Reusable Block Provisioning for Application Service Providers with Parlay/OSA

Opeyemi Oni and Hu Hanrahan
School of Electrical and Information Engineering
University of the Witwatersrand
Johannesburg, South Africa
e-mail: {o.oni,h.hanrahan}@ee.wits.ac.za

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

***Index Terms*—Services, OSA/Parlay, Software Reuse**

I. Introduction

The coming together of traditional telecommunication and the Internet leads to the concept of the Next Generation Network: "The Next Generation Network (NGN) is a multi-service bearer network capable of supporting multiparty, multimedia, real-time and information services" [10]. An important objective of the NGN is to enable development and deployment of services by third party service developers on telcos' network. An Open Service Market is facilitated by the NGN [11]. A number of different strategies to accomplish the NGN suggested by different industry groups include JAIN, TINA and OSA/Parlay. The general model of NGN architecture is an *application layer* at the highest level, followed by the *service layer* with the *network layer* lowest. The application layer explores Distributed Object Computing (DOC) and Object Oriented Design (OOD) to provide service differentiation and delivery with a short development time by third party application developers. Parlay uses the concept of Application Programming Interfaces (API) to describe a set of standardized programming interfaces.

The Parlay architecture has fourteen APIs; examples include call control, user interaction and mobility management [11]. APIs are implemented on the client side (application layer) as callback mechanisms and on the server side as Service Capability Functions (SCF) in Service Capability Servers (SCS). Figure 1 illustrates the interactions. The number of SCFs and richness of methods of each SCF makes the Parlay API powerful.

A resulting problem is that the method calls used to provide services using API prove complex for developers without telecommunication knowledge, for example knowledge of the call control mechanism. While the SCF contains the detailed control logic, the application must coordinate the sequence of calls on the SCF interface. We therefore conclude that there will be reoccurring logic in the application domain to implement this coordination. This research therefore seeks to improve service development and deployment by introducing a higher layer of generic services that can be accessed by applications through simpler calls. The research question states: *Starting with Parlay APIs, how can one create reusable blocks that provide a higher level of abstraction to enable rapid application creation and deployment by Application Service Providers (ASP)?* Section II provides background to the Parlay architecture. Aspects of the TINA architecture that contribute to this work are also reviewed. Section III outlines the approach.

The Centre is supported by Telkom SA Limited, Siemens Telecommunications and the THRIP Programme of the Department of Trade and Industry

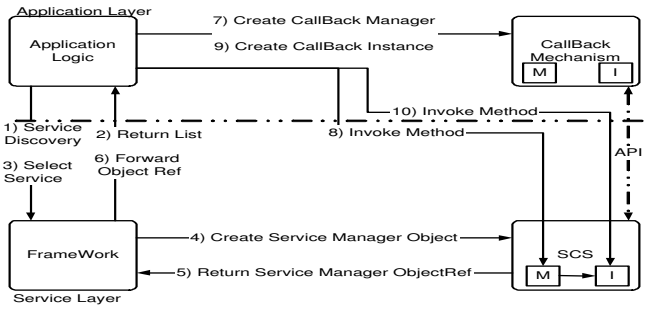

Fig. 1. Componential Interaction in Parlay/OSA

II. Background

In this section we examine factors that affect the internal structure of the Parlay application. A second architecture, TINA, plays an important role in this research is reviewed.

