (along with serviceIds) is accessible to the user. An access session exists between the PA and UA in the provider domain.

The App has access to the necessary OSA/Parlay gateway SCFs.

Post-Condition:

A service session exists between the ssUAP and UA. The SF has generated the necessary service session identifiers. The ssUAP has access to the UA's service session interface references. Only a single user is involved in the session.

Message Sequence Diagram

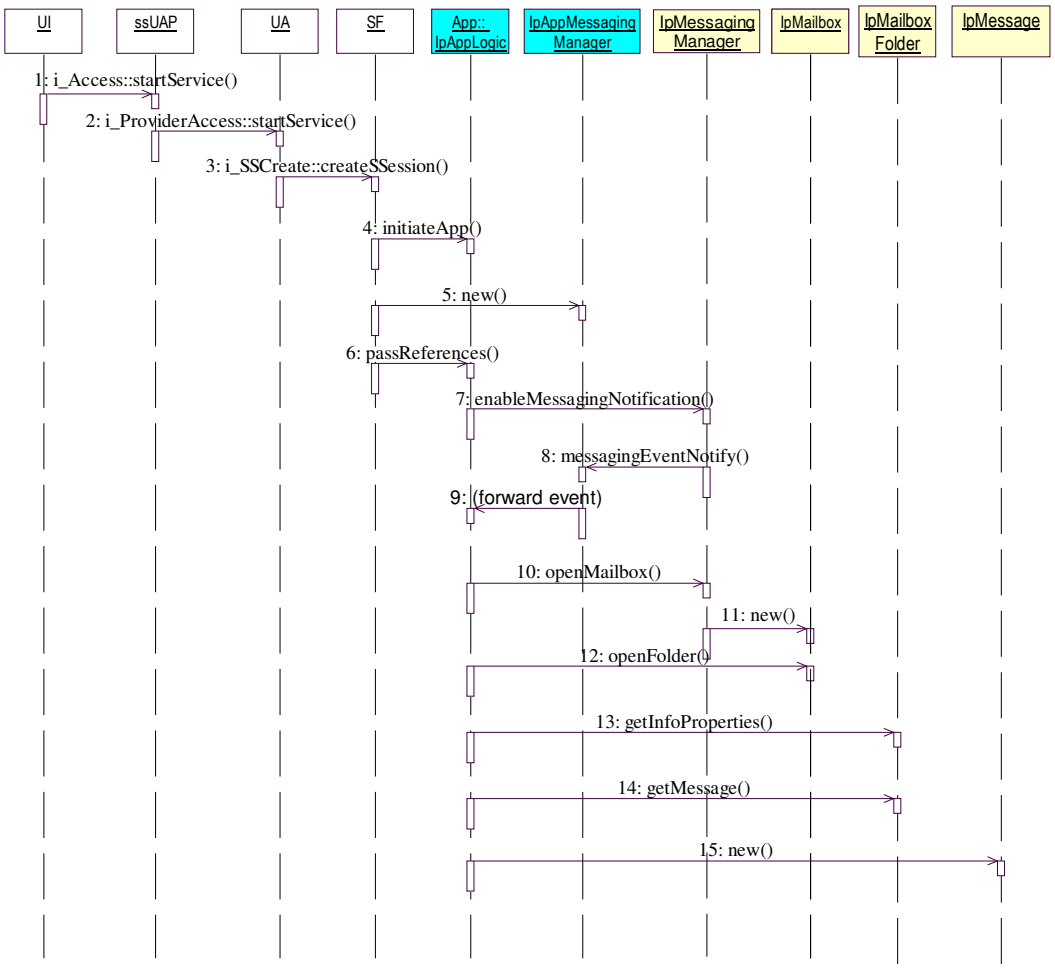

Figure 6.3: Initiate a Service Session Sequence Diagram

1. The user uses the User Interface to request the ssUAP to initiate a Parlay application.

2. The ssUAP requests from the UA a new service session of a particular service type (i.e. Generic, Multimedia, etc). Other properties for the service can be specified. The ssUAP may also give the interface types and references it will support in the session. The UA may perform various personalization actions before continuing. The UA may return unsuccessful and raise an exception to the ssUAP if the service request is declined.

3. The UA requests the SF to create a new service session for a particular service type requested by the user.

4. The SF initiates the 3rd party application by invoking the method initiateApp. The App returns a list of interfaces corresponding to callback interfaces that it requires to interact with the gateway.. (The SF is responsible for the management, creation and deletion of

all objects required to run the service. For example, each time the application logic requires the instantiation of an interface to interact with the Parlay gateway, the SF takes on the responsibility to create it and delete it when its function is complete. A complete definition of this interface is however outside the scope of this research.)

5. The SF uses the interfaceList to create an IpAppMessagingManager callback interface.

6. The SF passes a reference for the IpAppMessagingManager interface to the .App

7. The App requests the GMS to enable the notification mechanism so that events can be sent to the application.

8. The IpMessagingManager passes the new mail event to its callback interface on the application side, the IpAppMessagingManager.

9. The IpMessagingManager informs the Application (IpAppLogic) of the arrival of new mail.

10. The Applciation requests the IpMessagingManager to create a new mailbox (ie. IpMailbox interface)

11. Assuming that the criteria for creating a mailbox is met, the IpMessagingManager creates it.

12. The Application requests a folder to be opened and returns a reference to that folder.

13. The Application requests all of the folder information properties.

14. The Application requests a message from the opened mailbox folder.

15. Assuming that the criteria for creating an object implementing the IpMessage interface are met, the message is created.

6.1.6.2 Terminate a Single Party Service Session

Description:

A user ends the service session that he/she created. On completion all participants of the service session are removed and the user is ready to start a new service session.

Pre-Condition:

The components required to sustain a service session for all participants are present in their respective domains.

Post-Condition:

Accounting information is returned to all parties that were involved in the session.

Message Sequence Diagram

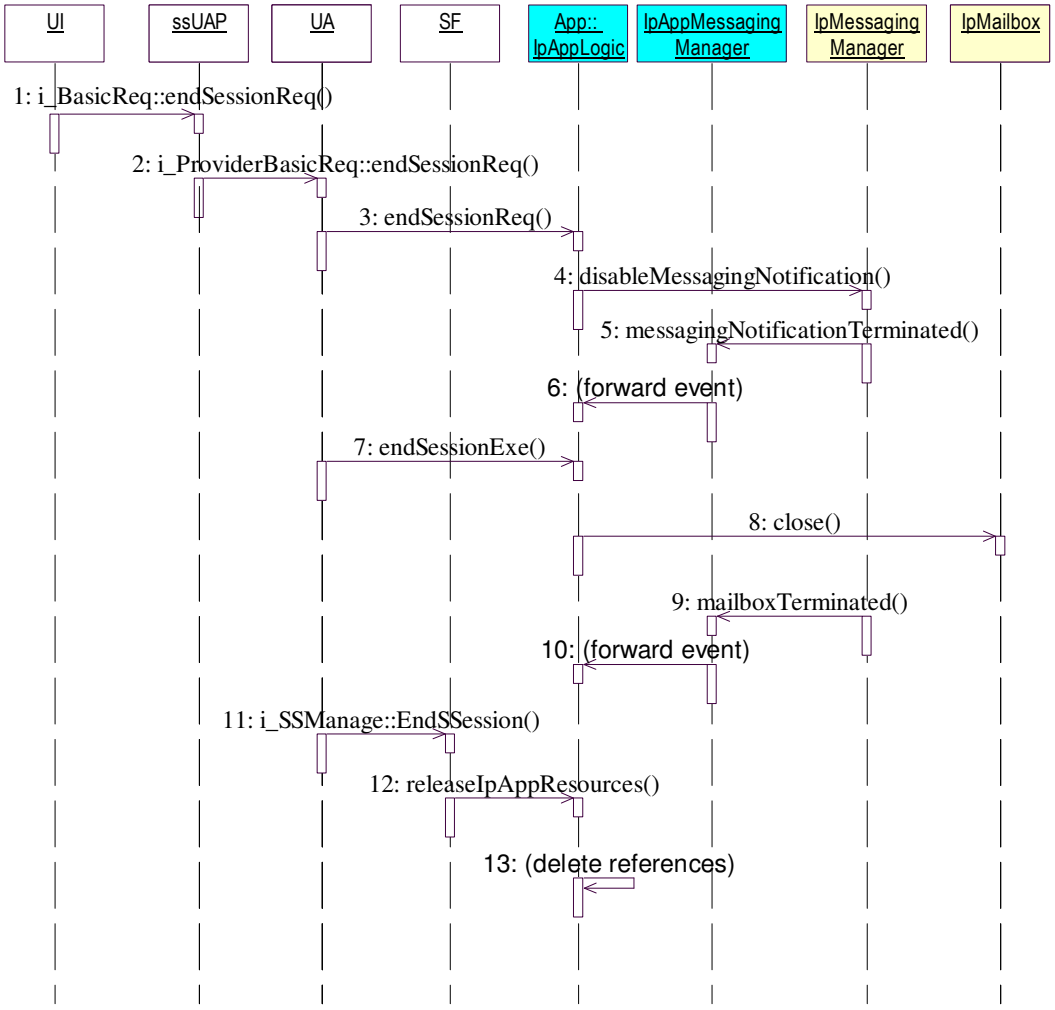

Figure 6.4: Terminate a Single Party Service Session Sequence Diagram

1. The User Interface requests the session to be ended by invoking the endSessionReq() on the ssUAP..

2. The request is passed on to the UA. The return value of this invocation is the accounting information for the requesting user.

3. The UA requests the Application to end the service session. The return value of this invocation is the accounting information for the requesting user. The UA stores this information in the user's profile.

4. The App requests the GMS to disable the notification mechanism

5. The IpMessagingManager notifies the IpAppMessagingManager.that messaging notification has been disabled.

6. The IpAppMessagingManager informs the Application (IpAppLogic) of the termination of messaging event notification.

7. The UA sends an Exe to the Application instructing it to end the session. The Application passes Accounting information back to the UA with this invocation, thereafter, the service session specific interfaces are made unavailable and all accounting is stopped.

8. The Application requests that the mailbox be closed.

9. The IpMessagingManager notifies the IpAppMessagingManager.that mailbox has been closed.

10. The IpAppMessagingManager informs the Application (IpAppLogic) of the termination of the mailbox.

11. The SF is informed that the session is about to end and eventually releases the Application specific resources.

12. The SF releases all application resources and instructs the Application that references are no longer valid. In this case the reference to the IpMessagingManager interface.

13. The App deletes all references that were in use during the session.

6.2 Initiating and Terminating Multi Party Service Sessions

Both a multiparty and a single party service session are initiated in the same way however, the number of connections in a multiparty service session is limited only by the connectivity operator's policy whereas a single party service is limited to one connection. This section describes the implementation of interfaces and methods to support multiparty connectivity in services. First, an implementation of the TINA multiparty feature set is discussed, followed by a look at OSA/Parlay's generic call control and multiparty call control SCF. The section concludes with examples of a service integrating the TINA multiparty feature set and OSA/Parlay's generic call control and multiparty call SCFs. Scenarios illustrating the initiation and termination of the integrated service are detailed.

6.2.1 The TINA Multiparty and MultipartyInd Feature Sets

In Section 3.3.1 we described the Multiparty and MultipartyInd feature sets. In this section we describe the interfaces used to implement the functionality provided by the feature sets.

6.2.1.1 The i_PartyMultipartyReq interface

The ***i_PartyMultipartyReq*** is a TINA interface which allows the MultipartyFS to support the execution of generic request operations such as:

- Request that a user is invited to join the session.

Using the **inviteUserReq()** an end user may request that another user is invited to join the session. It is used to invite a single specific user to join the session. The requesting user provides a resolvable name and address of the invitee in the *userDetails* structure and his own *userId*. *invitedUserDetails* contains the *userId* of the invited user, and a list of his *userProperty*'s. The *userPropertys* may allow the provider domain to locate the invited user, e.g. by including a reference to the service provider to contact for the user, (although the *userId* should be sufficient to locate the user). An *invitationId* parameter is used to identify the invitation request. The party may wish to cancel the request, and can use the *invitationId* to identify the invitation to be cancelled. The method returns an *invitationReply* which is the invited user's reply to the invitation.
The **joinSessionReq()** method allows an end user to request to join an existing service session. The method takes as input parameters a *userId*, to identify the user, *serviceId*, which is used to identify the service the consumer wishes to join, a *sessionId*, to identify the existing session, an *app* structure containing information on the application (ie. Application name, version, serial no., etc), and *uaProperties* which is used to pass some user preference information before the service is started.

6.2.1.2 The i_PartyMultipartyInd interface

The ***i_PartyMultiPartyInd*** interface allows TINA's MultipartyIndFS to support session indications that an action will be taken shortly. Supported actions are:

- a user is going to end the session.
- a user has been invited to join the session.
- a user is going to join the session.

The functionality provided by the complete interface is detailed in [23], however, only a subset of that functionality is implemented in this research.
The **endSessionInd()** method is used to indicate that a service session is about to end shortly, and corresponds to the **endSessionReq()** (see BasicFS in Section 6.1.3), and **endSessionExe()** methods. It takes as input a *sessionId*, and an *indId*. The *sessionId* is used to identify the session to be ended. The *indId* provides an indication id for the session. No *userId* is required since an **endSessionInd()** invocation means that an **endSessionReq()** method was invoked and a *userId* was found to be valid.
The **inviteUserInd()** method is used to indicate that a user has been invited to join a service session and corresponds to the **inviteUserReq()** invocation. It takes as input parameters a *sessionId*, an *indId*, and *userDetails*. The *sessionId* is used to identify which session the user has been invited to join. The *userDetails* parameter is used to provide other users with some details about the invited user.
The **joinSessionInd()** method is used to indicate that a user is about to join a service session shortly, and corresponds with a **joinSessionWithInvitation()** or **joinSessionReq()** invocation. It takes the same parameters as **inviteUserInd()**, however it also returns an *isValid* parameter of type Boolean which can be used to indicate whether an invitation is valid or not.

6.2.1.3 The i_PartyMultipartyExe interface

The ***i_PartyMultipartyExe*** interface allows the user to terminate a multiparty service session. The functionality provided by the complete interface is detailed in [23], however, only a subset of that functionality is implemented in this research. The ***i_PartyMultipartyExe*** interface implements the **endSessionExe()** method which is used to terminate a service session. The method corresponds to the **endSessionReq()** (see BasicFS in Section 6.1.3) and **endSessionInd()** methods, and is only invoked once the session has determined that an **endSessionReq()** is allowed. Once invoked the session must be ended. **endSessionExe()** takes a *sessionId,* and *userId* as an input parameter, and returns no output parameters.

