

https://dspace.org/

A simplified approach to application layer service signalling

Horwitz, Doron Leon

2011-10-21
http://hdl.handle.net/10539/10622

Downloaded from a DSpace repository. https://dspace.org/

A SIMPLIFIED APPROACH TO APPLICATION LAYER SERVICE SIGNALLING

Doron Leon Horwitz

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

Johannesburg, 2011

Declaration

I declare that this research report is my own, unaided work, other than where specifically 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 _____ 2011

_____

Doron Leon Horwitz

Abstract

The modern approach to complex services has some shortcomings which overcomplicate the implementation of application servers supporting service logic and also the actual structure of these services. Modern service development environments rely heavily on bearer network call/session signalling for the transport of service messages to remote terminals and manipulating bearer streams in the network.

This research report looks at the overheads which occur when the bearer network is used as the main transport for distributed service logic and then goes on to propose application layer focused service development as a replacement for this. Service logic developed in the application layer should be constrained to the application layer so as to avoid leaky abstractions and make service development more intuitive to programmers who do not have deep knowledge of telecommunications technologies since the details of the bearer network do not play a part in the application layer. Application layer signalling is introduced as a concept very important to keeping service logic in the application layer, by allowing service messages to bypass the bearer network. In this way, service sessions are started and maintained in the application layer and lower-layer functionality is only called on when bearer network streams are required by a service.

A framework is developed to support application layer focused service development. This framework acts as support for decoupled service logic by allowing easy abstracted use of application layer signalling and bearer network functionality. It also provides a simplified means for managing service logic. Reusability is also built in in the form of reusable building blocks which abstract out various functionality including that of the bearer network.

Using example services designed to be supported within the framework, the ideas of application layer focused service development are proved to simplify service development and offer robust support to services. Whilst this research report does not attempt to standardise the technologies used to constrain service logic in the

application layer, it does put forward important concepts which, when implemented, would enhance service development environments by providing a strong platform on which to develop services.

To Adrian.
You grew up faster than all of us.

To Dad.
You still had a lot to do.

Acknowledgements

The following research was performed under the auspices of the Centre for Telecommunications Access and Services (CeTAS) at the University of the Witwatersrand, Johannesburg, South Africa. This centre is funded by Telkom SA Limited, Nokia Siemens Networks, the Department of Trade and Industry's THRIP programme and Vodacom SA. This financial support was much appreciated.

I would like to thank my supervisor, Professor Hu Hanrahan, for his wisdom and continued support, guidance and patience throughout the duration of our research. I would also like to thank the members of staff at Wits University's School of Electrical and Information Engineering and my colleagues at CeTAS for their support and advice. Finally, I would like to thank my parents and sister for their love and support and for always putting up with me.

Contents

Declaration ii

Abstract iii

Acknowledgements vi

Contents vii

List of Figures x

Abbreviations xii

1 Introduction 1

1.1 The Current State of Telecommunications Services 1

1.2 Problems of Modern Telecoms Services 3

1.2.1 Call/Session Service Signalling 3

1.2.2 Unstructured Service Development Environment 9

1.3 Proposal for a Modern Service Development Environment 11

1.4 Conclusion 12

2 Application Layer Signalling 15

2.1 Direct Application-to-Application Signalling 16

2.2 Separating the Bearer Network from Services 18

2.3 Application Layer in a Technology Neutral Framework 20

2.4 Ideal Application Layer Signalling 21

2.4.1 Service Operation Centralised in the Application Layer . . . 21

2.4.2 Signalling Protocol and Structure 22

2.4.3 Appropriate Bearer Network Abstraction 23

2.5 Important Clarifying Concepts 23

2.5.1 Atomic and Composite Services and Orchestration 23

2.5.2 Types of Service Initiation and Invocation 24
2.5.3 Bearer Connectivity Control and Monitoring 25
2.5.4 Signalling Requirements in the Application Layer 30
2.6 Conclusion 34

3 Supporting Software Framework 36
3.1 The need for a Formalised Application Layer Signalling Framework . 36
3.1.1 Formalising The Level of Abstraction 37
3.1.2 Meeting The Needs of Direct App-to-App Signalling 38
3.1.3 Supporting Application Layer BCCM 39
3.2 Guiding Principles and Design Choices 40
3.2.1 Object-Orientation 40
3.2.2 TINA 42
3.2.3 AAA 43
3.2.4 Signalling Protocol 45
3.2.5 Dynamic Object Loading and Method Invocation 49
3.2.6 Message Context 50
3.2.7 Reusable Functionality 51
3.2.8 Location of Essential Value-added Functionality 52
3.2.9 User Identity 53
3.3 Main Framework Components 53
3.3.1 Reusable Building Blocks 54
3.3.2 Application Layer Signalling Interface 58
3.3.3 User Profiles 61
3.3.4 Managers 63
3.3.5 Dynamic Behaviour of Supporting Infrastructure 69
3.3.6 Services 79
3.4 Conclusion 88

4 Example Services 90
4.1 Presence Service 90
4.1.1 Functionality and Structure 91
4.1.2 Dynamic Operation 93
4.2 Two Party Call Service with Presence 98
4.2.1 Functionality and Structure 99
4.2.2 Dynamic Operation 100

4.3 Multiparty Call Control RBB 103
4.3.1 Functionality and Structure 104
4.3.2 Dynamic Operation 106
4.4 Conference Call Service 111
4.4.1 Structure and Functionality 111
4.4.2 Dynamic Operation 115
4.5 Conclusion 123

5 Conclusion **125**
5.1 Service logic exclusively in the application layer 125
5.2 Framework 127
5.3 Main features 127
5.4 Future work 129
5.5 Summary 130