A. Parlay

The service architecture in Parlay is divided into two parts, the application layer, that may be located in a third party service provider domain, and the service layer that contains the SCS and is located at the network operator's domain. The ASP accesses the services provided in the service layer through SCF interfaces. The service layer is divided into the Framework and the Gateway that contains the SCF implementations [1]. The SCFs are accessed through the Framework which stores information about applications and creates the required componential object for each SCF that the application uses. In developing the API, the Parlay group incorporated a number of service capabilities including call control, messaging management and user interaction [11]. Parlay thus separates application logic from generic service logic in the SCFs implementing the APIs

for example call control, messaging and event notification. The sequence of events that takes place while using the Parlay architecture to provide a service is shown in figure 1. The installation of SCS at the framework is assumed. The sequence starts where an application requires the capability offered at the SCS. The sequence in figure 1 is as follows:

- The application logic uses the service discovery interface to find the services that are registered at the Framework (1). The framework then returns the set of services the application is allowed to use (2). The application logic then selects a service (3); the framework then uses the SCS to create a service manager object (4) and the reference is returned to the application logic (steps (5)&(6)).
- The application logic then decides it needs to invoke a method on the SCF. It creates the callback interface for the object manager (7); the callback mechanism returns the reference of this object to the application logic (not shown). The application logic then uses the callback reference to invoke a method (for example to create a call) on the service manager object in the service layer (8). An event happens in the network and is propagated to the application logic through the callback mechanism. The application logic can now create callback and object instances (steps (9) & (10)) to handle the event.

The application logic must therefore select capabilities to invoke in the SCF by calling methods on instance objects and respond to network events. This complexity raises the question of whether is there is a way to separate the application specific logic from the generic logic that routinely interacts with the Gateway SCFs. A similar concept that separates application and generic logic in the TINA architecture [11] partly supports this proposition.

1) Parlay Services: Parlay service provisioning is accomplished through division of the interfaces, which allows delegation of call management work from common to specific requirement. Elaborating on this the interfaces include [4]:

- Service manager: This object is created for each application that communicates with the SCF. its purpose is to create new objects as needed by the application in the SCF. The combination of the service manager and objects it creates are referred to as a service instance.
- Call: represents a relationship between a number of parties i.e it relates all parties that have an association. Therefore it facilitates communication between parties. Different applications can have access to the same call object but share different views, with one application at the originating side and the other at the terminating side. As a result the applications are unaware of each other presences. Which implies the entire call is released from the applications perspective if an application releases the call object.
- Call Leg: The CallLeg object associates a Call object with an address. As a result attaching a call leg implies routing the call to the target address (or from originating address) and enabling the media or bearer channel in order to allow communication with the other parties connected to the call.

B. TINA

The service architecture in TINA encapsulates a number of concepts and principles. One relevant to this discussion is the session concept. A session is a period of time during which activities are performed to accomplish service goals. Of interest are the *service session* which provides management for a single service and a *user session* provides the management of a user within a service session. TINA uses defined computational objects to implement sessions. The Service Session Manager (SSM) contains the service logic while a User Session Manager (USM) represents each party to the service. The SSM is used for overall service coordination and the USM being used for user specific service management [2]. TINA also defines feature sets by grouping interfaces on the SSM (and USM) by service types of different complexity, for example the set of methods to suspend, modify, create parties. TINA is limited in that the available feature sets are confined to multiparty session control and that the SSM is a complex computational object containing both generic and application specific logic. Work in [11] demonstrates the separation of the TINA USM and SSM into application-specific and generic logic for different feature sets. We explore how some of the concepts of this separation principle can be adapted to the Parlay application layer.

III. Approach

This research aims to provide generic logic to shield individual Parlay applications from the complexity of the full Parlay API. While Parlay separates application and service logic, the application developer would benefit from further abstraction of the API. This research seeks to ease application development by having generic logic that is easily accessible to the ASP through simple methods (e.g. *create()*). TINA does not directly map to Parlay because of the latter's greater range of functionality, for example mobility and user interaction. We follow the general approach of [11] but seek a fuller range of generic logic that invoke Parlay APIs. Generic logic falls into two categories:

- That which can be mapped onto the TINA approach in [11], namely the Call Control capability. TINA's session models can be mapped onto the call control objects in Parlay. For example a single user session in TINA maps onto a call leg in Parlay. The Call Control API will therefore use TINA feature sets to cater for all possible generic logic in terms of methods applicable to different feature sets.
- That which cannot be mapped to TINA, namely all other capabilities, for example user interaction and mobility management.