6.2.1.4 The i_PartyMultipartyInfo interface

The ***i_PartyMultipartyInfo*** interface allows the session to inform the party domain of changes in the state of the session and its participants [23]. It allows the session to inform other users of actions such as:

- another party having joined the session.
- a user having been invited to join the session.
- a user having decided to end the session.

All the methods provided by the ***i_PartyMultipartyInfo*** interface correspond to methods in the ***i_PartyMultipartyInd*** interface.
inviteUserInfo() corresponds to the **inviteUserInd()** and **inviteUserReq()** methods. It is used to send the invited user's user information to all other participants of the current service session.

joinSessionInfo() corresponds to the **joinSessionInd()** and **joinSessionReq()** methods. It is used to send information about the user who has joined the service session.

endSessionInfo() corresponds to the **endSessionExe()**, **endSessionInd()**, and **endSessionReq()** methods. It is used to send information about an existing service session which has ended.

The **inviteReplyInfo()** is different to the other Info operations defined on this interface because it does not correspond to a Req operation. It is sent to all the parties in the session, when a reply to an invitation is received from an invited user.

6.2.2 The i_Invitation interface

This interface allows its clients to send invitations to the user's UA or cancel them, as well as view sent invitations or use them to join an existing service session.

The **invite**() method allows users to send an invitation to join a service session to be sent to a specified user. An identifier for the service the user is being invited to join, as well as the name of the inviting party, the purpose of the session, and the reason for the invitation, are included in an *invitation* data structure.

A user can view a list of sent invitations by invoking the **listSessionInvitations()** method. The method takes a *userId* as an input and returns an *invitationList* data structure.

To join a session with an invitation an end user must invoke the **joinSessionWithInvitation()** method. It takes as input an *invitationId* and *app* structure. The invitation is the same as that sent using the invite method, while the app is a structure containing information on the application. Both anonymous and known users may invoke this method. Both sets of users are the appropriate rights and privileges according to their user status. A *sessionInfo* structure is returned.

Invitations may be cancelled by invoking the **cancel()** method. It takes a *userId*, and *invitationId* as input and returns no output.

6.2.3 OSA/Parlay Interfaces

6.2.3.1 The Generic Call Control SCF

The Generic Call Control SCF is described in Section 2.2.2.1. For the purposes of this discussion, we describe in more detail the interfaces required for its proper function on both the service provider and connectivity provider sides. The GCCS is represented by the IpCallControlManager and IpCall interfaces that interface to functionality provided by the NSCF layer. Applications that use the GCCS to provide connectivity must implement the IpAppCallControlManager and IpAppCall interfaces to provide a callback mechanism which can be used to handle responses and reports. During the service session, the App service component requests access to the Application interfaces from the SF. The SF manages the lifecycle of both the IpAppCallControlManager and IpAppCall interfaces. The GCCS will be used to illustrate a basic two party voice call service. In the first scenario a user invites another user to join a service session. The second scenario illustrates the termination of a service session.

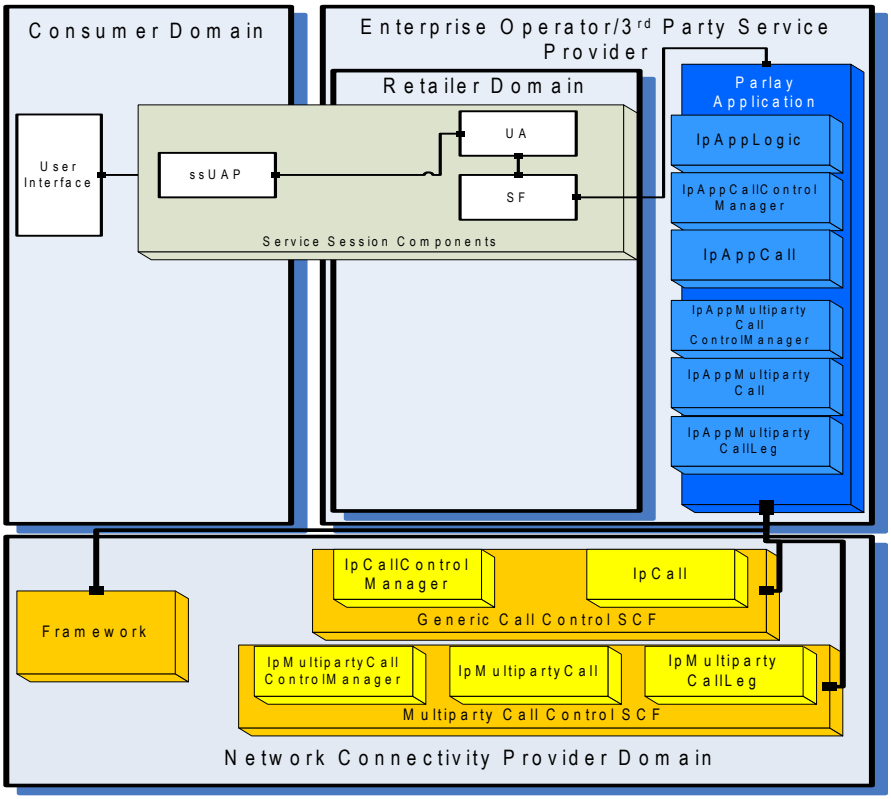

Figure 6.5: TINA Service Session Components and Parlay's Generic and Multiparty Call Control SCFs

6.2.3.2 The Multiparty Call Control SCF

The Multiparty Call Control SCF is described in Section 2.2.2.2. The MCCS implements the IpMultipartyCallControlManager, IpMultipartyCall, and IpMultipartyCallLeg interfaces which allow it to create multiparty sessions involving more than two users. Applications must implement the IpAppMultipartyCallControlManager, IpAppMultipartyCall, and IpAppMultipartyCallLeg interfaces to interact with it. The MCCS will be used to illustrate a scenario where a user uses an invitation to join an already existing service session.

Appendix A.3.2 provides a summary of all the interfaces presented in Section 6.2. Figure 6.5 illustrates the relationship between TINA's service session service components and the Parlay Generic Call Control SCFs IpCCM, IpCall, and their corresponding call back interfaces. Parlay's Multiparty Call Control SCF is also shown.

6.2.4 Proposed Implementation Scenarios

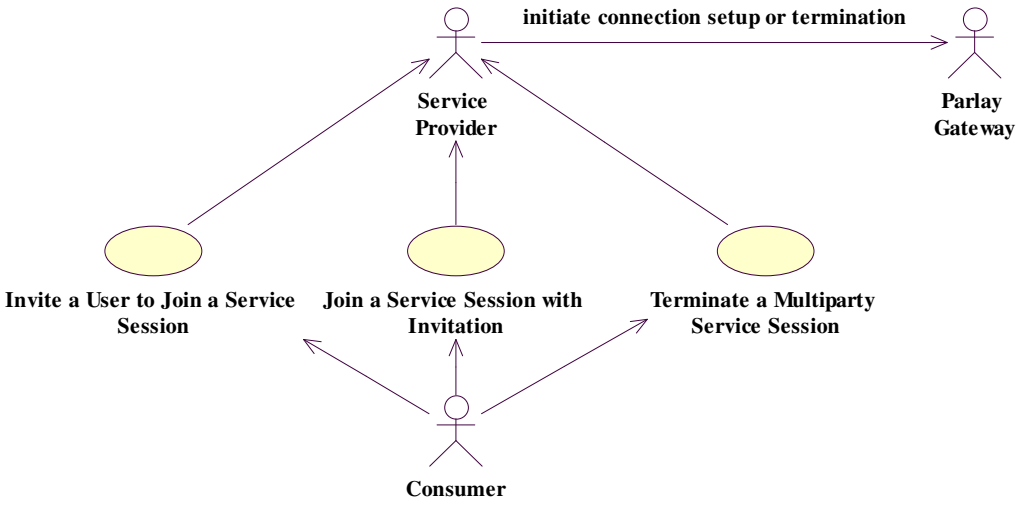

Figure 6.6: Multiparty Service Usage Management Use Case Scenarios

The multiparty service usage management use case scenarios are illustrated in figure 6.6. Their function is shown using sequence diagrams in the following chapters.

6.2.4.1 Invite a User to Join a Service Session (Initiate a Multi Party Service Session)

Description:

A user participating in a service session invites another user participating in the same service session to direct communication. The invitation contains sufficient information for the UA to locate the service session and allow the user to join it. On completion, the inviting user is receives a reply indicating an acceptance or refusal of the invitation.

Pre-Condition:

The ssUAP, Inviter's UA (user A), Invitee's UA (user B), and App are present in their respective domains. A service session exists for the user sending the invitation. It is not necessary for the invited user to have an active access or service session.

Post-Condition:

The inviting user has a reply to the invitation. A service session may or may not be setup between user A and user B.

Message Sequence Diagram

1. The User (inviter) uses the UI to request connection setup to another user. The UI instructs the ssUAP to issue an invitation to a potential participant (invitee) to join a session. The inviter supplies a resolvable name/address of the invitee.

2. The ssUAP instructs the UA to invite a user to join the session.

3. The UA provides a resolvable name and address of the invitee and instructs the application to begin routing the call.

4. This message is used to deliver the invitation request to the invitee's UA. Also in this case a response is requested.

The Invitee's UA may perform some actions before continuing. For example, the UA may check the user profile within the UA for a policy on invitations.(A user may for example set the policy to reject all invitations from unknown users.) The policy will then determine the UA actions and interactions with other objects.

5. The invited user replies to the invitation. Note that in the diagram it is assumed that the invitee accepts the invitation. In this case, a connection will be setup between the two parties.

6. The App indicates to all other UAs that a new user has been invited.

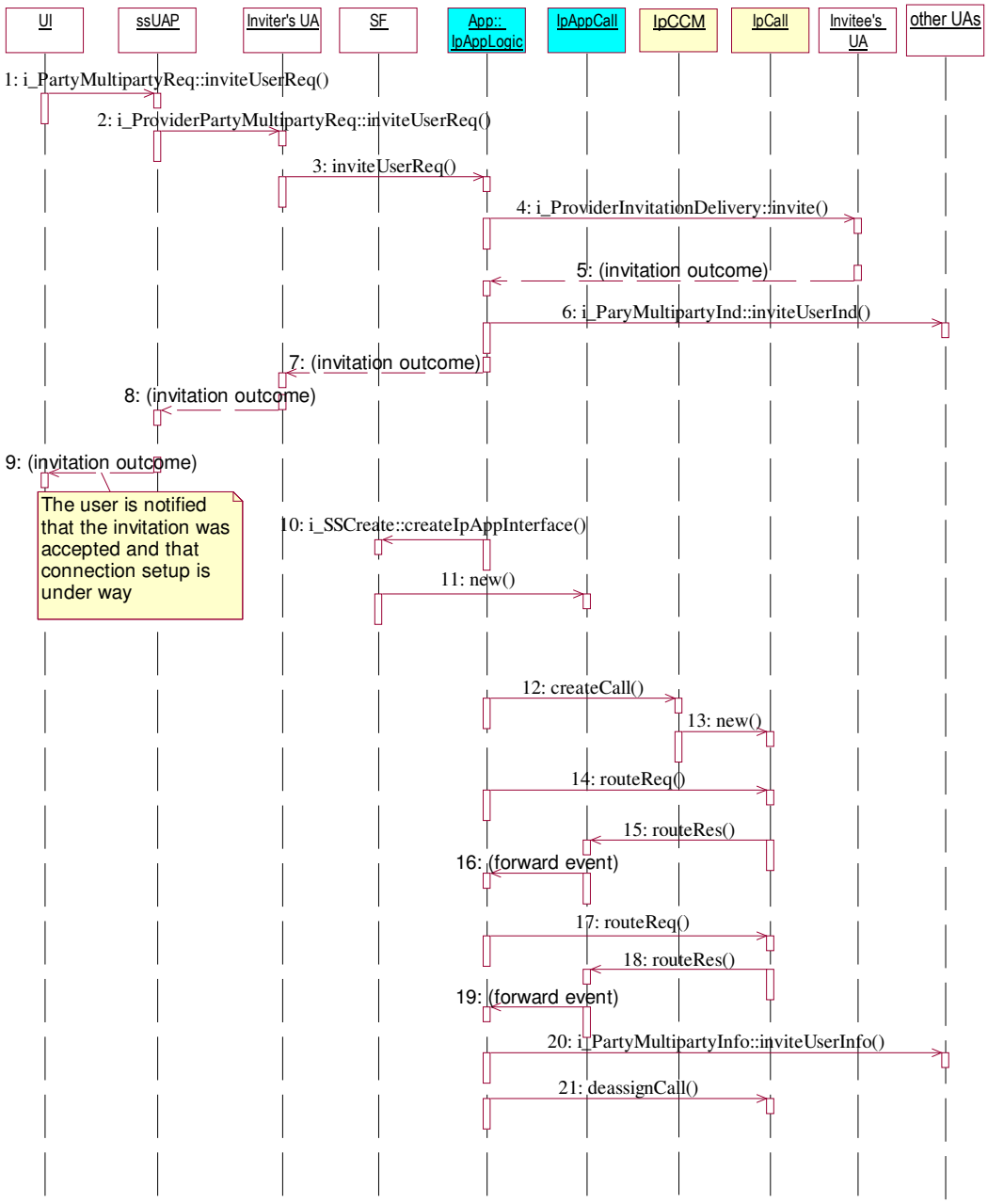

Figure 6.7: Invite a User to Join a Service Session Sequence Diagram

7. The App informs the Inviter's UA of an invitation reply.

When the invitation has been successfully delivered the App sends an **inviteUserInfo()** to the ***i_PartyMultipartyInfo*** interface of all the UAs that are currently involved in the session, (except the requesting party). However, in this case the Generic Call Control SCF is used so only two users can be involved in a service session.

8. The invitation reply is forwarded to the ssUAP.

9. The invitation reply is forwarded to the user. The user is notified that his invitation was accepted and a connection between himself and the invited user is about to be setup.

10. The App requests the SF to instantiate a new IpAppCall object.

11. The SF creates an object implementing the IpAppCall interface

12. The App requests the IpCallControlManager to create an IpCall object

13. The IpCallControlManager creates an IpCall object.

14. The App requests the IpCall object to route a call to the inviter (the inviting user).

15. The IpCall object returns a message indicating that the inviter answered the call.

16. The message is forwarded to the App.

17. The App requests the IpCall object to route a call to the invitee (the invited user).

18. The IpCall object returns a message indicating that the invitee answered the call. At this point a connection has been successfully setup between the two users.