A Proof of Concept **131**
A.1 Implementation Tools 131
A.1.1 Application Server 132
A.1.2 Terminal 134
A.2 Result 135

References **137**

List of Figures

1.1 Use of call/session signalling in a modern service infrastructure . . . 3
1.2 SIP signalling in the IMS implementation of presence 8
1.3 Intuitive signalling for presence 9
2.1 Operation of services without and with ALS 17
2.2 Relationship of services and bearer network to transport network . . 19
2.3 Context of research within NGN framework layered system 21
2.4 Stages in the progression of service initiation and invocation 25
2.5 Simplified service initiation and invocation with ideal ALS 26
2.6 Bearer connectivity FSM 29
2.7 Example of OO bearer connectivity control and monitoring 30
2.8 Differentiating between sideways and downwards communication . . 31
3.1 Access restriction types 44
3.2 UML class diagram of overall framework 54
3.3 Basic operation of the two party call RBB 55
3.4 Classes of the two party call RBB 56
3.5 ALS Classes 59
3.6 AAA service objects in context of user profiles 62
3.7 User profile class 62
3.8 Service manager classes 65
3.9 RBB manager classes 67
3.10 Application server class 68
3.11 Terminal connection and login SD 70
3.12 SD of remote method call from terminal to AS 71
3.13 SD of remote method call from AS to terminal 73
3.14 SD of a call to an RBB's method 75
3.15 Terminal disconnection and logoff SD 77
3.16 Classes of the call state model RBB 80
3.17 Classes of the two party call service 81

3.18 SD for the setup of a two party call 83
3.19 `CallModel` association to `Call` 84
3.20 SD for hanging up a two party call 86
4.1 Presence service classes 92
4.2 Example presence service subscription mappings 93
4.3 Presence service registration SD 94
4.4 Presence service subscription SD 96
4.5 Presence availability update SD - remote 97
4.6 Presence availability update SD - within AS 98
4.7 Two party call and presence integration - classes 100
4.8 Two party call and presence integration - `handleMakeCall()` hook . 101
4.9 Two party call and presence integration - `handleHangup()` hook . . 102
4.10 Multiparty call RBB classes 104
4.11 SD for creating a multiparty call 106
4.12 SD for adding a leg to a multiparty call 107
4.13 SD for removing a leg from a multiparty call 109
4.14 Destroying a multiparty call 110
4.15 AS conference call service classes 112
4.16 Terminal conference call class 114
4.17 SD for joining conference call service 116
4.18 SD for creating a conference call 117
4.19 SD for joining a conference call 118
4.20 SD for transferring ownership of a conference call 120
4.21 SD for a terminal leaving a conference 121

Abbreviations

AAA	Authentication, Authorisation and Accounting
ALS	Application Layer Signalling
API	Application Programming Interface
AS	Application Server
BCCM	Bearer Connectivity Control and Monitoring
BCSM	Basic Call State Model
CORBA	Common Object Request Broker Architecture
CS	Call/session
CSCF	Call Session Control Function
DCOM	Distributed Component Object Model
DP	Detection Point
EDP	Event Detection Point
FSM	Finite State Machine
GUI	Graphical User Interface
HLR	Home Location Register
HSS	Home Subscriber System
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technologies
IMS	IP Multimedia Subsystem
IN	Intelligent Network
IP	Internet Protocol
ISUP	ISDN User Part
IT	Information Technology
JAIN	Java APIs For Integrated Networks
JCC	Java Call Control
JSLEE	JAIN Service Logic Execution Environment
JTAPI	Java-based Telephony Applications Programming Interface
NGN	Next Generation Network
OMA	Open Mobile Alliance

OO	Object–Orientation
OS	Operating System
OSA	Open Service Architecture
OSE	OMA Service Environment
PLMN	Public Land Mobile Network
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RA	Resource Adapter
RBB	Reusable Building Block
RCMF	Resource Control and Management Function
REST	Representational State Transfer
RMI	Remote Method Invocation
S-CSCF	Serving Call Session Control Function
SCF	Service Control Function
SCF	Service Control Functionality
SCF	Service Capability Feature
SCP	Service Control Point
SD	Sequence Diagram
SDP	Service Data Point
SIB	Service Independent Building Block
SIP	Session Initiation Protocol
SMS	Short Message Service
SOA	Service Oriented Architecture
SS7	Signalling System No. 7
TINA	Telecommunication Information Networking Architecture
TIPHON	Telecommunications and Internet Protocol Harmonization Over Networks
UML	Uniform Modelling Language
VLR	Visitor Location Register
VoIP	Voice Over IP
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 The Current State of Telecommunications Services

The modern telecommunications (telecoms) infrastructure places great emphasis on value-added services (or simply, *"services"*). The number and complexity of services are increasing beyond those first proposed by the Intelligent Network (IN) standards. These enhanced services include presence, instant messaging, push-to-talk and video-on-demand and require an infrastructure to support more complex media formats (such as video) and greater flexibility in service design. In response to these improved services, telecoms operators (telcos) are restructuring their networks at administrative and technical levels and equipment vendors are upgrading the capability of the hardware they are marketing to telcos and end-users. These changes follow on from the move towards the Next Generation Network (NGN).

For the service developer there are two facets to these enhancements. The first is that of the service development environment. Where in the past they were constrained to developing services at a low-level using the C programming language, telecoms service programmers now have a large inventory of programming languages, platforms, application programming interfaces (API) and application server (AS) implementations at their disposal. All of these tools cater for the development of more complex services by providing reusable components and abstracting the programmer from less significant service detail thus making for a more intuitive service development process. The other facet of these enhancements is that of the final products of service development. Due to simpler service development and more powerful terminal and server hardware it can be expected by the end-user that services will be adequately complex to cater for the various forms of multimedia

used by the modern technology user. The functionality and quality of new services should at least match those of the distributed applications which are in common use, such as those found in the Internet, like social networking, online gaming and file sharing.