To handle the other APIs that are not catered for in TINA we analyse sequences of messages and events that usually occur during the use of the API and group such sequences to be invoked by simpler method calls, thereby providing the generic logic required by many applications.

A. Architectural Overview

In this section we give an overview of the Architectural Design in terms of the extended Parlay model. This is done by looking at the componential interaction in the structural and service session models. Figure 2 shows the separation proposed in the application layer, with each application having its own Provider Application (PApp) logic and using the Generic Application (GApp) logic provided in the GApp layer. As a result the ASP does not need knowledge of detailed APIs and simply uses the methods provided in the GApp layer.

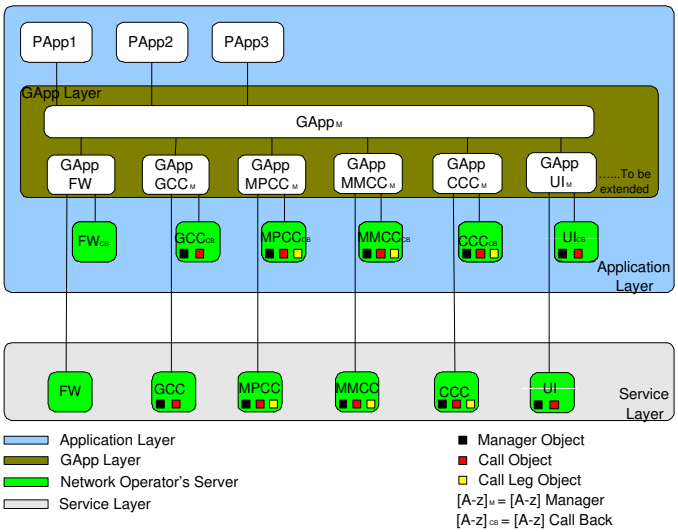

Fig. 2. Architectural View

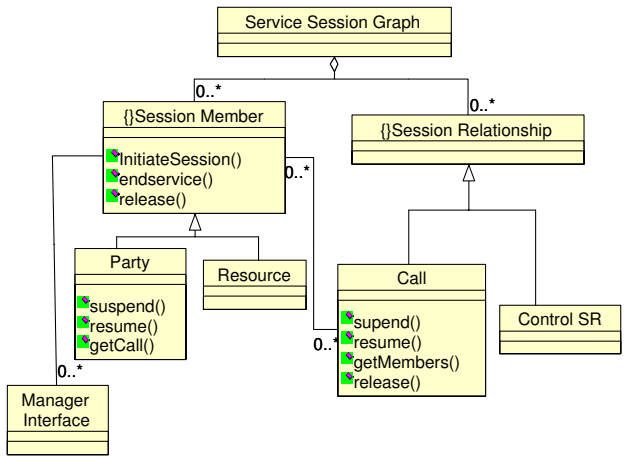

Fig. 3. Service Session Graph

1) Structural: The structural design was accomplished by dividing GApp into two level of abstraction. The lower level handles the complexity of SCF interfaces and higher level provides a simplified interface for communication with the ASP. In order to communicate with the SCF in a structured form each SCF service manager has a GApp SCS manager which manages complexities brought about by the specialization of each SCF service manager. The ASP communicates with the GApp SCS managers through a simple interface that hides GApp's structure. The result is abstraction through the following scheme.

- SCF methods that do not refer explicitly to objects that implement the call session can be accessed from the GApp Manager interface. Otherwise a method is created in the GApp layer. This method acts as a mediator containing the logical interaction of the SCF object therefore hiding its complexity from the application.