19. The message is forwarded to the App.

20. The App informs all other UAs with information about the invited user.

21. Since the application is no longer interested in controlling the call, the application deassigns the call. The call will continue in the network, but there will be no further communication between the call object and the application. (The application may continue to monitor call events if it so wishes by invoking the enableCallNotification method on the IpCallControlManager)

In the example above an access session already existed between user B's PA and UA. If user B is NOT currently in an access session with the UA, then there are several alternatives as to what happens. The alternatives are:

- UA stores the invitation until the invited user establishes an access session. When he does establish an access session, the invitation is delivered as above

or

- UA delivers the invitation to a registered terminal. (The terminal would have been selected by the user to receive invitations when no access session was present)

6.2.4.2 Join a Service Session with Invitation (Establish a Multi Party Service Session)

Description:

A user (user B) participating in a service session accepts the invitation from another user (user A) to engage in direct communication with him/her. User B joins this session from

any terminal, from which he has established an access session. The inviting user is informed of the decision of the invited user. On completion, the two users have a connection setup and may engage in direct communication with each other and users already involved in the session are informed of the new user.

Pre-Condition:

A service session utilizing the MPCCS has been setup by the inviter (user A).

An access session has been setup by the invitee (user B).

Post-Condition:

User A and User B are involved in the same service session.

Message Sequence Diagram

1. The user sends a request to the ssUAP to join a session with an invitation.

2. The ssUAP forwards the request to the UA.

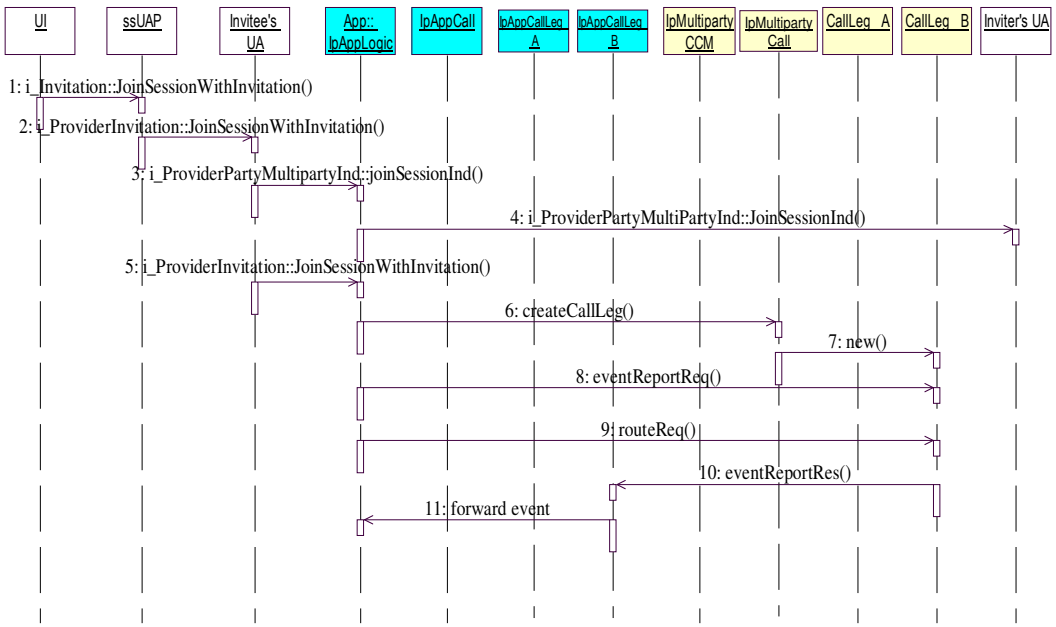

Figure 6.8: Join a Service Session with an Invitation Sequence Diagram

3. The UA requests a check on the validity of the invitation (i.e. does the session still exist?...Can a new user join the session?, etc) from the App. If the invitation is no longer valid, the UA will reject the request.

4. The App forwards the request for a check on the validity of the invitation to the Inviter's UA. The inviter can then choose to allow the invitation, or to deny it. In the case of a denial, an exception is raised, and the User must begin a new service session. In the case of an acceptance, a connection is setup and the user is allowed to join the session.

5. Assuming the invitation is still valid, the UA forwards the requests to join a session, to the App.

6. The App requests the IpMultipartyCall to create an IpCallLeg object for the user B.

7. The IpMultipartyCall creates an IpCallLeg object.

8. The App requests the IpCallLeg object for user B to inform the application when the call leg answers the call.

9. The call is then routed to user B's call leg.

10. Assuming the call is answered, the object implementing user B's IpCallLeg interface passes the result of the call being answered back to its callback object. At this point a connection has been successfully setup between the two users. (Note: There may be other users currently active in the session or other users may still join the session.)

11. The message is forwarded to the App.

6.2.4.3 Terminate a Multi Party Service Session

Description:

A user ends the service session that he/she created. On completion all participants of the service session are removed and the user is ready to start a new service session.

Pre-Condition:

The components required to sustain a service session for all participants are present in their respective domains.

Post-Condition:

Accounting information is returned to all parties that were involved in the session.

6.2.4.4 Message Sequence Diagram

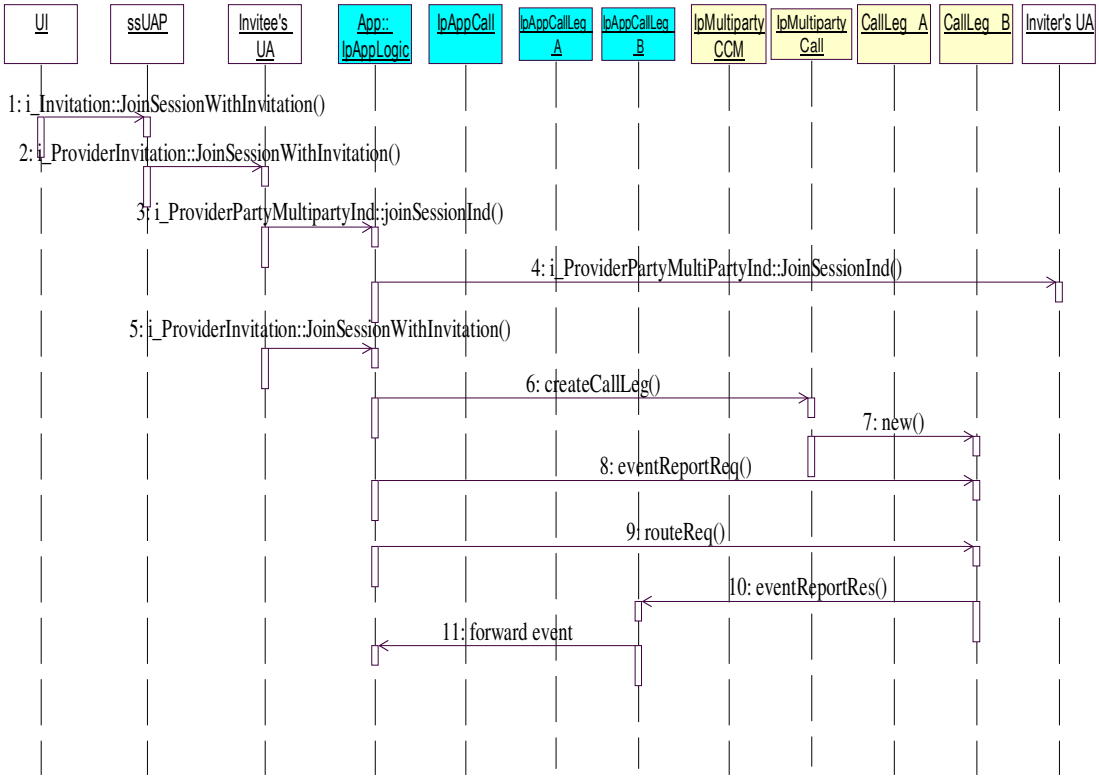

Figure 6.9: Terminate a Multiparty Service Session Sequence Diagram

1. The User Interface requests the session to be ended by invoking the endSessionReq() on the ssUAP.

2. The request is passed on to the UA. The return value of this invocation is the accounting information for the requesting user.

3. The UA requests the Application to end the service session. The return value of this invocation is the accounting information for the requesting user. The UA stores this information in the user's profile.

4. The App sends an indication about the request to end the session to all the UA's participating in the session.

5. The UA sends an Exe to the Application instructing it to end the session. The Application passes Accounting information back to the UA with this invocation, thereafter, the service session specific interfaces are made unavailable and all accounting is stopped.

6. The App passes the instruction to end the session to the respective UAs. (However, no connection exists at this point.)

7. The application terminates the call.

8. IpCall informs the callback interface that the call was successfully ended.

9. The message is forwarded to the Application.

10. All other participating UA's are informed about the session being ended and accounting information is exchanged.

11. The SF is informed that the session is about to end and eventually releases the Application specific resources.

12. The SF instructs the App to releases all references to the callback objects. (These references will no longer be valid)

13. The App deletes all references.

6.3 Context Aware Service Delivery

In order to support context aware service delivery, a new interface defined by the author is implemented. The ***i_ContextManagement*** interface implements a set of methods which can be used to access context information. The methods implemented are categorized according to context type and are shown below:

It is important to note that the consumer interface's support of context awareness is not completely service independent, but requires supported OSA/Parlay applications to already have subscriptions to SCFs which provide context awareness. Applications may be requested by the end user interface to provide different types of context information based on which SCFs they are subscribed to. For example, if an application is not subscribed to the mobility SCF, location based service delivery cannot be provided by the consumer interface.

6.3.1 Device Characteristics

The Terminal Capabilities SCF (TCS) can be used to retrieve terminal capability information. The following user defined methods are implemented to support the retrieval and reporting of terminal capability information to and from the UA:

- **getTerminalCapabilityInfo()** is used to retrieve terminal capability information from the TCS. One input parameter is specified, *entityId*, and one output parameter *t_terminalConfig*, which contains terminal capability data. (see Section 3.4.1)
- **monitorTerminalCapabilityReq()** is used to request for terminal capability reports when the capabilities change. Only one input parameter, *entityId*, is specified. There are no output parameters.

- **monitorTerminalCapabilityReport()** provides periodic reports on terminal capabilities when the capabilities change. Only one input parameter *t_terminalConfig* is specified.
- **monitorTerminalCapabilityStop()** is used to stop the monitoring of changes in terminal capability information of an entity. It takes an *entityId* as a parameter.

Appendix B.2 shows the login to a service provider sequence diagram (excluding the authentication part) with added context awareness provided by the terminal capability SCF. In this diagram, the UA requests added terminal capability information to supplement that provided by the PA and sets a monitor to provide reports in case of a change in capability.

6.3.2 User Preferences

User Preference information is obtained during the subscription phase when negotiating the service contract. It is the only manually obtained context component.

6.3.3 User State

6.3.3.1 User Location Information

Physical location information can be retrieved using the Mobility SCF.

- **getLocationInfo()** requests a report on the location of a set of end user. It has one input parameter, *entityIdList*. and returns *locations* which specifies the location(s) of one or several users.
- **monitorLocationInfoReq()** requests periodic reports on the location of a set of end users. It has one input parameter, *entityIdList*, and no output parameters.
- **monitorLocationInfoReport()** returns periodic reports on the location of a set of end users. It has one input parameter, *locations*.
- **monitorLocationInfoStop()** terminates the periodic reporting of user location information for a set of end users. It has one input parameter, *entityIdList*, and no output parameters.

6.3.3.2 Application Environment

Determining the state of the user's application environment is partially handled using the *t_ApplicationInfo* structure in the *t_TerminalConfig*. The parameters are passed from the user to the consumer domain to the service provider during access session setup (see. **setUserCtxt()** in Section 4.1.3).

6.3.4 Proposed User Context Implementation Scenario

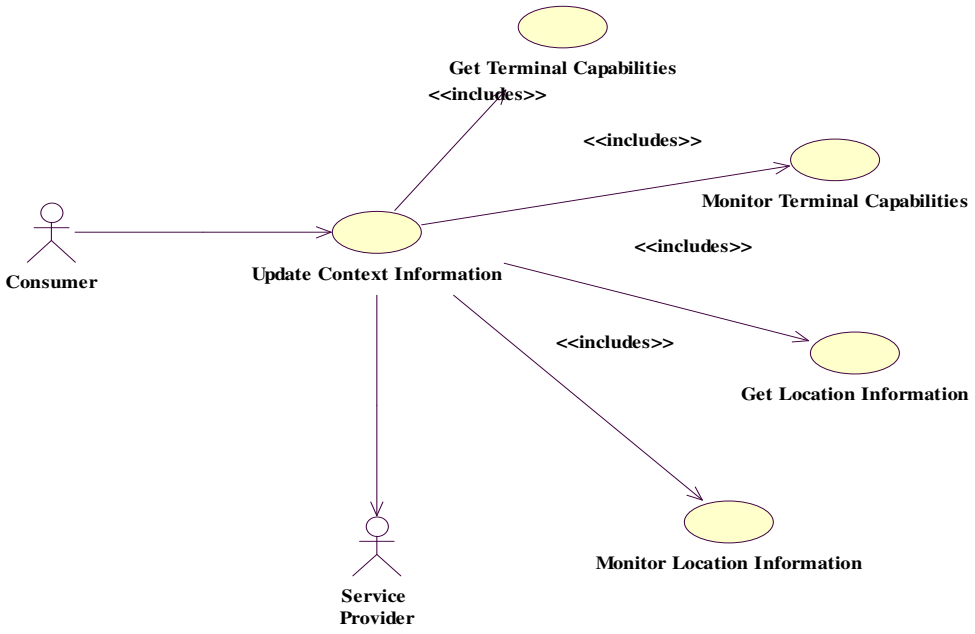

Figure 6.10: User Context Management Use Case Scenarios

6.3.4.1 Update Context Information

Description:

A User Agent participating in a service session requires an update of context information before making a decision on service delivery. The UA requests the App for updated information as well as periodic updates whenever any changes in context occur. Upon completion, the UA has updated its context information and will receive updates periodically when context changes.

Pre-Condition:

The App is subscribed to the SCF providing the requested context information.

Post-Condition:

The UA receives periodic updates from the App whenever context changes.

Alternate Flows:

The UA is participating in an access session setup and no service sessions exist. The UA requests the App for updated information as well as periodic updates whenever any changes in context occur. Upon completion, the UA has updated its context information and will receive updates periodically when context changes.