Modern telecoms standardisation is attempting to incorporate this modern stance towards services: at the service layer, the NGN calls for the integration of services with complex functionalities into the service environment [1]. Similarly, the IP Multimedia Subsystem (IMS), which is the current main contender for the complete packet-switched telecoms network core, has been created to allow

> *"public land mobile network (PLMN) operators to offer their subscribers multimedia services based on and built upon Internet applications, services and protocols."* [2]

It is possible to view the current telecoms service environment as a halfway point between the legacy IN which is controlled, limited and more robust and the Internet which is very open but lacks any service guarantees such as quality of service (QoS) control. Various examples illustrate. Firstly, we see that individual Internet services have to implement their own forms of authentication, authorisation and accounting (AAA) or use what is provided by other 3rd-party developers, whilst modern telecoms services can rely on the underlying network for these functions. Further, we see that the Internet has no central body which enforces quality control, whilst telcos can ensure quality control on the software which makes up a complex service – a necessity, since customers will direct complaints about, or technical support requests for a 3rd-party developed service to the telco itself. From the reverse perspective, the IN has only basic benchmark services, all oriented towards two party or very basic multiparty voices calls. The benchmark services for the modern telco services include advanced multiparty voice call control converged with multimedia services such as instant messaging and video calling. This integration of telco and Internet mentalities has already been considered in the overall IMS perspective [3].

The service environment of the NGN has the advantage of being able to incorporate the best aspects of services in the Internet and legacy telecoms, whilst ensuring that it does not suffer from their shortcomings. However, the current trends in services demonstrate that the telecoms world has still not properly integrated the advantages of the Internet and legacy IN paradigms.

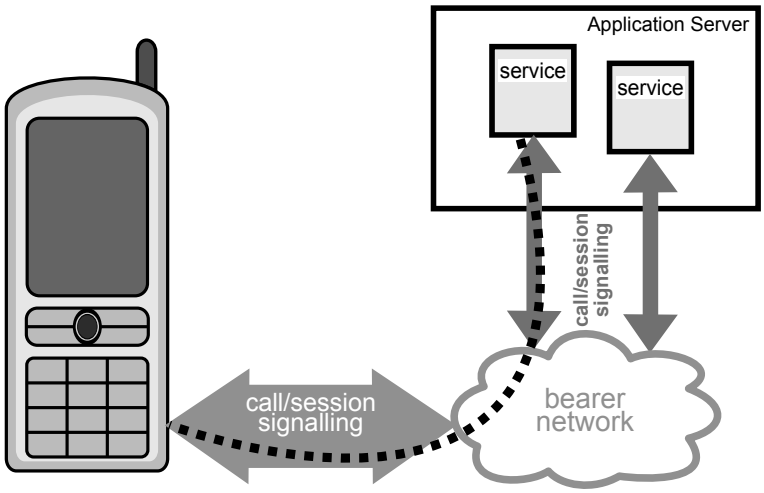


Figure 1.1: The use of call/session signalling in a modern service infrastructure. The dotted line indicates the path that all messages take when travelling between end-points during a service session.

1.2 Problems of Modern Telecoms Services

Whilst the end-user is perceiving improvements to modern telecoms services with enhanced user interfaces and flexibility, it is in the development and protocol implementation of the services where problems have arisen. Certain legacy approaches to service infrastructure have become entrenched in modern service technologies. These approaches may have been applicable to much simpler services and necessary for a closed service development environment, but for the complex multimedia services which consumers require today these approaches can be seen as shortcomings in development and performance. Descriptions of the weaknesses of current NGN service architectures, such as IMS, follow.

1.2.1 Call/Session Service Signalling

The modern service environment makes heavy use of call/session (CS) signalling. The issues with this are twofold. The first problem relates to services' direct use of CS signalling protocols to interface with the underlying network, where an abstracted API would be more suitable. The second problem relates to the use of CS signalling between the end-points of a service, for which it is unsuitable. These problems are detailed below.

Programming and Interfacing with The Network

In the legacy IN, the service control function (SCF) resided in the network core, as part of the bearer network, and contained the logic for services. The logic was triggered at detection points according to the Basic Call State Model (BCSM) [4] and would execute on physical entities controlled by the telco. Service signalling was implemented using the Signalling System no. 7 (SS7), an infrastructure and set of protocols orientated towards call setup and tear down, which provided an adequate and robust service infrastructure for the benchmark services of the time. Since the capabilities of these services were a lot simpler, SS7 was designed for a closed development environment where services were vertically integrated into the rest of the network. However, due to SS7's being made up of very low-level transport-oriented protocols, implementing services required an intimate knowledge of both the details of the host network and of telecoms technologies. Despite the prospect of building services quickly, which was touted for the IN, the timescale for developing a single service was in the order of a year, if not more [5, pp 23].