Figure 2 shows the architectural view of each of the components. In the GApp layer there are seven components:

- *GApp*$_M$ Hides GApp structural information from the ASP. It is included in the design to provide a simple interface for the ASP, therefore it provides a higher level of abstraction. The rest of the GApp interfaces provide a simple set of interfaces for GApp$_M$ and interact with the network operator interfaces and call back through a more complex interface.
- *GAppFW* Manages creation and removal services at the ASP's request, also providing service discovery functionalities.
- *GAppGCC*$_M$ Manages the GCC interface and its call back interface therefore hides GCC complexity. It provides a simple set of interfaces for GApp$_M$ and interacts with the network operator interface and call back through a more complex interface.
- *GAppMPCC*$_M$ Manages the MPCC interface and its call back interface therefore hides MPCC complexity.
- *GAppMMCC*$_M$ Manages the MMCC interface and its call back interface therefore hides MMCC complexity.
- *GAppCCC*$_M$ Manages the CCC interface and its call back interface therefore hides CCC complexity.
- *GAppUI*$_M$ Manages the UI interface and its call back interface therefore hides UI complexity.

2) Service Session Model: In this section we look at the active components during service delivery. Each of the active members has operations applicable to it. Going through figure 3, the Session Member is a collection of participants and is an abstract class. The Session member is specialized to a party (Call leg end) or resources. Session members are associated through calls. Whilst the operation that can be accomplished is specified in control SR. Control SR also act as an abstraction provider by providing exception messages to the user at the level required therefore it is a part of the GApp manager shown in figure 2. This model provides a global view of the interaction between the SCF object and the GApp methods that can be called on the SCF objects, for example a party (SCF call leg object) can be suspended, released etc.

Sessions are used to identify relationships between parties. Therefore relationships such as calls and call Legs sessions are identified by sessionIDs which are 32 bit integers. Parlay requires that a session ID be "unique within the context of a specific instance of an SCF" [5]. We adopt sessionID in GApp interface to uniquely identify each Call and call leg objects in a GApp Manager instance. This is done through structurally naming the call and the corresponding call legs that belong to the call. The call is identified by a call ID which is a 32 bit integer whilst the call leg is identified by appending the call ID to the call leg's (target/originating depending on whether the leg is an originating or terminating leg) E164 number. On invoking a service with the call ID or call ID with E164 number appended, it is mapped by GApp to the appropriate call and call Leg identifier respectively. An identify is a location and a session ID; the location is used to find the SCF instance that performs the logic and the session ID is used to identify the session (usually passed as an input parameters).

B. Overview of GApp use case analysis

A number of generic GApp methods where identified by observing the different stages that occur in the life cycle of services offered. These include

- Creation of service:
  1) Obtaining SCF Manager.
  2) Creation of service instance:

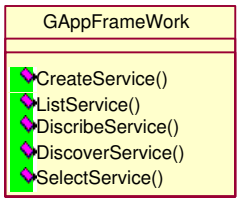

Fig. 4. GApp Framework class

a) Enabling notification points to prompt session
b) Application creation of session

- Service Usage:
  1) Detection of notification point.
  2) Initialization of appropriate interface.
  3) Usage of this interface to perform required service.
  4) Deletion of the interfaces thereafter.
- Remover of Service:
  1) Removal of service instance
     a) Disabling/destruction of Notification point.
     b) deletion on session.
  2) deletion of SCF.

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

C. Creation of Frame Work reusable Block

GApp Frame Work (GAppFW) manages the creation and deletion of services. As a result provides the application service provider with access to

1) Service List.
2) Service Description.
3) A reference to a new SCF manager.
4) Disposition of SCF managers and resources

D. Creation of SCF Reusable Blocks

Depending on the SCF manager returned by the framework, GAppFW creates the appropriate GApp SCS Manager by querying the database to obtain the list of the appropriate interfaces and required logic. GApp SCS Manager contains translation of the higher level methods that is provided on its interface to methods that is executed on the SCF and vice versa. When application logic executes an operation on GApp manager it forwards the message to the appropriate GApp SCS Manager interface, which checks its list of operation and executes the logic for that operation. A GApp SCS manager exist for each service manager obtainable in Parlay. Therefore there is a one to one relationship between the GApp SCS managers and the service managers in the SCF. The information provided in the rest of this section was obtained by combining interfaces obtained from [5–9] to provide the global architectural logic.