Message Sequence Diagram

1. The user requests terminal capability information. He/she specifies the *entityId* for the specified terminal.

2. The ssUAP forwards the request to the UA.

3. The UA requests the Parlay Application (App) to retrieve terminal information using the Terminal Capability SCF.

4. The App retrieves the terminal capability of the specified terminal.

5. The user requests terminal capability reports when capabilities change.

6. The request is forwarded to the UA.

7. The UA forwards the request to the Parlay Application.

8. The Application requests the SF to create an IpAppExtendedTerminalCapabilities callback interface.

9. The SF creates an IpAppExtendedTerminalCapabilities callback object.

10. The SF passes the reference to the callback object to the Application.

11. The terminal capabilities changes are started to be monitored.

12. The terminal capabilities have changed and they are reported as requested.

13. The report is forwarded to the Application.

14. The application forwards the report to the UA. The UA may store the terminal capability information at this point.

15. The UA forwards the report to the ssUAP.

16. The ssUAP forwards the report to the user.

17. The user requests the terminal capability monitoring to stop.

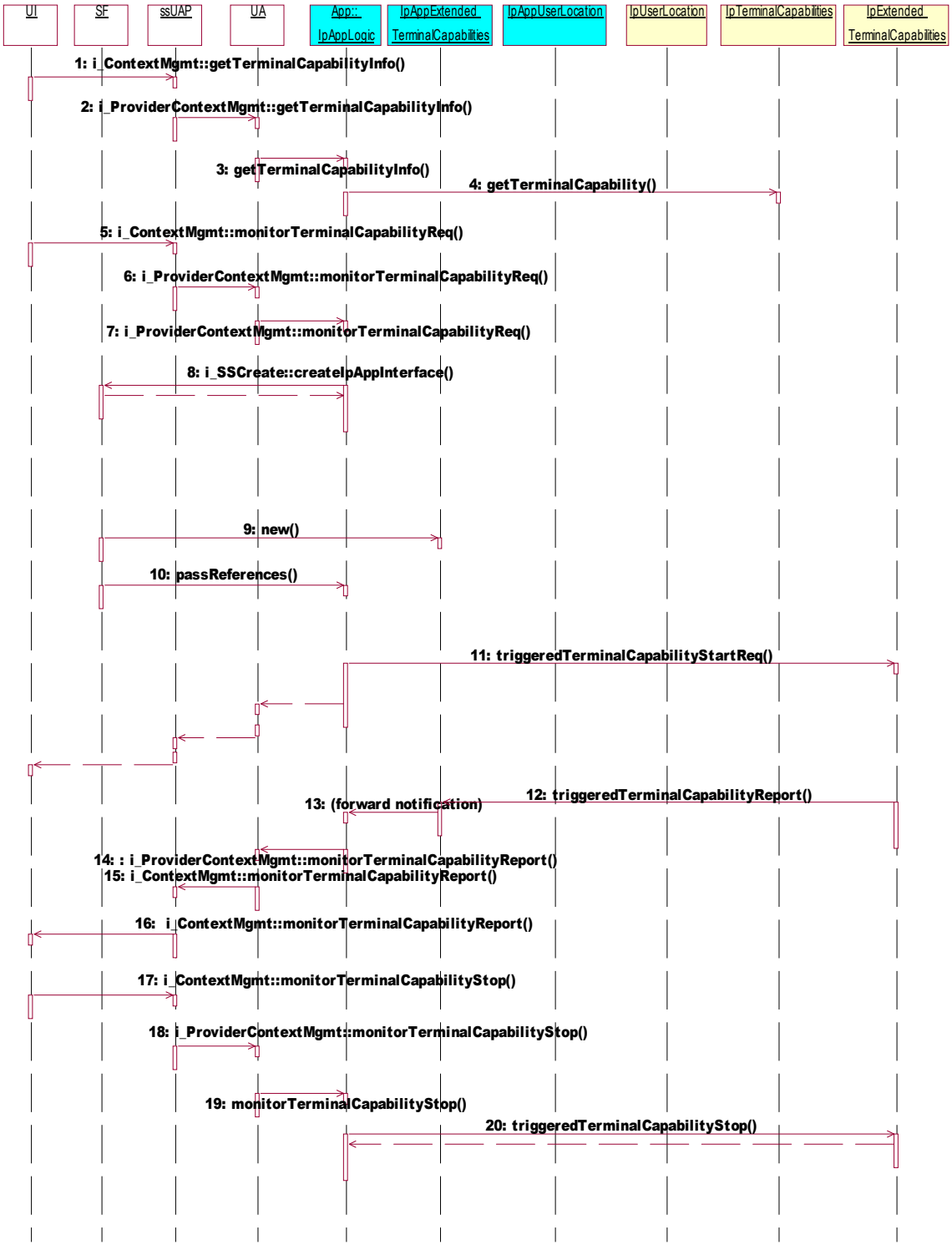

Figure 6.11: Update User Context (Terminal Capability Update) Sequence Diagram

18. The request is forwarded to the UA.

19. The UA forwards the request to the Application.

20. The Application stops the terminal capability monitoring.

The updating of Location Information is performed in the same way. Appendix B gives an example of Location Information updating.

6.4 Chapter Summary

This chapter presented the end user interface's Service Usage Management API. The main purpose of this API is to initiate and terminate single and multiparty OSA/Parlay applications. Both single and multiparty OSA/Parlay services were initiated using TINA's ***i_Access***, and ***i_SSManage*** interfaces, and terminated using the BasicFS. However, the multiparty service termination scenario required the support of the Multiparty and MultipartyIndFS. The single party scenarios were illustrated using the Parlay's Generic Messaging SCF, whereas the multiparty scenarios were illustrated using the generic and conference call control SCFs. Users involved in multiparty service sessions were also allowed to invite other users to join them, as well as use received invitations to join already existing service sessions. These features were supported by TINA's Multiparty and MultipartyInd FS which provided methods to support user invitation requests, the distribution of indications and information of session events such as ending of a session, a user being invited, or a user joining a session, to all users involved in a session. TINA's ***i_Invite*** interface supported the multiparty invitation scenarios by providing methods to allow clients to send invitations, as well as view sent invitations or use them to join existing service sessions.

An integrated view of the hybrid OSA/Parlay – TINA service architecture was provided by using sequence diagrams and use cases to describe the interaction between the TINA service components and Parlay's Applications and SCFs for the single and multiparty service initiation and termination, as well as the invitation scenarios.

The chapter concluded with a discussion of Context Awareness. A new author defined interface to support context awareness in the consumer interface was created. The ***i_ContextManagement*** interface provides a set of operations which can be used by to access context information from Parlay's Terminal Capabilities, and User Location SCFs. The behaviour of service components implementing this interface and their interaction with OSA/Parlay applications and SCFs was shown using sequence and use case diagrams.

7 IMPLEMENTATION ENVIRONMENT

In this chapter we discuss the implementation environment used to realize the OSA/Parlay end user interface. We also discuss the distributed processing environment used to implement the service components.

7.1 The CORBA DPE

NGN applications are typically distributed in a heterogeneous environment that comprises different hardware platforms, operating systems, databases, and network protocols. CORBA is a distributed computing architecture, which supports the development of systems for heterogeneous computing environments. It supplies a set of abstractions and services needed to realize practical solutions for the problems associated with distributed heterogeneous computing [45]. CORBA provides the following for distributed application developers:

- **Implementation independent interface definition**

  CORBA defines an Interface Definition Language (IDL) to support the separation of interfaces from distributed object application implementations. IDL provides high level definitions for all methods on the distributed objects. It is not a programming language and thus objects and applications cannot be implemented in it. The main purpose of the IDL is to allow object interfaces to be defined in a manner that is independent of any particular programming language [45].

- **Programming language transparency**

  CORBA supports multiple language mappings for OMG IDL so that different parts of a system or application can be implemented in different programming languages. For example, in C++, IDL interfaces are mapped to classes, and operations are mapped to member functions of those classes. Similarly, in Java, IDL interfaces are mapped to public Java interfaces. This arrangement allows applications implemented in different programming languages to interoperate. The language independence of IDL is critical to the CORBA goal of supporting heterogeneous systems and the integration of separately developed applications [45].

- **Location transparency**

CORBA provides location transparency, which means that an object is identified independently of its physical location and can potentially change its location without disrupting the system. The CORBA Object Request Broker (ORB) provides the necessary mechanisms for this transparency. Generally, an ORB enables communication between components over a network.

- **Automatic code generation to deal with remote invocations**

Vendor specific IDL compilers create language specific representations of IDL defined constructs such as constants, data types, and interfaces. The generated code for the client side, that is, the code invoking an operation on an object, is known as the client stub. Client stubs allow a request invocation to be made via a normal local function call. The server-side generated code is called the skeleton. A skeleton allows a request invocation received by a server to be dispatched appropriately [45].

- **Access to standard CORBA services and facilities** such as [45, 46, 48]:
  - Naming
  - Trading
  - Notification
  - Transaction
  - Persistent State
  - Security

7.2 Network Implementation

The real-time ACE TAO ORB [48], a CORBA Version 3.0 compliant Object Request Broker (ORB), was used to provide the distributed processing mechanism for the implementation. In this implementation, all computational objects and interfaces were developed in C++. The service components were compiled and installed on a TAO version 1.3.3 ORB on a Linux SUSE 9.0 operating system. A fundamental requirement of the consumer interface is to be able to distinguish between service components and their interfaces. The hybrid OSA/Parlay – TINA service architecture therefore supports a framework for referencing components. In order to simulate the distributed NGN environment each component utilizes a CORBA reference resolution mechanism called the IOR (Interoperable Object reference) which provides location transparency, and location independence. An IOR allows an application to make remote method calls on a CORBA object. Once an application obtains an IOR, it can access the remote CORBA

object. The IOR contains all the information needed to route the message directly to the appropriate server.

Each service component is installed in a separate directory and stores an IOR for each of the interfaces it has implemented. Object references are non-transparent to applications, but they contain information that ORBs require to establish communication between service components.

Object references can be made available in several ways. The following examples provide some approaches however they are not all inclusive. They can be published in a shared folder to be read by client applications, or to a known location in the server's document root using a servlet or CGI script, enabling client programs to easily retrieve stringified references from the Web server using the HTTP protocol [45, 46], or through a Naming Service [46, 48]. In this research, we implemented all the components on one machine and thus we publish an object reference by converting it to a string and writing it to a file stored in the same directory as the relevant service component. Initially, the asUAP has a reference for the PA, and the PA has a reference for the SPF, however, service components generally acquire object references in response to successful operation invocations. For example, the PA receives an object reference for the UA's i_ProviderAccess interface after the successful invocation of a requestAccess() method.

In this research, the functionality of the consumer premises end user interface was tested according to the sequence diagrams presented in Chapters 4, 5, and 6. The next section presents a discussion of the results.

7.3 Implementation results

The OSA/Parlay end user interface's functionality was tested by implementing the computational objects and examining whether they were able to execute the interface methods correctly and in the sequence specified in the proposed implementation. The results are presented in this section.

7.3.1 Access Session APIs

The main goal of the access session components is to establish a secure context between the consumer and service provider thereby allowing an end user access to services in a controlled manner. The consumer domain service components were able to successfully initiate contact with the service provider on behalf of the user.

In terms of authentication, the combination of usernames (userId) and passwords (userProperties) for known users can be used by service providers to support user

authentication. Domain authentication was emulated by invoking the sequence of authentication methods prescribed in the proposed implementation scenarios, however, no actual authentication algorithms were implemented.

The setup of default context was achieved by passing terminal and application information between the consumer and service provider domains. It was assumed that this information was known by the PA and passed to the UA during access session setup. A service provider could use the UA to provide context by storing the application and terminal information and examining the stored information at the appropriate times (ie. when starting a service, or receiving an invitation).

Consumer service discovery was successfully emulated by passing parameters that allow a user to select desired properties of a service, and returning a list of matching services, and service ids based on the specified desired properties.

Access session termination was successfully emulated by implementing the sequence of termination operations as specified in the "logout of a provider" scenario in Chapter 4.

7.3.2 Subscription and Profile Management

With regard to the management of subscriber, information structures designed for subscription and profile management as shown in Chapter 5 were successfully passed between the service provider and consumer. The interfaces, methods, and attributes proposed in the sequence diagrams were successfully implemented. The consumer interface successfully emulated user subscription by passing subscriber information, service templates, SAE lists, SAG lists, service profiles, and service contract information between the consumer and service provider domains. This was done using the ssUAP, UA, and SUB components as proposed in the design.

The sequence of operations proposed to support the modification of subscription information were also successfully implemented. Therefore, the system provides service providers with the necessary information structures to be able to acquire consumer subscription information, and using the defined relationships between the structures, provide the necessary mappings within a database to manage user and service information. With regard to the consumer, the end user interface supports the passing of subscription information to the service provider. This includes preference information regarding services and consumer premises entities. Consumers are also able to modify their subscription information (ie. service profiles, SAG and SAE lists, service templates, etc) through a comprehensive set of operations as specified in Chapter 5.

7.3.3 Service Usage Management

The main aim of the service usage management API was to support single and multiparty call connection setup for OSA/Parlay applications. Using the ssUAP, users were able to request the initiation and termination of single and multiparty service sessions from the service provider's UA, SF, and App components across the DPE as specified in the proposed implementation scenarios. Invitations were successfully passed between domains during the emulation of the invite user and join a service session scenarios. The provision of context awareness was supported by the UA. Using the set of available operations and subscription identifiers (ie. subscriberId, entityId, etc), service providers implementing the consumer interface can use the UA to access any consumer information stored by the SUB. This means that the UA can access any subscription information and use it to make decisions regarding service delivery context. The SF successfully emulated the creation of OSA/Parlay specific interfaces by creating dummy object references and passing them to the App when requested.

7.3.4 Context Awareness

The provision of context awareness in service delivery is discussed in the previous sections. It was stated Section 7.3.1, that *user state* (the state of the applications in the user's environment) and *device characteristics* information is provided by the PA to the UA during access session setup.

The provision of *user preference* information is also supported. Interfaces, operations, and data structures to support user preference information storage are defined and implemented, however, no specific services are demonstrated. It is left to the service provider to support user preference by specifying configurable and non configurable service related information within the service template, and thereafter allow the consumer to define preference by modifying the configurable information when negotiating the service contract. The implementation of the i_ContextManagement interface is suggested and not implemented as the 'proof of concept' did not go as far as to link the Parlay consumer interface with an actual gateway.

7.3.5 Personal Mobility

To provide personal mobility within a Parlay service system, we stated that the requirements specified in Chapter 2.3 would have to be met.

- Users should have the ability to register so that service invitations are sent to a terminal specified in the registration.

This is covered during subscription. Operations and information structures are provided to allow a subscriber to register a set of terminals, each of which is allowed to access services according to some assigned rights and privileges. The implementation of a registration object accessible to the UA is necessary for this requirement to be fulfilled, however, this was not possible due to time constraints.

- Users may respond to a service invitation on a different terminal than the one specified for getting the invitation.

  Users are allowed to list their invitations from any terminal which they are currently using, therefore, a user may list an earlier stored invitation on any chosen terminal, and use it to respond to a service invitation.

- The user should be able to "access the system" according to the service provider's and user's own preferences, as much as possible, independent of the terminal used. The delivery of a service is subject to terminal and network capabilities.

  The OSA/Parlay architecture provides functionality to request for terminal capability. Terminal capability information is also given by the user during access session setup. Before initiating a service session, the user's UA can check the user's preferences and whether the user's terminal is capable of participating in the service session and takes appropriate action. (User preference is specified during subscription).