The modern stance towards services promotes horizontal integration: services can be hosted and developed by both a telco itself and by 3rd-party providers. This means that the telco must give outside service programmers access to the resources provided by the network and also provide simplified and abstracted interfaces into these resources. Ideally, utilisation of the access provided to outside developers should not require intimate knowledge of the technical, administrative and business workings of the network. At the same time, the manner in which access is granted should give assurance that the services developed by a 3rd-party will fully meet the high quality specifications set by the telco. To achieve this, telcos need to make a lot of decisions on behalf of the programmer, meaning that high abstraction is required. This is due to the inherent complexity of telco infrastructures and the distributed nature of services. Following on from this, one would consider it bad practice for a telco to provide an interface which would require direct manipulation of CS signalling protocols like ISDN User Part (ISUP) or Session Initiation Protocol (SIP). Yet, IMS makes heavy use of SIP and as such its native AS, the SIP AS, hosts and executes multimedia applications which interface with the network using SIP directly [6, pp 33] as shown by the vertical arrows in figure 1.1. This means that programmers developing services which will be deployed in an IMS context have to be knowledgeable in the usage of SIP. Some do not view this as problematic [7], however it is a lot more intuitive to use an abstracted interface into the network.

A well-engineered system uses abstractions to ensure that its users need not have an understanding of its internal workings or in the case of a layered system, its lower layers. If we view a service development environment as a layered system, then a service programmer should be able to expect an adequately abstracted interface into the functionality of the underlying network. This means that a programmer should not have to manipulate directly or even have knowledge of CS signalling protocols' messages and parameters. Whilst these protocols play a major role in determinining a service architecture's capabilities and robustness, their correct usage should remain the responsibility of the telco. Should this not be the case and the abstraction between the service development environment and the CS system have incomplete separation of concerns, then the result is a *leaky* abstraction whose interface provides an insufficiently complex model of the CS system [8, pp 83] – as is certainly the case with the IMS and the SIP interface used by service logic in its ASs.

This point is emphasised further when protocols are compared with APIs. Whilst both provide a degree of isolation between two logically different systems, their purposes differ. An API allows an application to access the functionality and resources of the underlying platform on which it is executed. The underlying platform is presented as a model, which application logic can manipulate to achieve a certain objective. The goal of (standard) protocols is to provide interoperability between different distributed entities [5, pp 25]. If SIP (which is a protocol) is used as an API, then the model of the underlying network which it presents is unsuitable for service development, since it is incapable of describing the overall logical functionality of the network and thus is not complex enough for abstracted CS control. This is partly due to the fact that an AS in the IMS interacts only with Serving Call Session Control Function (S-CSCF), considering only the messages and parameters which have to get passed to it, and the error conditions generated by it without considering the many other events (message passing and errors) that occur amongst the other Call Session Control Functions (CSCF) and Home Subscriber Systems (HSS). This means that the model that the IMS provides to the AS is too simple as it views in isolation a small part of a much larger system.

A properly designed service development framework would use APIs like the Java APIs for Integrated Networks (JAIN), the Java-based Telephony Applications Programming Interface (JTAPI) and Parlay to ensure that a service does not require the fine details of the resources and functionality of the underlying network, but still be supplied with holistic and simplified control and an appropriately detailed model of the network. Depending on the angle from which this is viewed, providing such interfaces could be seen as following either the *Façade* or *Bridge* software

design patterns. Looking from the bottom-up, the *Façade* design pattern is used where software classes are required to simplify already existing complex underlying subsystems [9, pp 185–193], such as providing a Parlay interface into the IN. From the top-down, the *Bridge* design pattern is used "when an abstraction can have one of several possible implementations" [9, pp 151–161]; this could mean that a current implementation could vary, or a completely new implementation could be used, whilst the *Bridge* remains the same – this could be seen as a reason for including the Open Service Architecture (OSA) AS as a standard entity in the IMS. In whatever way abstractions are analysed, they are a necessity for a modern service development environment. Conversely, it can be said that interfacing without abstraction with underlying CS signalling protocols when compared to interfacing via the call control APIs mentioned, can be likened to programming in assembler where a high-level programming language would be more applicable [5, pp 27].

Service Message Communication

Since the operation of services by their nature is distributed, message passing between the entities involved in a session must be robust and easy to implement. Ideally, for a modern service environment, a properly designed message passing system oriented directly towards complex services should result in a simplified development environment. This means that signalling should be designed to allow terminals to signal directly to service entities, bypassing CS entities where they are not required. Despite this being a seemingly intuitive design decision, current service infrastructures rely on CS signalling protocols with messages traversing CS entities for end-to-end service communication, as shown by the horizontal arrow in figure 1.1.

Based on this, there is a need to differentiate between protocols which are designed for CS control and those which are designed for services. CS protocols enable the setup, teardown and maintenance of bearer connections. This means that the protocols ensure that there is a path (a *circuit* in a circuit-switched network or a *stream* in a packet switched network) between the end-points involved in a call. The signals made available by the protocol facilitate the location of a destination terminal at the request of an originating terminal and the maintenance of a path throughout the length of a call. As such, all signals represent the basic operations required for these functions. For example, in discovering the path between the end-points in a call, the IMS sends a series of SIP `INVITE` messages and SS7 sends ISUP `IAM`'s (initial address message). Similarly for all other CS functionality these protocols

have sets of message flows made up of simple messages which are fine-grained as a consequence of the requirement that protocols offer a high level of flexibility to allow telcos to differentiate their bearer connectivity offerings from other telcos (note that in this research report, *bearer connectivity* is a generalisation of the concept of a simple call in legacy telecoms and refers to any kind of multimedia connection including two party or multiparty calls and voice and video streaming).