1) GApp SCF Design Principle: We need to provide reusable logic in a structural and extendable way. To do this we adapt a well defined way of defining logical blocks and relating the different SCF. Structural provisioning of the reusable blocks was accomplished by rigorously analysing the specification and sequentially combining SCF methods to provide logical blocks (functions) that provide the application developer with powerful functional interfaces. Extensibility is provided

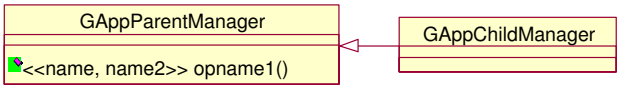

Fig. 5. Class diagram for GApp manager inheritance

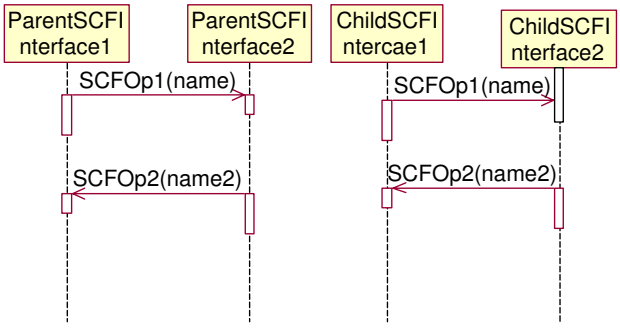

Fig. 6. Sequence of events used to accomplish opname1() in figure 5

by exploiting the relationship that exist between different SCF interfaces (inheritance) and reproduced the same set of relationship between GApp SCS manager classes. We needed a way to transform inheritance relationships between SCF classes to inheritance relationship between GApp SCS manager Classes. Hence a child GApp SCS manager inherits all parent GApp SCS manager functions. This is elaborated in figure 5 and 6

To enable scalability we define a set of rules for use when developing new methods for the child GApp call manager classes. A new function is created in the child GApp SCS Manager class if logical changes occur in comparison to the parent's logical blocks. Logical changes occur when (The parent GApp class and SCF sequence are as shown in figures 5 and 6)

- The sequence of events that accomplish a function change:
  - When a new function is introduced in the child GApp SCS class. This is explained with the diagrams in fig-

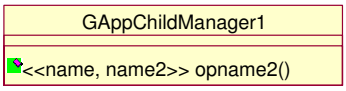

Fig. 7. Class diagram for a GApp child Manager with a new functional block

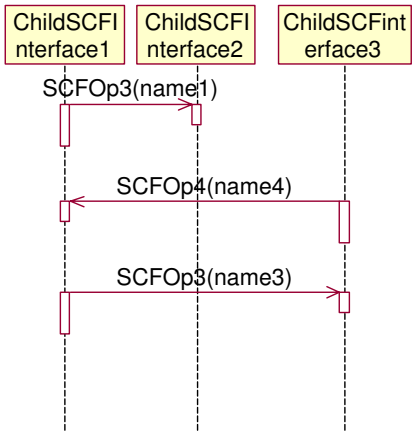

Fig. 8. Sequence diagram of events used to accomplish opname2() i.e the new logical block in the child GApp SCS manager.

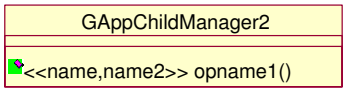

Fig. 9. Class diagram for a GApp child Manager with a changed functional block

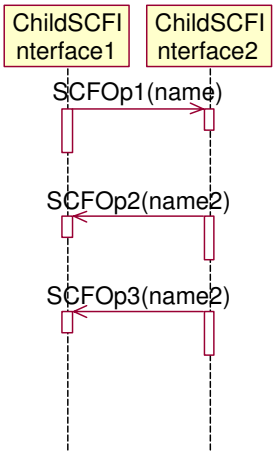

Fig. 10. Sequence diagram for a GApp child Manager with a changed logical block

ure 7 and 8.