8 CONCLUSION

8.1 Discussion

This report examined service creation and delivery in the NGN, specifically the rapid creation and delivery of cross network applications enabled by network independent APIs. The main focus was on the OSA/Parlay service architecture. The OSA/Parlay Architecture enables application convergence through the specification of open standard APIs which allow access for applications to transport network functionality in a manner independent of the transport network used. They also provide a powerful set of Service Capability Functions that can be used to provide context awareness and location based service delivery.

OSA/Parlay does not however identify a consumer or service provider domain in its application layer. This means that 3rd party applications housed in the service provider domain have no defined interfaces to manage service delivery to the consumer domain. The OSA/Parlay model does not specify how applications manage user access, authentication, service usage or subscription and profile management.

The lack of a standardized set of APIs to support end user service access and usage management presents a number of significant concerns for service providers. Without an API to support user access to services, service providers cannot offer secure, customized access to the set of OSA/Parlay services. Operations to support user and domain authentication are necessary to ensure the establishment of a secure usage context between the consumer and service provider. Also, in order for service providers to offer services to end users, it is essential that they have sufficient information to handle end users, subscribers, and the subscription life cycle. This requires a subscription management API within the consumer domain to be implemented. Customized access to services can then be arranged by taking into account context information such as end user preferences. It is also important for service providers to be able to provide users with the ability to manage their services during service usage. Parlay offers no API to initiate or terminate a service, or even to invite other users to join an existing service.

The main research problem was to design and implement a standard consumer interface which could be used by application providers within an OSA/Parlay system to deliver service content to end users. The main objectives with regard to the functionality provided by the interface included the integration of facilities which would assist application providers to manage end user access and authentication (to enable users to establish a secure context for service usage), subscription (to handle the subscription life

cycle), and service usage management (to enable the initiation and termination of services). The integration of NGN capabilities such as context awareness, location based service delivery and personal mobility into the consumer interface was also examined.

In order to achieve the defined objectives we explored the feasibility of using the TINA service architecture within an OSA/Parlay system to support consumer domain service delivery. The TINA architecture provided a comprehensive set of concepts and principles that could be used in the design of NGN services. Mainly the TINA service architecture provided a logical separation of business roles and administrative domains within Parlay's application layer. This allowed us to define a business model which provided a high level definition of the OSA/Parlay consumer interface. The model logically separated the Consumer, Service Provider/Enterprise Operator, and Connectivity Provider/Parlay Gateway, and assigned responsibilities to each. TINA's Retailer Reference point [23], was then used to specify the relationships between the roles in order to define a generic service independent API set between Parlay's consumer and enterprise operator domains. TINA's component model was then used to define a set of reusable and interoperable service components which encapsulated the generic APIs to provide Access and Authentication, Subscription, and Service Usage management functionality to the OSA/Parlay consumer interface. The encapsulated API also provided inherent support for personal mobility. Support for context awareness was provided by the subscription and profile management API which allowed users to register their preferences. The Access API also allowed a user to send some consumer domain information which could be used to provide context during service delivery. TINA's session model was integrated into the end user interface to clarify the purpose of interactions between consumers and service providers. The implementation of the component model, and Ret RP interfaces using a set of implementation scenarios provided a reusable consumer interface for the Parlay service architecture. The implementation was demonstrated by deploying the service components over a distributed computing platform.

8.2 Conclusion

This project set out to describe the need for a consumer to service provider interface to support the provision of OSA/Parlay applications in an NGN environment. An underdeveloped OSA/Parlay service architecture lacking the necessary functionality to provide efficient service delivery between the consumer and service provider domains provided the basis for this study. In conclusion, we can say that this research has demonstrated that TINA concepts and principles are useful in the design and implementation of APIs to support consumer premises service delivery in an OSA/Parlay environment. Utilizing the TINA service architectures rich set of features and comprehensive API, we were able to design and implement a standard Consumer to Provider API set for Parlay applications. The interface provided the generic logic

required to support OSA/Parlay service delivery by providing an interworking between TINA and OSA/Parlay specific logic.

The OSA/Parlay consumer interface allows for the specification and management of a relationship between a service provider and a set of consumers within the Parlay service architecture. An important conclusion we can make is that the end user interface provides the necessary logic to support the creation and provision of a diverse set of Parlay applications through use of generic and reusable service logic. Within the research, Parlay applications implementing the Generic Messaging, Generic Call Control, Multimedia Call Control SCFs were demonstrated, however, the end user interface is capable of providing support to any applications implementing other SCFs. Using TINA's business model, reusable service components, session concept, and Retailer Reference Point, the consumer interface is able to provide the following functionality to any applications with access to the relevant Parlay SCFs:

- **Access and Authentication**

  The access and authentication API allows consumers to transparently locate a OSA/Parlay service providers and request the establishment of an access session. The functionality provided by the API are encapsulated by the asUAP, PA, UA, and SPF computational objects. The asUAP and PA are the initial point of contact for the consumer, and allow him/her to authenticate him/herself and/or the consumer domain to a service provider specific SPF, as well as to establish a secure and controlled access session to the system of OSA/Parlay services. The PA provides ubiquitous access to the system of OSA/Parlay services, irrespective of the terminal being used and the point of attachment to the network. The access session also allows the consumer to discover services, initiate usage of those services, and register the consumer's context with the service provider through a UA. Registration of consumer context provides a user with the opportunity to customize access to services, and for the service provider to customize the delivery of services based on the end user's preference and/or terminal capabilities.

- **Subscription and Profile Management**

  The subscription and profile management APIs provide OSA/Parlay service providers with capabilities to manage the set of consumers subscribed to their services, and subscribers with the capability to manage their services and service entities. Using the subscription API users can subscribe, modify and release services offered by providers. Using the profile management API users can manage user, terminal, NAP and service information and preferences. The functionality provided by the API are encapsulated by the ssUAP, UA, and SUB service components. The subscription and profile management API allow subscribers to control context information by defining their preference when negotiating a service contract. This allows subscribers to tailor a service's behavioural characteristics to their specific requirements and needs.

We can conclude that without the subscription API the OSA/Parlay service providers would be unable to identify or authenticate users as no previous user or subscriber information would be available. Access to services would be limited to anonymous users, complicating service delivery as well as billing. Context awareness would also be limited. Only dynamically obtained information from the OSA/Parlay gateway would be accessible, meaning that users would be unable to register their preferences and would therefore be unable to customize services.

- **Service Usage Management**

  The service usage management APIs cover the interaction between the consumer and service provider domains during the use of a service session (based on policies agreed upon in the service contract). These API are key to the interworking between TINA and OSA/Parlay specific logic as they interact with OSA/Parlay applications to initiate connection setup and management with the gateway on behalf of consumers. The functionality provided by these API enable consumers to initiate and terminate single or multiparty multimedia service sessions. Their functionality is encapsulated by the ssUAP, UA, and SF service components. An App component is used as a generic representation of OSA/Parlay services.

- **Context Aware Service Delivery**

  The integration of context aware service delivery into the OSA/Parlay consumer interface was a key problem in this research. To deal with this problem we provided a definition of context and required that service delivery be adaptable based on the defined context. Using TINA's information structures and subscription API we were able to support service delivery based on user preference and user state (ie. user domain specific information). Dynamically obtained context such as Terminal and User state (ie. Location information) was provided by the Parlay SCFs, it should be noted however that obtaining dynamic context was suggested but not implemented. In conclusion, we were able to provide adaptable service delivery based on a defined context, incorporating device characteristics and user context, by exploiting available information structures within the TINA API and suggesting a set of interfaces to interact with service capability functions within the Parlay gateway, in order to create a unified context management system within the Parlay end user interface.

- **Personal Mobility**

  Personal mobility was also a key feature which we intended to integrate into the consumer interface. Achieving this objective was facilitated by the TINA service architecture's inherent support for personal mobility. The integration of personal mobility into the consumer interface provides end users with the ability to

ubiquitously access customized services independently of physical location or specific equipment.

It is important to realize that the Interfaces and operations presented within this research to provide the abovementioned functionality present a guideline which can be used by service providers to provide standardized functionality within Parlay's consumer domain. The defined end user interface is by no means intended to provide an exact solution, however, it is expected that the defined service components and APIs will provide a framework which service providers can 'build around' with some degree of freedom. For example, although during access an authentication sequence is described, no formal algorithms are prescribed. The implementation specifics with regard to Access and Authentication, Subscription, and Service Usage are left to the enterprise operator.

8.3 Recommendations for Future Work

The goal of this project was to provide a "Proof of Concept" case study which explored how TINA concepts could be used to support the provision of service in an OSA/Parlay environment. The study mainly explored how the generic functionality defined by TINA in the areas of access and authentication, subscription and profile management, and service usage management could be interlinked with OSA/Parlay applications to increase efficiency in service creation and deployment.

The consumer interface could have been extended by including generic logic to handle the addition and deletion of services. It was initially hoped that the SPF would be able to take up a similar role as the OSA/Parlay Framework within the consumer interface. This would have implied a 'priority' role for the SPF in service registration. Another responsibility would have been integrity management of OSA/Parlay applications. It may be worthwhile to explore the role of the SPF in terms of monitoring application load conditions, and taking appropriate action according to load condition changes or in the case of failure, as well as service registration.

In terms of service usage management, the API could be extended to enable session mobility. Session mobility allows a user that has an active session on a particular terminal to move that session to another terminal. In order to enable session mobility, the consumer interface is required to allow a user to suspend a service session on a terminal, and resume participation using a different terminal. Furthermore, the end user interface must assess how the requirements of the service match the capabilities of the new terminal and access point and adapt the way the service content is delivered to the end user [4]. The next section examines content adaptation and considers some important issues for its enablement.

8.3.1 Content Adaptation

The NGN amalgamates different types of media designed for use on different types of devices. However, different types of media such as images, audio, and video are usually designed with specific devices in mind [26, 28]. The OSA/Parlay consumer interface currently considers the terminal capability of a device receiving multimedia content, however, it makes no effort to adjust the content according to attributes such as screen size, or processing power. In order for the interface to provide an efficient service to users with devices that are wide ranging in capability, it is important that media adaptation technology be integrated. For appropriate multimedia presentations to be displayed on different devices, [27] states that the following issues need to be addressed:

- Multimedia information model for adaptation.
- Media authoring and processing techniques for adaptation.
- Mechanisms for reliably detecting device capabilities and network bandwidth.
- Standard approaches to describe and exchange context information.
- An adaptation agent that analyzes the context information, calculates the adaptation cost, and selects different adaptation strategy.
- A framework to integrate all of the above technologies.

To fully support NGN services, the end user interface must implement a generic, service independent media adaptation unit taking into account the above factors.

8.3.2 Federation

Since one of the main goals of this research is to provide a standardized API for OSA/Parlay applications, it can be assumed that should this goal be achieved, a multitude of service providers will eventually utilize the end user interface for service creation and provision. An important issue concerning the interface then is how two or more service providers utilizing the end user interface would federate to provide services to end users. Service providers have to be confident that only appropriate external users are provided access to service content, and that the access granted is consistent with the rights and privileges specified for that user in a profile. In terms of the access and authentication, and service usage scenarios, questions that may arise are:

- Which other service providers and users can be trusted?
- What types of credentials and requests can be accepted?

In terms of subscription and profile management, questions that may arise are:

- What user and/or service information can be shared?
- How can the overlapping rights and privileges of a user across service provider policy boundaries be managed?

REFERENCES

[1]The Parlay Group, Homepage at http://www.parlay.org, Last Accessed 28 July 2006.

[2] 3GPP TS 22.127 v5.0.0, *"Service Requirement for the Open Service Access (Release 5)"*, June 2001, http://webapp.etsi.org/exchangefolder/ts_122127v050300p.pdf, Last accessed July 28, 2006.

[3]UMTS Forum, *"3G Portal Study -- A Reference Handbook for Portal Operators, Developers and the Mobile Industry"*, Report No. 16, November 2001.

[4]TINA-C , *"Service Architecture"*, Deliverable version 5.0, June 1997.

[5]P. Nana, *"On a hybrid TINA-Parlay service architectecture for next generation networks"*, MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2003.

[6]A. Liotta, A. Yew, C. Bohoris, and G. Pavlou, *"Delivering service adaptation with 3G technology"*, Center for Communication Systems Research, Univ. of Surrey, http://www.ee.surrey.ac.uk/CCSR/Networks/

[7]O. Risnes, *"Developing Advanced Parlay-Enabled Value Added Services"*, OSA/Parlay and Convergent Services Delivery Platforms Forum, December 2003, http://www.telenor.com/rd/pub/not03/N_76_2003.pdf, Last Accessed 28 July 2006.

[8]*"Next Generation Network Intelligence"*, http://www.mobilein.com/NGN-1.htm. Last accessed 07 December 2005.

[9]R. K. Prasad, *"QoS provisioning to a SOHO telecommunications client using a DPE-enabled gateway"*, MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2004.

[10]TINA-C, *"Service Component Specification"*, Deliverable 1.0 b, January 1998, http://www.tinac.com, Last accessed 07 December 2005.

[11]ETSI ES 202 915-3, *"Open Service Access (OSA); Application Programming Interface (API); Part 3: Framework (Parlay 4)"*, ETSI Standard V1.2.1, August 2003.

[12]C. Egelhaaf, K. Eckert, P. Loosemore, N. Quinn, and G. Gylterud, *"Technology Assessment of Middleware for Telecommunications"*, Eurescom Project P910, April 2001.

[13]F. Steegmans (ed.), C. Abarca, J. Forslow, T. Hamada, S. Hogg, H. J. Beom, D. S. Kim, H. Y. Lee, and N. Natarajan. *"TINA Network Resource Architecture,Version 3.0"*, TINA-C, Febr. 1997; public. File: /u/tinac/resources/viewable/nra_v3.0.ps. Last accessed 07 December 2005.

[14]TINA-IN Work Group RFP, *"IN access to TINA services & connection management (IN-TINA Adaptation Unit)"*, Response of Alcatel, Deutsche Telekom, France Télécom and Lucent Technologies, 1999

[15] J. C. Crimi, *"NGN Services (A White Paper)"*, Telcordia Technologies, , http://www.mobilein.com/NGN_Svcs_WP.pdf, Last accessed 07 December 2005.

[16] P. Falcarin, C.A. Licciardi, *"Technologies and Guidelines for Service Creation in NGN"*, December 2003 http://phoenix.labri.fr/documentation/sip/Documentation/Papers/Programming_SIP/Paper_Publication_and_Draft/VOL03.pdf, Last Accessed 28 July 2006.

[17]Y-C Shou, *"Control and management of a customer premises network with a DPE-enabled residential gateway"*, MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2004.