Service control protocols differ from CS protocols in that they facilitate terminals' communication with services in ASs (or SCFs in the IN context). Further, while CS protocols are designed to be used directly by the telco, service control protocols in the modern, open service environment should ideally be designed to meet the needs of the software programmers who will develop services. One cannot assume that these programmers have knowledge of telecoms' technologies or its concepts of sessions or signalling. Instead, to cater for service software programmers who come from an Information Technology (IT) background, it would be more applicable to have a service development environment based on IT concepts. Since telecoms services are distributed, we look to the IT concept of Service Oriented Architecture (SOA) which is playing an important role in bringing the telecoms and IT worlds together [10, 11]. A SOA is a distributed system which exposes services which perform well-defined operations and can be accessed by remote clients by means of message exchange [12]. Whilst coming from the IT world, a SOA can also support telecoms service control protocols. Thus it is possible to implement service signalling using SOA tools such as Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI) and the web service flavours of SOAP and Representational State Transfer (REST).

An important aspect of SOA is that, logically, once the server which houses a published service is discovered, the client using the service signals directly to the server. This coupling is increased by having the client obtain a reference to a remote object which represents the exposed service. Messages are then exchanged when the client invokes a method on the remote object. This direct signalling means there are no intermediate nodes which inspect or process the contents of messages passed between the end points in the SOA communication. Service signalling based on message passing between remote objects can therefore be computationally less expensive than the use of CS signalling which is prevalent in modern networks, such as those based on the IMS. Further, SOA is more familiar to IT software developers, than is a CS protocols such as SIP.

The IMS makes heavy use of SIP for setting up sessions. It has been shown

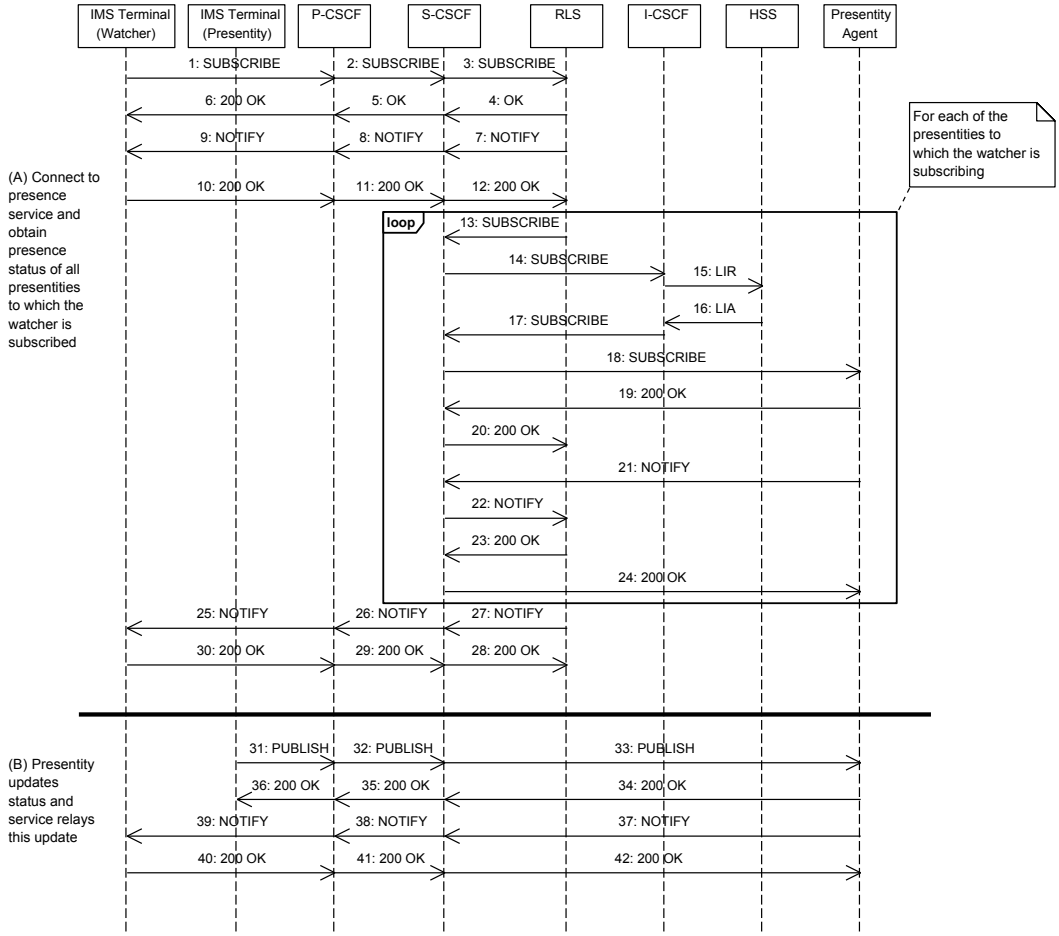


Figure 1.2: SIP signalling in the IMS implementation of presence, where both the watcher and all presentities are in the same network. Two cases are shown: (A) a watcher initially subscribing to the presence state of a list of presentities and (B) a presentity updating its presence status to a watcher.

that within the context of a modern telco infrastructure this incurs unjustified overhead and is overly complex – even for the setup of a simple two party call [13]. The presence service of the IMS requires that besides for the main entities participating in the service, messages must traverse multiple CSCF entities and the HSS. Figure 1.2 is a Uniform Modelling Language (UML) sequence diagram (SD) of certain aspects of the presence service within the IMS (adapted from [14] and [6, ch 17]). The figure shows that this large number of entities along with the use of SIP results in overly complex message sequences for seemingly simple cases of the presence service. A more logical design, which is shown in figure 1.3 and fits into the SOA approach, would firstly allow the terminal (in this case, acting as either the *watcher* or *presentity* in the presence service) to signal directly to the AS. Secondly, messages would be more logically named (corresponding to methods