- When the number of method calls made to accomplish a function by the child GApp SCS manager on the SCF interfaces is different relative to the parent function. This is shown in figures 10 and 9
- When no possible mapping can be established between the parameter set of the child and the parent function. That is the number of parameters have changed or each child object parameter cannot be mapped to a distinct parent object parameter. This is shown in figures 12 and 11

Using these design principle we obtained the GApp class diagram in figure 13.

2) SCF Service Model: Firstly we provide a general view of the Parlay service logic by discussing the GAppServiceM-

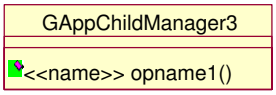

Fig. 11. Class diagram for a GApp child Manager with a changed functional block

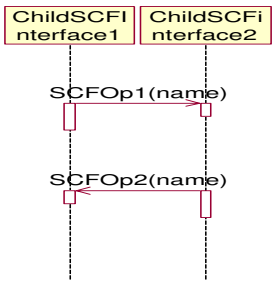

Fig. 12. Sequence diagram for a GApp child Manager with a changed functional block

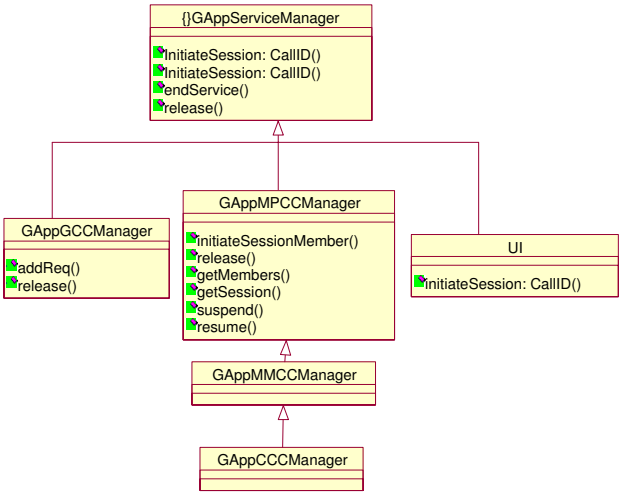

Fig. 13. GApp Call manager inheritance Structure

anager. As shown in figure 13 the GAppServiceManager is an abstract class that forms the bases for all GApp SCS Manager Classes. It contains methods that all GApp SCS Manager Classes should possess. This was obtained by studying GCC, MPCC, MMCC ,CCC and UI. GCC is dealt with in this research to provide a general view of the mechanism used to deploy services in Parlay. It was identified that the objects and logic used for service deployment is the same with the newly enhanced SCF, but some of the method have being further enhanced for example enableCallnotification is implemented in MPCC by createNotification and enableNotification whilst routeReq is know provided through eventReportReq and routeReq. The study of GCC provided us with the general concept of the logical groupings of reusable blocks that can be obtained in Parlay. As a result the design is robust as it is easily comprehendible and extendable.

Figure 13 shows the overall view and relationship between GApp SCS Manager classes. This was modeled using the relationship between GCC, MPCC, MMCC, CCC and UI. The relationship is such that CCC inherits from MMCC which inherits from MPCC. Therefore as the Parlay call control interfaces get more complicated (through provisioning of more functionality) we can follow the inheritance structure to structurally provide enhancement to reusable logic obtained in the parent class in the GApp model. As a result we obtain a basic set of reusable blocks for the base class and extend these blocks to cater for the functionality provided by the more complex Parlay interfaces. The complexity brought about in the child GApp class is handled by extension of the logic from the parent GApp class and the creation of new logic in the child as required.