[18] H. E. Hanrahan, *"A Comparative Study of Telecommunications Architectures: Methodology and case studies"*, South African Telecommunication Networks and Applications Conferernce (SATNAC), George, South Africa, September 2003

[19]J. Bakker, J.R. McGoogan, W.F. Opdyke, and F. Panken, "*Rapid Development and Delivery of Converged Services Using APIs*", Bell Labs Technical Journal, July-September 2000

[20] I. Venieris, F. Zizza, T. Magedanz, (Eds.), *"Object Oriented Software Technologies in Telecommunications: From Theory to Practice"*, John Wiley & Sons, 2000.

[21] R. Christian, *"Providing User Context Support for Next Generation Network Services"*, MSc(Eng) Project Report, University of the Witwatersrand, Johannesburg, 2005.

[22]TINA-C, "*Overall Concepts and Principles of TINA*," Deliverable version 1.0, February 1995, http://www.tinac.com, Last accessed 07 December 2005.

[23]TINA-C, *"Ret Reference Point Specification"*, TINA-C Document Version: 1.0, 1998, http://www.tinac.com , Last accessed 07 December 2005.

[24]TINA-C, *"Business Model and Reference Points"*, TINA-C Document Version 4.0, 1997, http://www.tinac.com, Last accessed 07 December 2005.

[25] D. M. Sow, G. Banavar, J. S. Davis II, J. Sussman, and M. R. Rwebangira, *"Preparing the Edge of the Network for Pervasive Content Delivery"*, Workshop on Middleware for Mobile Computing, November 16, 2001.

[26]N. Chan Wah and T. Pek Yew, *"QoS and Delivery Context in Rule-Based Edge Services"*, 7th International Workshop on Web Content Caching and Distribution http://2002.iwcw.org/papers/18500015.pdf, Last accessed 07 December 2005.

[27] Z. Lei, and N. D. Georganas, *"Context-based media adaptation for pervasive computing"*, Canadian Conference on Electrical and Computer Engineering (CCECE 2001), Toronto, May 2001.

[28]N. B. Sheridan-Smith, J. Soliman, D. Colquitt, J. R. Leaney, T. O'Neill, and M. Hunter, *"Improving the user experience through adaptive and dynamic service management"*

[29]Rococo Software, "*An Introduction to OSA/Parlay, a white paper from Rococo Software"*, http://rococosoft.com/docs/osa_parlay_wp.pdf , Last accessed 07 December 2005.

[30]M. Walkden, N. Edwards, D. Foster, M. Jankovic, B. Odadzic, G. Nygreen, G. Gyterud, C. Moiso, S.M. Tognon, and B. de Bruijn, *"Open Service Access – Advantages and opportunities in service provisioning"*, Eurescom project P1110, January 2002.

[31]G. Berhe, L. Brunie, and J. Pierson, *"Modeling Service-Based Multimedia Content Adaptation in Pervasive Computing"*, Conference on Computing Frontiers, Proceedings of the 1st conference on Computing frontiers, April 14-16, 2004.

[32]A. Held, S. Buchholz, and A. Schill, *"Modeling of Context Information for Pervasive Computing Applications"*, Proc. Of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SC12002), Orlando, FL, USA, Jul 14-18, 2002.

[33]M. E. Anagnostou, M.A. Lambrou, E. D. Sykas, *"Context aware service engineering in support of future business networks",* ICCS, Institute of Communication and Computer Systems, Computer Networks Lab, Athens, Greece, http://context.upc.es/Papers/ContextCOCONET.pdf, 2003, Last accessed 07 December 2005.

[34]ETSI ES 202 915-7, *"Open Service Access (OSA); Application Programming Interface (API); Part 7: Terminal Capabilities SCF (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[35]ETSI ES 202 925-6, *"Open Service Access (OSA); Application Programming Interface (API); Part 6: Mobility SCF (Parlay 4)"*, ETSI Standard V1.2.1, August 2003.

[36]ETSI ES 202 915-14, *"Open Service Access (OSA); Application Programming Interface (API); Part 14: Presence and Availability Management SCF"*, ETSI Standard V1.1.1, January 2003.

[37]D. Adamopoulos, *"A Structured Approach to the Development of Telematic Services Using Distributed Object-Oriented Platforms"*, Phd Thesis, University of Surrey, UK, November 2000.

[38]H. E. Hanrahan, and D. Mwansa, *"A Vision for the Target Next Generation Network"*, School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg, September 2003.

[39]ETSI ES 202 925-4-2, *"Open Service Access (OSA); Application Programming Interface (API); Part 4:Sub-part 2: Generic Call Control SCF (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[40]ETSI ES 202 925-4-3, *"Open Service Access (OSA); Application Programming Interface (API); Part 4:Sub-part 3: Multiparty Call Control SCF (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[41]ETSI ES 202 925-4-4, *"Open Service Access (OSA); Application Programming Interface (API); Part 4:Sub-part 4: Multimedia Multiparty Call Control SCF (Parlay 4)"*, ETSI Standard V1.2.1, August 2003.

[42]ETSI ES 202 925-4-5, *"Open Service Access (OSA); Application Programming Interface (API); Part 4:Sub-part 5: Conference Call Control SCF (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[43]ETSI ES 202 925-8, *"Open Service Access (OSA); Application Programming Interface (API); Part 8: Data Session Control SCF (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[44]ETSI ES 202 925-9, *"Open Service Access (OSA); Application Programming Interface (API); Part 9: Generic Messaging SCF (Parlay 4)"*, ETSI Standard V1.2.1, August 2003.

[45]M. Henning, and S. Vinoski, *"Advanced CORBA Programming with C++"*, Addison-Wesley Professional, 1st edition, February 1999.

[46]G. Brose, A. Vogel, and K. Duddy, *"JAVA Programming with CORBA, Advanced Techniques for Building Distributed Applications"*, John Wiley & Sons, 3rd Edition, June 2001.

[47]P. Moodley, "*Parlay Gateway Emulator for Performance Studies"*, University of the Wiwatersrand, School of electrical and information engineering, 2004.

[48]Object Computing, Inc. *"TAO Developer's Guide"*, Version 1.3a, 2003. See also http://www.cs.wustl.edu/~schmidt/TAO.html, Last Accessed 28 July 2006.

[49]Centre for Telecommunications Access and Services, "*South African Telecommunications Information Network Architecture (SATINA).*" Last accessed 07 December 2005, http://www.satina.ee.wits.ac.za.

[50]S. Beddus, S. Davis, and G. Bruce, "*Opening up networks with JAIN Parlay"*, IEEE Communications Magazine, vol. 38, no. 4, pp. 136 – 143, April 2000.

[51]ETSI ES 202 925-1, *"Open Service Access (OSA); Application Programming Interface (API); Part 1:Overview (Parlay 4)",* ETSI Standard V1.2.1, August 2003.

[52] A. J. Moerdijk, and L. Klosterman, *"Opening Networks with OSA/Parlay APIs: standards and aspects behind the APIs"*, Ericsson, March 2002.

[53] P. Kuuppelomaki, and A. Mustonen, *"Open architectures in telecommunications convergence"*, Satakunta Polytechnic, http://trc.pori.tut.fi/tots/Open%20architectures%20in%20telecommunication%20and%20IP%20convergence.pdf , Last Accessed 07 December 2005.

[54] The JAIN Community, *"The JAIN APIs: Integrated Network APIs for the Java Platform*", January 2002, http://java.sun.com/products/jain/WP2002.pdf , Last Accessed July 28, 2006.

[55] ITU-T Recommendation F.850, *"Principles of Universal Personal Telecommunication (UPT)"*, March 1993, http://www.itu.int/rec/T-REC-F.850-199303-I/en, Last Accessed 28 July 2006.

APPENDIX A

SUMMARY OF INTERFACES, METHODS, AND ATTRIBUTES

A.1 Access and Authentication

Interfaces	Supported Methods	Attributes
i_Initial	contactProvider()	providerName
	requestAccess()	userId userProperties asId
	setupAccessSession()	userInfo asId sIOR
i_Authenticate	getAuthenticationMethods	desiredProperties authMethods
	authenticate	authMethod authenData privAttribReq privAttrib authStatus
	continueAuthentication	authenData privAttrib continuationData authStatus
i_Access	setUserCtxt()	userCtxt
	listServices()	desiredProperties howMany serviceList
	startServices()	(see Service Usage)

	endAccessSession()	asId

Table 1: Access and Authentication Management Interfaces, Operations, and Attributes

A.2 Subscription and Profile Management

Interfaces	Supported Methods	Attributes
I_Subscribe	listServices()	desiredProperties howMany serviceList
	subscribe()	serviceList subscriberInfo subscriberId interfaceList
	Unsubscribe()**	subscriberId subscriberInfo serviceIdList unsubscribedServices
I_SubscriberMgmt	createSAEs	entityList subscriberId entityIdList
	deleteSAEs	subscriberId entityIdList
	createSAGs	subscriberId sagList sagIdList
	deleteSAGs	subscriberId sagIdList
	assignSAEs	subscriberId entityIdList sagId
	removeSAEs	Same as assignSAEs

	listSAEs	subscriberId entityIdList sagId
	listSAGs	subscriberId sagIdList
	getSubscriberInfo	subscriberId
	setSubscriberInfo	subscriberId
	listSubscribedServices	subscriberId serviceList
i_ServiceContractInfoMgmt	getServiceTemplate	serviceId template
	setServiceTemplate	serviceId template spId
	getServiceProfiles	userId serviceIdList serviceProfileList
	assignServiceProfile	spId sagIdList saeIdList
	removeServiceProfile	Not Implemented
	activateServiceProfile	spIdList
	deactivateServiceProfile	Not Implemented
	deleteServiceProfile	Not Implemented
	getServiceContractInfo	Not Implemented
	defineServiceContract	serviceContract spIdList

Table 2: Summary of Subscription and Profile Management Interfaces, Operations, and Attributes

A.3 Service Usage Management

A.3.1 Single Party Services

Interfaces	Supported Methods	Attributes
i_Access	startService**	serviceId app uaProperties sessionInfo
i_SSManage	createSSession	serviceId userId app uaProperties sessionInfo (contains ssId)
	createIpAppInterface	interfaceList ssId
	endSSession	ssId
	releaseIpAppResources	ssId
i_BasicReq	endSessionReq	ssId userId
IpAppLogic	initiateApp	sessionInfo interfaceList

Table 3: Summary of Single Party Service Usage Management Interface, Operations, and Attributes

A.3.2 Multi Party Services

Interfaces	Supported Methods	Attributes
i_PartyMultipartyReq	inviteUserReq**	userId invitedUserDetails invitationId invitationReply
	joinSessionReq	userId serviceId sessionId uaProperties sessionInfo
i_PartyMultipartyInd	endSessionInd**	sessionId indId
	inviteUserInd	Not Implemented
	joinSessionInd**	sessionId indId userDetails isValid
I_PartyMultipartyExe	endSessionExe**	userId sessionId
I_PartyMultipartyInfo	inviteUserInfo	Not Implemented
	joinSessionInfo	Not Implemented
	endSessionInfo	Not Implemented
	inviteReplyInfo	Not Implemented
i_Invitation	Invite	Invitation reply
	listSessionInvitations**	userId invitationList

	joinSessionWithInvitation**	invitationId app sessionInfo
	cancel	userId invitationId

Table 4: Summary of Multiparty Service Usage Management Interfaces, Operations, and Attributes

APPENDIX B
UPDATING OF CONTEXT

B.1 The User Location Case

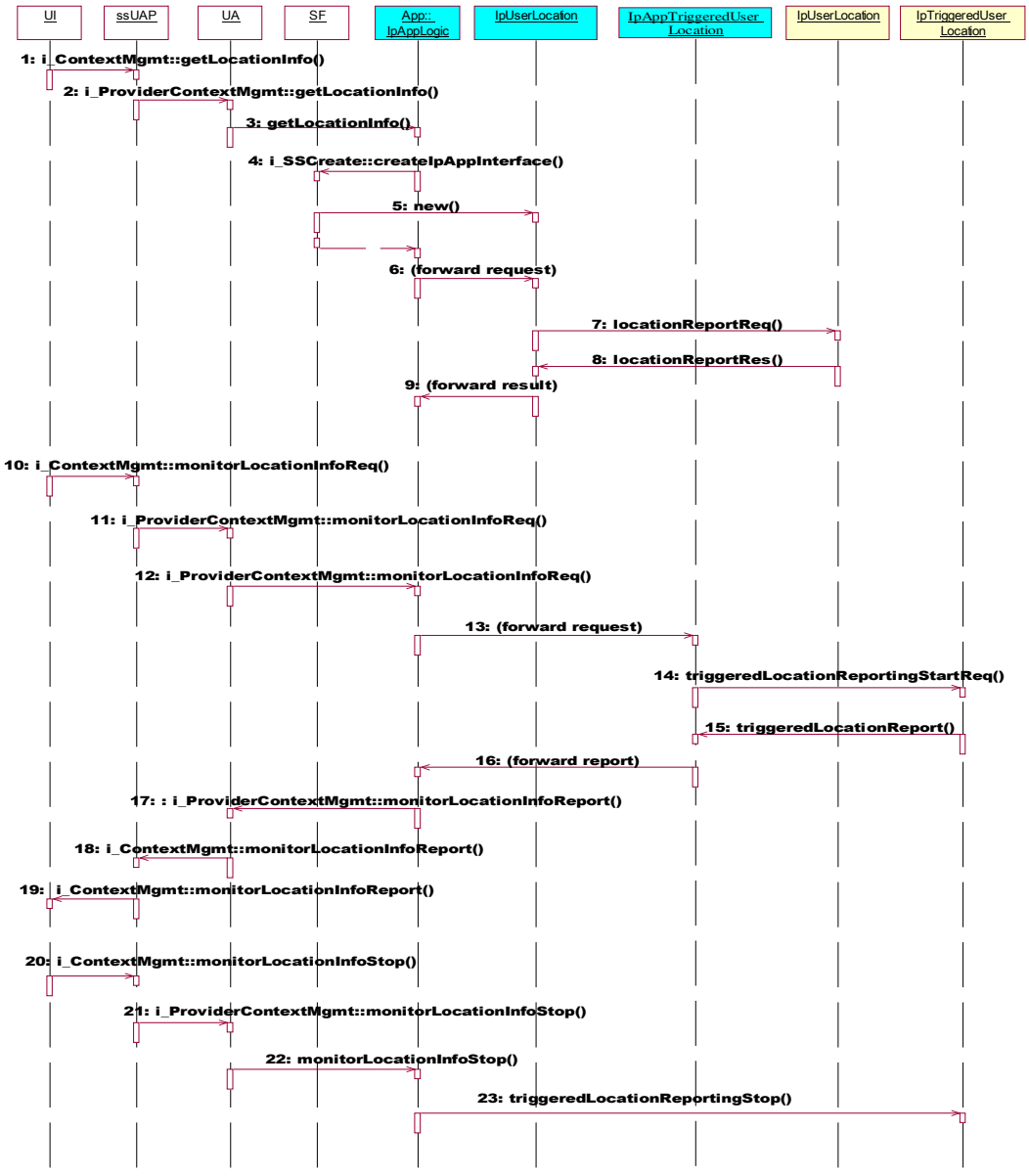

Figure B.1 Update User Context (User Location Update) Sequence Diagram

1. The user requests user location information.

2. The ssUAP forwards the request to the UA.

3. The UA requests the Parlay Application (App) to user location using the Mobility SCF.

4. The Application requests the SF to create an IpAppUserLocation callback interface.

5. The SF creates an IpAppUserLocation callback object.

6. The App requests the IpAppUserLocation object to retrieve the user location information.

7. The IpAppUserLocation object requests the user location.

8. The IpUserLocation object provides a user location report.

9. The report is forwarded to the IpAppLogic.

The report is then forwarded to the user. This is not shown here.