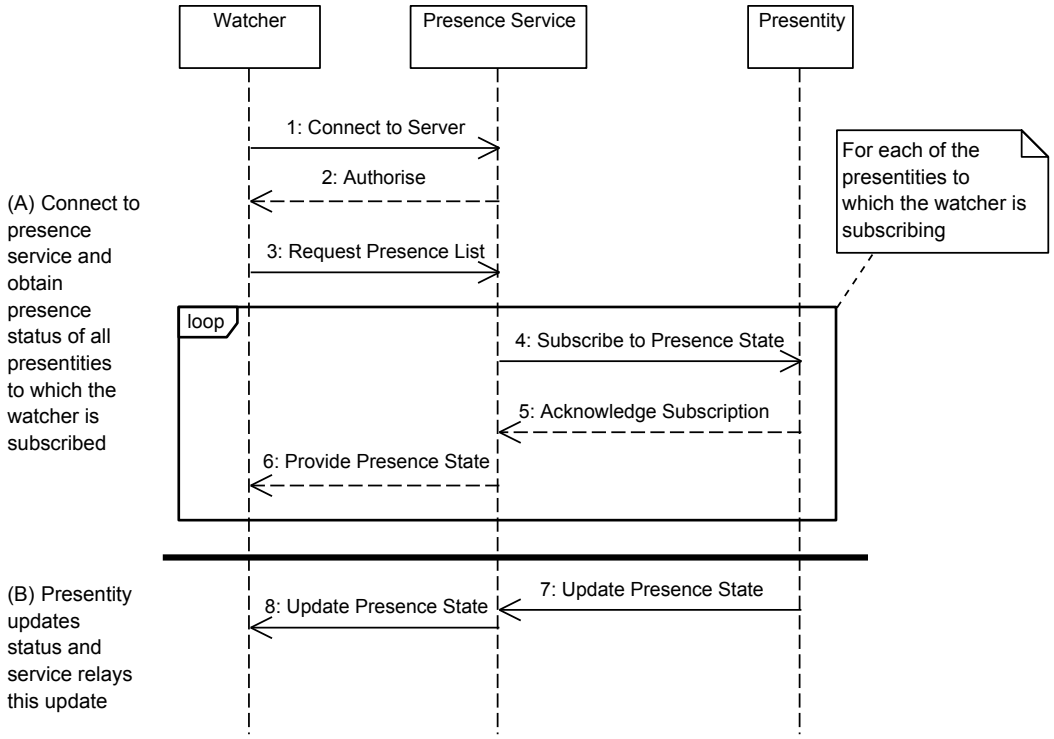


Figure 1.3: A more intuitive approach for presence signalling. Use cases (A) and (B) are the same as for figure 1.2

of remote service objects), giving a better overview of the operation of the service, thus simplifying implementation and debugging.

1.2.2 Unstructured Service Development Environment

Over the years, service development has moved further and further out of the network. Where in the IN, programmers worked directly with Service Control Points (SCP), Service Data Points (SDP), Visitor Location Registers (VLR) and Home Location Registers (HLR) which were found in the core network, it is now possible to use abstracted APIs, such as Parlay, with well-established programming languages, such as Java, to interface with the network. This opens the doors to a wider audience of software programmers allowing for a more competitive service environment.

To assist in removing service developers from the network, enabling technologies and standards which assist in both structuring services and abstracting them from the details of the underlying network have been developed. The JAIN Service Logic Execution Environment (JSLEE) structures the application logic of services into

reusable object-oriented (OO) components called *service building blocks* and uses *resource adapters (RA)* to abstract out details of the CS signalling of the underlying network [15]. Various vendors have their own service creation environments for example, [16] and [17]. Further, the Open Mobile Alliance (OMA) publishes its OMA Service Environment (OSE) specifications which formalises how services are built up using *enablers* and how access by service applications to these enablers is controlled using *policy enforcers* [18]. These technologies and standards all contribute to the modern service development environment and share ideas of abstraction, reusability and openness. However, with the various technologies available for abstracting network resources there is still no global standardised agreement on the structure or method of deployment of services which interact with the network. Moreover, even though there are links between them, the above mentioned technologies are still separate and service developers will usually choose a single one to use for development. Ideally, we would want to extract useful concepts from all of these technologies to contribute towards a standardised service development environment which will cater for a broad range of software programmers – whether their background be telecoms or IT.

One common concept is that all these solutions for service development are structured as *frameworks*. From a software perspective, a framework is a "a set of cooperating classes that make up a reusable design for a specific class of software" which "dictates the architecture of your application" [9, pp 26–28]. The software programmer develops the logic of the desired application using various abstractions provided by the framework. Then to execute the application, the framework calls on this logic. This is an inversion of control from the usual software approach where the software developer has to write code for application initialisation. The framework therefore constrains the structure of the program and decides when the application logic should be invoked. This fits in well with a telecoms environment by allowing the telco to still have some influence and control in the design of 3rd-party developed services. Further, a telco can enforce policies on what framework components are available to a service and when and by whom service functionality can be accessed. Naturally, service development environments have been built up as frameworks, however none explicitly identifies the framework concept as being a key design aspect.

While frameworks are the high-level structure, OO is an IT concept which is a very effective tool for the detailed structuring of software. Its use too has not been standardised in modern service development. Most modern programming languages support OO and good OO designs can be implemented in any of these languages.

This generic nature of OO, supported by the technology-neutral UML, makes it useful for the design of a service environment and the means of developing services within it. It is also useful for the realisation of the abstraction which was covered in section 1.2.1, since it is highly useful for describing systems from a high-level. The importance and benefits of OO are explored further in section 3.2.1.