IV. Example

In this section we give an example of the logical groupings in the GApp layer. The GApp method discussed is CreateService on the GApp frame work interface. The logic to accomplish this is provided through the aid of a sequence diagram. The original sequence was established using [3]. The GApp Manager is not included in the sequence diagrams for simplicity, but it is the only interface that the application communicates with. Hence

the application communicates with the other GApp interface through the GApp Manager.

In order to provide a service list we require two interfaces, the first provides access to the service discovery interface (CreateService) whilst the other is to list services available on these interface (ListService not discussed in this paper). Figure 14 shows the way GApp is used to provide service discovery interface.

- The application logic executes createService() (1) on the GAppFW interface(On GApp Manager which executes it on GAppFW). (2) GAppFW queries the database to obtain the configuration script for this particular service. The configuration script contains information such as domain ID authentication interface, authentication mechanism, singing algorithm, agreementText etc (These are information that is specific to the ASP for the SCF to be obtained).
- GAppFW triggers initiateAuthenticationWithVersion() (3) which initiates authentication with a publicly available framework (through url or orb) by passing the client domain's identifier, the authentication type, the framework version number and returns the Frame Work identifier.
- The next operation invoked is the selectAuthetication-Mechanism (4) which provides the Frame Work with a list of authentication mechanisms the ASP supports. The framework selects one from the list and it is passed back as the return parameter. (5) Depending on the authentication mechanism a number of challenges are invoked on the ASP, which it must respond correctly to using the return parameter. (6) The authenticationSucceeded is then invoked to inform the ASP of its success. (7, 8) There after depending on the ASP's policy it might authenticate the Frame Work with a similar process.
- The ASP then invokes the requestAccess (9) which return the reference to the Frame Work access interface that provides access to other Frame Work interfaces. (10) Before the client can use the Access interface it must provide a list of all signing algorithms it supports for use in cases where digital signature is required through the operation select-SigningAlgorithm. The Frame Work selects ones from the list and it is passed back as the return parameter. (11) The client then invokes obtainInterface which returns the reference to the required Frame Work interface in this case service discoveries.

In figure 14 message 3-11 is the normal sequence which was accomplished by the ASP previously but know is all performed by the GAppFW.

V. Conclusion

This research creates reusable communication session control blocks at a higher level of abstraction for the application programmer in the Parlay environment. The approach builds on, but goes beyond a method of splitting generic and application logic developed in the TINA environment.

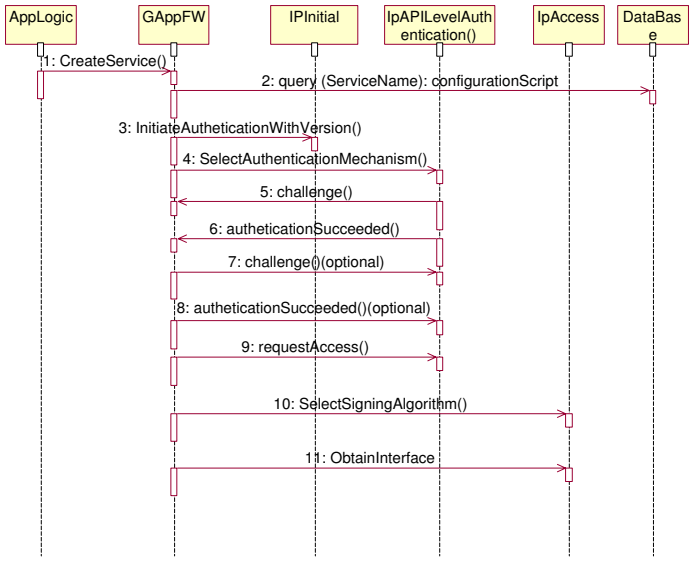

Fig. 14. Application Initiates Service

References

[1] A. Moerdijk and L. Klostermann. Opening the Networks with Parlay / OSA: Standards and Aspects behind the APIS. http://www.parlay.org/specs/library.

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

[3] 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.

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

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

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

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

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

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

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

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