10. The user requests user location reports when location changes.

11. The request is forwarded to the UA.

12. The UA forwards the request to the Parlay Application.

(At this point we assume that the SF has already created the IpAppTriggeredUserLocation callback object)

13. The App requests the IpAppTriggeredUserLocation object to setup user location information monitoring.

14. The user location changes are started to be monitored.

15. The user location information has changed and is reported as requested.

16. The report is forwarded to the Application.

17. The application forwards the report to the UA. The UA may store the user location information at this point.

18. The UA forwards the report to the ssUAP.

19. The ssUAP forwards the report to the user.

20. The user requests the user location monitoring to stop.

18. The request is forwarded to the UA.

19. The UA forwards the request to the Application.

20. The Application stops the user location monitoring.

B.2 Updating User Context during Access Session Setup

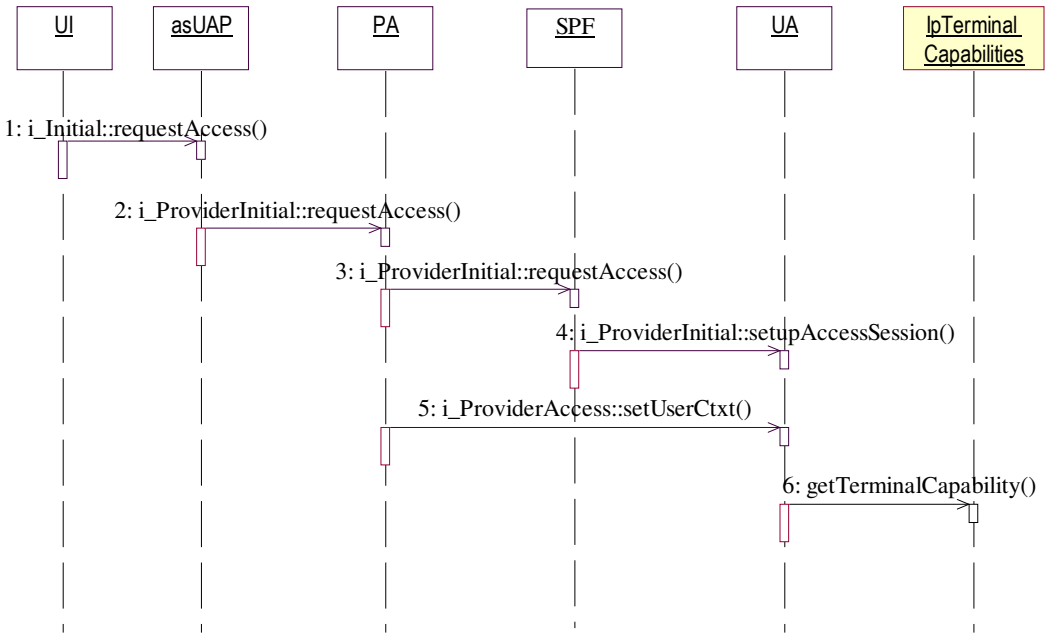

Figure B.2 User context update during access session setup

1. The user uses the User Interface to request to log in to the provider.

2. The asUAP invokes the **requestAccess()** method on the PA.

3. The PA invokes **requestAccess()** on the provider specific SPF to establish an access session that allows the user access to the provider's services.

We assume the user has already been authenticated.

4. The SPF contacts the UA and requests the setup of an access session with the authenticated user.

5. The PA completes access session setup by invoking the **setUserCtxt()** operation on the ***i_ProviderAccess*** interface. This gives the user's UA some information about the user's domain, such as interface references, and terminal capability information.

The user's UA acknowledges the receipt of this user's domain information. Once the PA receives the UAs confirmation, the access session can be officially considered to be established.

6. The UA requests terminal capability information from the Terminal Capability SCF.

(We assume that the UA is authorized to communicate with the SCF.)

The PA informs the asUAP of the successful establishment of the access session. The asUAP in turn will inform the user.

In the above scenario, if we assume that the UA is authorized to request information from the SCF, then the UA may also request terminal capability and user location reports as shown in the update user context management scenarios.

B.3 IDL Specification for i_UserContextManagement Interface

```
interface i_UserContextManagement{

void getTerminalCapabilityInfo(
                in TINASubCommonTypes::t_EntityId entityId,
                out TINAAccessCommonTypes::t_TerminalConfig terminalCap
        );
void monitorTeminalCapabilityReq(
                in TINASubCommonTypes::t_EntityId entityId,
        );
void monitorTerminalCapabilityReport(
                in TINAAccessCommonTypes::t_TerminalConfig terminalCap
        );
void monitorTerminalCapabilityStop(
                in TINASubCommonTypes::t_entityId entityId,
        );

void getLocationInfo(
                in TINASubCommonTypes::t_EntityIdList entityIdList,
                in TpUserLocationSet locations
        );
void monitorLocationInfoReq(
```

```
            in TINASubCommonTypes::t_EntityIdList entityIdList,
        );
void monitorLocationInfoReport(
            in TpUserLocationSet locations
        )
void monitorLocationStop(
            in TINASubCommonTypes::t_EntityIdList entityIdList,


        )


};
```

TpUserLocationSet *locations* is defined in the Parlay Mobility SCF common data definitions and contains fields as shown by the following table:

Field	**Type**	**Description**
UserID	TpAddress	The address of the user
StatusCode	TpMobilityError	Indication of error
GeographicalPosition	TpGeographicalPosition	Specification of a position and an area of uncertainty.

Table 5 Fields contained within the locations data structure. Redrawn from [35]

APPENDIX C
IDL SPECIFICATION FOR CONSUMER INTERFACE

The following section presents a complete listing of all the interfaces, methods, and attributes implemented for each of the service components using the Interface Definition Language (IDL). The TINA data definition IDLs are found in [4],[10], and [23]. The Terminal Capability and Mobility data definition IDLs are found in the corresponding OSA/Parlay SCF standards [34],[35].

C.1 asUAP

```
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"

module asUAP{

interface i_Initial {

void startAccess(
in TINAUserInitial::t_ProviderId providerId
);
};

        interface i_Access{

void requestLogin (
inout TINACommonTypes::t_UserId userId,
in TINACommonTypes::t_UserProperties userProperties,
out TINAAccessCommonTypes::t_AccessSessionId asId
);

void discoverServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);
```

```
void endAccessSession(
in TINAAccessCommonTypes::t_AccessSessionId asId
);
};
};
```

C.2 PA

```
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderInitial.idl"
#include "TINAAuthenticationTypes.idl"
#include "TINAProviderAccess.idl"

module PA{



interface i_Initial {

 void contactProvider (
 in TINAUserInitial::t_ProviderId providerId
 );

 void requestAccess (
 inout TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_UserProperties userProperties,
 out Object PAAccessIR,
 out Object PAAuthenticateIR,
 out TINAAccessCommonTypes::t_AccessSessionId asId,
 out string authenticateReply
 );
 };

interface i_Authenticate{

 void getAuthenticationMethods (
                        in TINAAuthenticationTypes::t_AuthMethodSearchProperties
                        desiredProperties,
                        out TINAAuthenticationTypes::t_AuthMethodDescList
                        authMethods
);

 void authenticate(
 in TINAAuthenticationTypes::t_AuthMethod authMethod,
 in TINAAuthenticationTypes::t_securityName securityName,
 inout TINAAuthenticationTypes::t_opaque authStruct,
 out TINAAuthenticationTypes::t_AuthenticationStatus authStatus
```

```
);

void continueAuthentication(
inout TINAAuthenticationTypes::t_opaque authStruct,
out TINAAuthenticationTypes::t_AuthenticationStatus authStatus
);
};


interface i_Access{

void discoverServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);


void startService (
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINAProviderAccess::t_ApplicationInfo app,
in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
in TINAProviderAccess::t_StartServiceSSProperties ssProperties,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

void endAccessSession(
in TINAAccessCommonTypes::t_AccessSessionId asId
);

};
};
```

C.3 SPF

```
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderInitial.idl"
#include "TINAAuthenticationTypes.idl"
#include "TINAProviderAccess.idl"



module SPF{



interface i_ProviderInitial{

void requestAccess (
in string userId,
in TINACommonTypes::t_UserProperties userProperties,
out TINAAccessCommonTypes::t_AccessSessionId asId,
out Object SPFAuthenticateIR_obj,
out Object UAAccessIR_obj,
out string authenticateReply
);
};



interface i_ProviderAuthenticate{

void getAuthenticationMethods (
                        in TINAAuthenticationTypes::t_AuthMethodSearchProperties
                        desiredProperties,
                        out TINAAuthenticationTypes::t_AuthMethodDescList
                        authMethods
);

void authenticate(
in TINAAuthenticationTypes::t_AuthMethod authMethod,
in TINAAuthenticationTypes::t_securityName securityName,
inout TINAAuthenticationTypes::t_opaque authStruct,
```

```
out TINAAuthenticationTypes::t_AuthenticationStatus authStatus
);

void continueAuthentication(
inout TINAAuthenticationTypes::t_opaque authStruct,
out TINAAuthenticationTypes::t_AuthenticationStatus authStatus
);


};



interface i_ProviderAccess{

void joinSessionWithInvitation (
in TINAAccessCommonTypes::t_InvitationId invitationId,
in TINAProviderAccess::t_ApplicationInfo app,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

};

};
```

C.4 UA

```
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderInitial.idl"
#include "TINAProviderAccess.idl"
#include "TINASubCommonTypes.idl"

module UA{

interface i_ProviderInitial {


void setupAccessSession (
in TINAAccessCommonTypes::t_UserInfo userinfo,
out TINAAccessCommonTypes::t_AccessSessionId asId,
out string sIOR
);
};

interface i_ProviderAccess{



void setUserCtxt (
in TINAProviderAccess::t_UserCtxt userCtxt
);


void discoverServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);


void startService (
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINAProviderAccess::t_ApplicationInfo app,
in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
```

```
in TINAProviderAccess::t_StartServiceSSProperties ssProperties,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

void endAccessSession(
in TINAAccessCommonTypes::t_AccessSessionId asId
);
};

interface i_PartyMultipartyReq{

void joinSessionReq(
in TINACommonTypes::t_UserId userId,
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINACommonTypes::t_SessionId sessionId,
in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

void inviteUserReq (
in TINACommonTypes::t_UserId userId,
in TINACommonTypes::t_UserDetails userDetails,
out unsigned long invitationId,
out TINACommonTypes::t_InvitationReply invitationReply
);

};



interface i_Invitation {

                void invite (
in TINAAccessCommonTypes::t_SessionInvitation invitation,
out TINACommonTypes::t_InvitationReply reply
);


void cancel (
in TINACommonTypes::t_UserId userId,
in TINAAccessCommonTypes::t_InvitationId invitationId
);

void joinSessionWithInvitation (
in TINAAccessCommonTypes::t_InvitationId invitationId,
in TINAProviderAccess::t_ApplicationInfo app,
```

```
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

void listSessionInvitations (
in TINACommonTypes::t_UserId userId,
in TINAAccessCommonTypes::t_InvitationList invitationList
);

};


interface i_PartyMultipartyInfo{

oneway void inviteUserInfo (
in TINACommonTypes::t_SessionId sessionId,
in TINACommonTypes::t_UserDetails userDetails,
in unsigned long invitationId
);

oneway void endSessionInfo (
in TINACommonTypes::t_SessionId sessionId
);

oneway void joinSessionInfo (
in TINACommonTypes::t_SessionId sessionId,
in TINACommonTypes::t_UserDetails userDetails
);

oneway void inviteReplyInfo (
in TINACommonTypes::t_SessionId sessionId,
in unsigned long invitationId,
in TINACommonTypes::t_InvitationReply reply
);
};

interface i_PartyMultiPartyInd{

void joinSessionInd (
in TINACommonTypes::t_SessionId sessionId,
in unsigned long indId,
in TINACommonTypes::t_UserDetails userDetails,
in boolean isValid
);


void endSessionInd (
```

```
in TINACommonTypes::t_SessionId sessionId,
in unsigned long indId,
);
};


interface i_BasicReq{

void endSessionReq (
in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_SessionId sessionId
);
};

interface i_PartyMultiPartyExe{

void endSessionExe (
in TINACommonTypes:: t_UserId userId,
in TINACommonTypes::t_SessionId sessionId,
                        out any accountInfo
);

};

interface i_SessionInfo{


void sessionEnded (
in TINACommonTypes::t_GlobalSessionId globalSessionId,
in TINACommonTypes::t_PartyId partyId,
in any AccountingInfo
);
};

interface i_ProviderSubscribe{


void listServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);

void subscribe(
                        in TINASubCommonTypes::t_Subscriber subscriberInfo,
```

```
in TINASubCommonTypes::t_ServiceIdList serviceList,
out TINASubCommonTypes::t_AccountNumber subscriberId,
out TINACommonTypes::t_InterfaceList interfaceList
);

void unsubscribe(
in TINASubCommonTypes::t_AccountNumber subscriberId,
                        in TINASubCommonTypes::t_Subscriber subscriberInfo,
in TINASubCommonTypes::t_ServiceIdList serviceIdList,
                        out TINASubCommonTypes::t_ServiceIdList
                        unsubscribedServices
);
};

interface i_ProviderSubscriberMgmt{

void listServices (
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);

void listSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
          out TINASubCommonTypes::t_SagIdList sagIdList
);

void listSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_SagId sagId,
out TINASubCommonTypes::t_entityIdList entityList
);

void listSubscribedServices(
in TINASubCommonTypes::t_AccountNumber subscriberId,
out TINAAccessCommonTypes::t_ServiceList service_list
);

void createSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_EntityList entityList,
out TINASubCommonTypes::t_entityIdList entityIdList
);
```

```
void createSAGs(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_SagList sagList,
 out TINASubCommonTypes::t_SagIdList sagIdList
 );

 void assignSAEs(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_entityIdList entityList,
 in TINASubCommonTypes::t_SagId sagId
 );

 void removeSAEs(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_entityIdList entityList,
 in TINASubCommonTypes::t_SagId sagId
 );

 void deleteSAEs(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_entityIdList entityList
 );

 void deleteSAGs(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_SagIdList sagIdList
 );

 void setSubscriberInfo(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 in TINASubCommonTypes::t_Subscriber subscriberInfo
 );

 void getSubscriberInfo(
 in TINASubCommonTypes::t_AccountNumber subscriberId,
 out TINASubCommonTypes::t_Subscriber subscriberInfo
 );
 };

interface i_ProviderServiceContractMgmt{

 void assignServiceProfile(
 in TINASubCommonTypes::t_ServiceProfileId spId,
                          in TINASubCommonTypes::t_SagIdList sagIdList,
                          in TINASubCommonTypes::t_entityIdList saeIdList
 );
```

```
void activateServiceProfiles(
in TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void defineServiceContract(
in TINASubCommonTypes::t_ServiceContract serviceContract,
out TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void checkServiceProfile(
in TINACommonTypes::t_UserId userId,
                        in TINASubCommonTypes::t_ServiceProfile serviceProfile,
                        out boolean accepted
);

void getServiceProfiles(
in TINACommonTypes::t_UserId userId,
                        in TINASubCommonTypes::t_ServiceIdList serviceList,
                        out TINASubCommonTypes::t_ServiceProfileList
                        serviceProfileList
);

  void getServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
out TINASubCommonTypes::t_ServiceTemplate template
);

void setServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINASubCommonTypes::t_ServiceTemplate template,
out TINASubCommonTypes::t_ServiceProfileId spId
);

};


};
```