1.3 Proposal for a Modern Service Development Environment

Using the shortcomings of modern services and their development which were identified above, this research report proposes a novel approach to service signalling and bearer connectivity abstraction as the basis for a robust platform for intuitive service development. To realise this, focus is placed on the application layer of the information and communication technologies (ICT) layering system presented in [8, ch 2]. The purpose of this is to remove as much service logic from the bearer network as possible, reducing the use of CS signalling and moving service logic computation to ASs and terminals. This is reminiscent of the design of the Internet, where complexity is kept at the edge of the network, and the network is used only for transporting data from end-point to end-point. One benefit of this being that this results in a reduction in the computational overhead which compounds when using a service infrastructure in which service messages are inspected and processed by multiple nodes in the network core. The other benefit being that service developers gain the same flexibility in service design that the Internet provides its service developers.

Application Layer Signalling (ALS), which is formally introduced in chapter 2, is the defining feature of the proposed service environment. Its purpose is to support the idea presented above, by providing the infrastructure over which terminals and ASs communicate with each other without using CS signalling. At a technical level, ALS is not new. The Internet has always operated using it in some form. However, the Internet has no bearer connectivity infrastructure supporting it, so there is no purpose in differentiating its ALS from any other type of signalling, as ALS is its sole mechanism for signalling. In the telecoms world there is merit to differentiating ALS from CS signalling as it allows us to make a clear distinction: ALS is for service signalling, allowing application logic in distributed nodes to communicate with each other; CS signalling for setting up bearer connections.

With ALS providing the signalling mechanism for application layer focused service development, a software framework needs to be developed to provide programmers with an interface into ALS and also to support the simplified development of service logic. We can specify that ALS is presented using remote objects, whereby application logic in one end-point wishing to invoke some functionality of a second end-point will do so by remotely calling a method on an object existing in the second end-point. This then constrains the software development framework to being built up using OO. All functions of the framework, including the setting up of bearer connections, are presented as objects. Chapter 3 aims to build up a software framework using OO to convey various aspects common to all service development environments, such as reusable building blocks, user profiles and message passing between terminals and ASs.

Besides the defining features of ALS and OO, the proposed framework also takes a unique approach to the setup, maintenance and teardown of bearer connections which is conceptually different in the proposed framework to that of the legacy approach. A bearer connection is seen as secondary to service logic – not in importance, but rather from the fact that a request for a bearer connection is always made from a service. Even a simple two party call that requires no service logic, is setup when a *Two Party Call* service requests that a call be setup between two terminals. This idea was advanced in [19] and termed *"3rd-party call initiation"* – the intention being that a terminal will make a request to an AS (the *3rd-party* in this case) to setup a call between itself and a second terminal. Thus the AS, and not the originating terminal, makes the call initiation request to the bearer network to setup the call. [19] also quantified the extent to which this kind of call initiation overcomes the heavy signalling overhead which impedes the operation of modern telecoms infrastructures. Whilst the aim of this report is not to quantify the efficiency of the proposed service environment, it is important to note the benefits, besides service development simplicity, which are gained in its implementation.

1.4 Conclusion

We have thus identified the shortcomings of the current service environment and have outlined the crucial elements of the proposed service development environment which attempts to correct these shortcomings. We close this introductory chapter by summarising the main points which determine the features of the proposed service development environment:

1. The Internet does complex services better.
2. Telecoms does bearer connectivity better.
3. Current service platforms use the bearer network for relaying messages between end-points. For example CSCF entities in the IMS bearer network inspect and process all service messages and are involved in communication between ASs and terminals.
4. Instead of this, rather use the Internet approach and move all service logic complexity to the edge (i.e. no service message processing occurs in core network).
5. Do not replace current bearer network connectivity infrastructures (such as is done in the Internet with voice over IP (VoIP)). Telecoms is best at bearer connectivity so we can continue to use its current infrastructure for bearer connections. However, bearer connectivity becomes secondary to services in that it gets called on by service logic (as opposed to vice versa).
6. Everything is therefore initiated in the application layer (even bearer connections).

The process of moving service development and operation into the application layer is made up of three important steps. These steps are detailed throughout this research report, not specifically in the order presented here:

1. Abstract out bearer network functionality from ASs since from the point of view of services, bearer connectivity is secondary to service logic.
2. Remove processing of service messages from the bearer network. Replace this with ALS to ensure that service logic is only executed in ASs and terminals.
3. Develop a standardised and generic software framework to support application layer focused service development.

These steps are not performed in isolation. We look to existing technologies to support the framework. For example, Parlay provides much of the inspiration for the abstraction of bearer connectivity. Moreover we are not aiming to replace an entire telecoms infrastructure. As we shall see, application layer service development can be integrated into existing telecommunications networks without the need for replacing software or hardware.

Having outlined the process of implementing our proposed service development environment, in the following chapters we will detail its requirements, the issues and technical challenges surrounding its implementation and the effects it may have on value-added service business all with the aim of producing a robust platform for modern, complex and competitive service development.