C.5 ssUAP

```
#include "TINASubCommonTypes.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"


module ssUAP{

interface i_Access {

 void startService (
 in TINAAccessCommonTypes::t_ServiceId serviceId,
 in TINAProviderAccess::t_ApplicationInfo app,
                              in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo
 );

 };


interface i_PartyMultipartyReq {

 void joinSessionReq(
 in TINACommonTypes::t_UserId userId,
 in TINAAccessCommonTypes::t_ServiceId serviceId,
 in TINACommonTypes::t_SessionId sessionId,
 in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo
 );

 void inviteUserReq (
 in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_UserDetails invitedUserDetails,
 out unsigned long invitationId,
 out TINACommonTypes::t_InvitationReply reply
 );

 };
```

```
interface i_Invitation{


 void joinSessionWithInvitation (
 in TINAAccessCommonTypes::t_InvitationId invitationId,
 in TINAProviderAccess::t_ApplicationInfo app,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo
 );

 void cancel (
 in TINACommonTypes::t_UserId userId,
 in TINAAccessCommonTypes::t_InvitationId inviteId
 );

 void listSessionInvitations (
 in TINACommonTypes::t_UserId userId,
 in TINAAccessCommonTypes::t_InvitationList invitationList
 );
 };


interface i_BasicReq{

 void endSessionReq (
 in TINACommonTypes::t_UserId userId,
  in TINACommonTypes::t_SessionId sessionId
 );
 };

interface i_PartyMultiPartyInd{

 void endSessionInd (
 in TINACommonTypes::t_SessionId sessionId,
 in unsigned long indId,
 );
 };

interface i_PartyMultiPartyExe{

 void endSessionExe (
 in TINACommonTypes:: t_UserId userId,
 in TINACommonTypes::t_SessionId sessionId,
                              out any accountInfo
 );

 };
```

```
interface i_Subscribe{


void discoverServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);

void subscribe(
in TINASubCommonTypes::t_Subscriber subscriberInfo,
in TINASubCommonTypes::t_ServiceIdList serviceIdList,
out TINASubCommonTypes::t_AccountNumber subscriberId,
out TINACommonTypes::t_InterfaceList interfaceList
);

void unsubscribe(
in TINASubCommonTypes::t_Subscriber subscriberInfo,
in TINASubCommonTypes::t_ServiceIdList serviceIdList,
                        out TINASubCommonTypes::t_ServiceIdList
                        unsubscribedServices
);
};


interface i_SubscriberMgmt{

void listServices (
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);

void listSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
          out TINASubCommonTypes::t_SagIdList sagIdList
);

void listSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_SagId sagId,
out TINASubCommonTypes::t_entityIdList entityIdList
);
```

```
void listSubscribedServices(
in TINASubCommonTypes::t_AccountNumber subscriberId,
out TINAAccessCommonTypes::t_ServiceList service_list
);

void createSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_EntityList entityList,
out TINASubCommonTypes::t_entityIdList entityIdList
);

void createSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_SagList sagList,
out TINASubCommonTypes::t_SagIdList sagIdList
);

void assignSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_entityIdList entityIdList,
in TINASubCommonTypes::t_SagId sagId
);

void removeSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId
in TINASubCommonTypes::t_entityIdList entityIdList,
in TINASubCommonTypes::t_SagId sagId
);

void deleteSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_entityIdList entityIdList
);

void deleteSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId
in TINASubCommonTypes::t_SagIdList sagIdList
);


void setSubscriberInfo(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_Subscriber subscriberInfo
);
```

```
void getSubscriberInfo(
in TINASubCommonTypes::t_AccountNumber subscriberId,
out TINASubCommonTypes::t_Subscriber subscriberInfo
);

};

interface i_ServiceContractMgmt{

                void assignServiceProfile(
                        in TINASubCommonTypes::t_ServiceProfileId spId,
              in TINASubCommonTypes::t_SagIdList sagIdList,
            in TINASubCommonTypes::t_entityIdList saeIdList
);

void activateServiceProfiles(
in TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void defineServiceContract(
in TINASubCommonTypes::t_ServiceContract serviceContract,
out TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void checkServiceProfile(
in TINACommonTypes::t_UserId userId,
                in TINASubCommonTypes::t_ServiceProfile serviceProfile,
                out boolean accepted
);
void getServiceProfiles(
in TINACommonTypes::t_UserId userId,
                in TINASubCommonTypes::t_ServiceIdList serviceIdList,
                        out TINASubCommonTypes::t_ServiceProfileList
                        serviceProfileList
);

  void getServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
out TINASubCommonTypes::t_ServiceTemplate template
);

void setServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINASubCommonTypes::t_ServiceTemplate template,
out TINASubCommonTypes::t_ServiceProfileId spId
);
```

```
};
};
```

C.6 SF

```
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"

module SF{


interface i_SSManage {

 void endSSession (
 in TINACommonTypes::t_SessionId sessionId
 );

 void createSSession (
 in TINAAccessCommonTypes::t_ServiceId serviceId,
 in TINACommonTypes::t_UserId userId,
 in TINAProviderAccess::t_ApplicationInfo app,
 in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo,
 );

 void createIpAppInterface(
 in TINACommonTypes::t_InterfaceList ipAppInterfaceList,
 out TINACommonTypes::t_InterfaceList parlay_Interfaces,
 out string reply
 );

 };
 };
```

C.7 SUB

```
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderInitial.idl"
#include "TINAProviderAccess.idl"
#include "TINASubCommonTypes.idl"


module SUB{

interface i_Subscribe{

void listServices (
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);

void subscribe(
in TINASubCommonTypes::t_Subscriber subscriberInfo,
in TINASubCommonTypes::t_ServiceIdList serviceList,
out TINASubCommonTypes::t_AccountNumber subscriberId,
out TINACommonTypes::t_InterfaceList interfaceList
);

void unsubscribe(
in TINASubCommonTypes::t_Subscriber subscriberInfo,
in TINASubCommonTypes::t_ServiceIdList serviceIdList,
                        out TINASubCommonTypes::t_ServiceIdList
                        unsubscribedServices
);

};

interface i_SubscriberMgmt{

void listServices (
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
```

```
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);


void createSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_EntityList entityList,
out TINASubCommonTypes::t_entityIdList entityIdList
);

void createSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_SagList sagList,
out TINASubCommonTypes::t_SagIdList sagIdList
);

void assignSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_entityIdList entityIdList,
in TINASubCommonTypes::t_SagId sagId
);

void removeSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_entityIdList entityIdList,
in TINASubCommonTypes::t_SagId sagId
);

void deleteSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_entityIdList entityIdList
);

void deleteSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_SagIdList sagIdList
);

void listSAGs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
        out TINASubCommonTypes::t_SagIdList sagIdList
);

void listSAEs(
in TINASubCommonTypes::t_AccountNumber subscriberId,
```

```
in TINASubCommonTypes::t_SagId sagId,
out TINASubCommonTypes::t_entityIdList entityIdList
);

void listSubscribedServices(
in TINASubCommonTypes::t_AccountNumber subscriberId,
out TINAAccessCommonTypes::t_ServiceList service_list
);


void setSubscriberInfo(
in TINASubCommonTypes::t_AccountNumber subscriberId,
in TINASubCommonTypes::t_Subscriber subscriberInfo
);

void getSubscriberInfo(
in TINASubCommonTypes::t_AccountNumber subscriberId,
out TINASubCommonTypes::t_Subscriber subscriberInfo
);

};

interface i_ServiceContractMgmt{

void assignServiceProfile(
                        in TINASubCommonTypes::t_ServiceProfileId spId,
                        in TINASubCommonTypes::t_SagIdList sagIdList,
                        in TINASubCommonTypes::t_entityIdList saeIdList
);

void activateServiceProfiles(
in TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void defineServiceContract(
in TINASubCommonTypes::t_ServiceContract serviceContract,
out TINASubCommonTypes::t_ServiceProfileIdList spIdList
);

void getServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
out TINASubCommonTypes::t_ServiceTemplate template
);

void setServiceTemplate(
in TINAAccessCommonTypes::t_ServiceId serviceId,
```

```
in TINASubCommonTypes::t_ServiceTemplate template,
out TINASubCommonTypes::t_ServiceProfileId spId
);

void checkServiceProfile(
in TINACommonTypes::t_UserId userId,
                    in TINASubCommonTypes::t_ServiceProfile serviceProfile,
                    out boolean accepted
);

void getServiceProfiles(
in TINACommonTypes::t_UserId userId,
                in TINASubCommonTypes::t_ServiceIdList serviceIdList,
                    out TINASubCommonTypes::t_ServiceProfileList
                    serviceProfileList
);


};


};
```

C.7 App

```
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"

module App{

interface IpAppLogic{

 void initiateApp(
 in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_UserProperties userProperties,
 in TINAAccessCommonTypes::t_SessionInfo sessionInfo,
 out TINACommonTypes::t_InterfaceList ipAppInterfaceList
 );

 void joinSessionReq(
 in TINACommonTypes::t_UserId userId,
 in TINAAccessCommonTypes::t_ServiceId serviceId,
 in TINACommonTypes::t_SessionId sessionId,
 in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo
 );

 void inviteUserReq (
 in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_UserDetails invitedUserDetails,
 out unsigned long invitationId,
 out TINACommonTypes::t_InvitationReply reply
 );

 void joinSessionInd (
 in TINACommonTypes::t_SessionId sessionId,
 in unsigned long indId,
 in TINACommonTypes::t_UserDetails userDetails,
 out boolean isValid
 );

 void joinSessionWithInvitation (
 in TINAAccessCommonTypes::t_InvitationId invitationId,
 in TINAProviderAccess::t_ApplicationInfo app,
 out TINAAccessCommonTypes::t_SessionInfo sessionInfo
 );
```

```
void passReferences (
in TINACommonTypes::t_InterfaceList parlay_Interfaces
);

void releaseIpAppResources (
in TINACommonTypes::t_SessionId sessionId
);

void endSessionReq (
in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_SessionId sessionId
);

void endSessionExe (
in TINACommonTypes:: t_UserId userId,
in TINACommonTypes::t_SessionId sessionId,
                              out any accountInfo
);

};

};
```

C.8 UA2

```
#include "TINASubCommonTypes.idl"
#include "TINAUserInitial.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderInitial.idl"
#include "TINAProviderAccess.idl"

module UA2{

interface i_Initial {


void setupAccessSession (
in TINAAccessCommonTypes::t_UserInfo userinfo,
out TINAAccessCommonTypes::t_AccessSessionId asId,
out string sIOR
);
};

interface i_ProviderAccess{



void setUserCtxt (
in TINAProviderCommonTypes::t_UserCtxt userCtxt
);


void discoverServices(
                        in TINAProviderAccess::t_DiscoverServiceProperties
                        desiredProperties,
in unsigned long howMany,
out TINAAccessCommonTypes::t_ServiceList services
);


void startService (
in TINAAccessCommonTypes::t_ServiceId serviceId,
in TINAProviderAccess::t_ApplicationInfo app,
in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo,
```

```
);

void joinSessionWithInvitation (
in TINAAccessCommonTypes::t_InvitationId invitationId,
in TINAProviderAccess::t_ApplicationInfo app,
out TINAAccessCommonTypes::t_SessionInfo sessionInfo
);

};

interface i_PartyMultipartyReq{

void inviteUserReq (
in TINACommonTypes::t_ParticipantSecretId myId,
in TINACommonTypes::t_UserDetails invitedUser,
out unsigned long invitationId,
out TINACommonTypes::t_InvitationReply reply
);
};



interface i_Invitation {

                    void invite (
in TINAAccessCommonTypes::t_SessionInvitation invitation,
out TINACommonTypes::t_InvitationReply reply
);

void cancel (
in TINACommonTypes::t_UserId userId,
in TINAAccessCommonTypes::t_InvitationId inviteId
);
};



interface i_PartyMultipartyInfo{

oneway void inviteUserInfo (
in TINACommonTypes::t_SessionId sessionId,
in TINACommonTypes::t_UserDetails userDetails,
in unsigned long invitationId
);

oneway void endSessionInfo (
```

```
in TINACommonTypes::t_SessionId sessionId
);

oneway void joinSessionInfo (
in TINACommonTypes::t_SessionId sessionId,
in TINACommonTypes::t_UserDetails userDetails
);

oneway void inviteReplyInfo (
in TINACommonTypes::t_SessionId sessionId,
in unsigned long invitationId,
in TINACommonTypes::t_InvitationReply reply
);
};

interface i_PartyMultiPartyInd{

void joinSessionInd (
in TINACommonTypes::t_SessionId sessionId,
in unsigned long indId,
in TINACommonTypes::t_UserDetails userDetails,
out boolean isValid
);

void inviteUserInd(
in unsigned long indId,
in TINACommonTypes::t_UserDetails userDetails,
in TINACommonTypes::t_SessionId sessionId
);

void endSessionInd (
in TINACommonTypes::t_SessionId sessionId
in unsigned long indId,
);
};



interface i_BasicReq{

void endSessionReq (
in TINACommonTypes::t_UserId userId,
 in TINACommonTypes::t_SessionId sessionId
);
};
```

```
interface i_PartyMultiPartyExe{

void endSessionExe (
in TINACommonTypes:: t_UserId userId,
in TINACommonTypes::t_SessionId sessionId,
                              out any accountInfo
);


};
};
```