Chapter 2

Application Layer Signalling

The catalyst for all our desired improvements is ALS. The idea, whilst relatively simple in theory and which can be implemented with current technology, provides a completely new methodology for developing services. We arrive at the idea using two key design considerations, which we can explain by two layman statements:

- *"Modern Internet services provide lots of fantastic features which I couldn't get using only the plain old telephone. I can get them on my cellphone, but I am still accessing them through the Internet. I can also make cheap VoIP calls over the Internet, but the reliability and quality of these calls still aren't as good the old telephone system or even my cellular voice service."*
- *"I'm very happy with the service I get from my telephone and cellular providers. I know that if I phone someone his or her phone WILL ring (assuming they are not on another call). Also, I know that the voice quality will always be adequate for my needs. Its just a shame that I cannot expect much more than a few basic extra services like voicemail and call holding."*

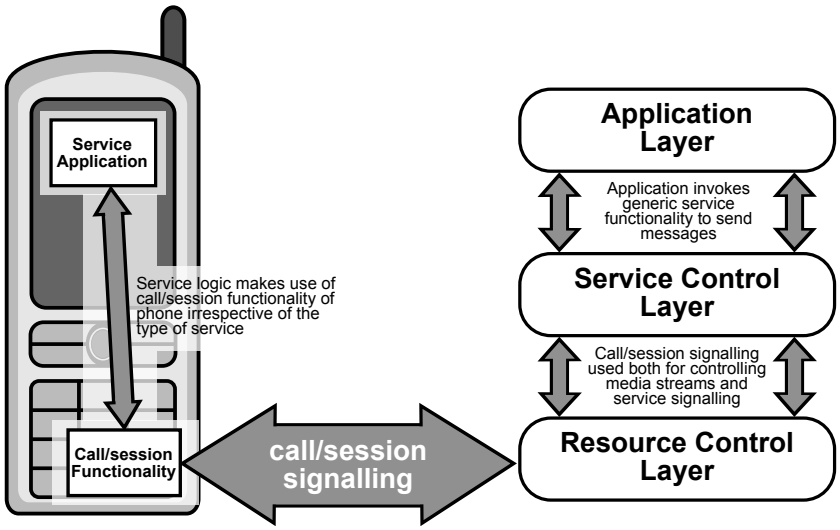
The two statements verbalise from different angles the perception of the main difference between telecoms and Internet services: telecoms networks guarantee quality of service for voice calls but only offer limited services whilst the Internet provides complex feature-rich services but cannot guarantee carrier-grade voice call quality and reliability. From this we see that the first of the two important design aspects which ALS incorporates is that having the terminal signal directly to the AS (as is done in the Internet world) allows for more complex services to be developed and easily deployed. The second is that a reliable modern telecoms network must, at least for simple voice calls, provide the same robustness and QoS levels which were experienced in legacy networks.

The integration of the Internet and telecoms to produce a modern network with Internet-like services and telecoms' QoS and charging and the ability to integrate these services has already been undertaken under the IMS umbrella. IMS deployments require a complete overhaul of current networks to replace all circuit-switched components with Internet Protocol (IP) packet-switched nodes. This resulting network provides bearer connectivity for which SIP provides CS signalling carried over an IP transport network. As is expounded upon in [13] and echoed by our layman with the comment about VoIP, a SIP-based core results in inefficient CS signalling. This overhauled network design also relies on SIP for service signalling, which is problematic as per reasons given in section 1.2.1. Of these two issues surrounding SIP, we will not cover its use for CS signalling. It is in its use for service signalling between terminals and ASs where we focus attention in order to introduce ALS as an effective alternative.

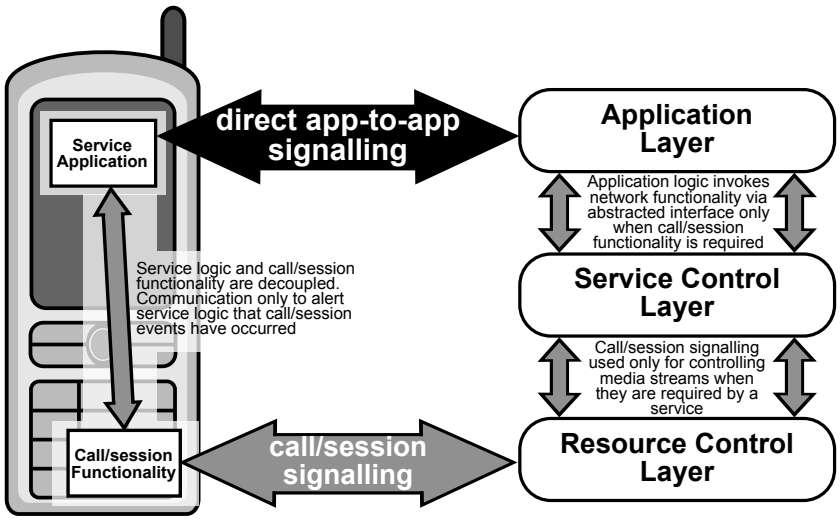
2.1 Direct Application-to-Application Signalling

Distributed applications which rely on the signalling of terminals directly to the AS is not a new concept. The Internet has a plethora of services which are implemented in exactly this way – instant messaging, video streaming and online gaming are all services which involve a client application residing in one node signalling directly to a server application residing in a remote node. Internet services based on client-server architectures are ubiquitous. This is because the Internet is a best-effort network that only provides a means for transporting packets between connected nodes. Alone, it does not provide any complex services other than domain name to IP address resolution. Instead, it relies on its openness to host complex services provided exclusively by 3rd-parties (who, in the Internet world, is everybody other than the few global Internet administrative bodies), which effectively keeps complexity at its edges. Direct signalling between applications fits into this environment well, since the network has no intermediary nodes that process the application layer payload of packets.

In moving towards the incorporation of Internet-like services into the modern telecoms service environment, it would be prudent to follow Internet methodology where computational processing occurs at the network edge. Doing so would make use of the computing power that is available to ASs and modern end-user terminals. Unlike in legacy networks where end-user terminals had little or no computational power, it is now possible to shift some of the service logic to terminals which have



(a) without ALS



(b) with ALS

Figure 2.1: The operation of services without and with ALS in place. Layering system taken from top three layers of the NGN framework in [8, pp 56]

adequate computational power to both execute complex service logic and perform the signalling required to support this service logic.

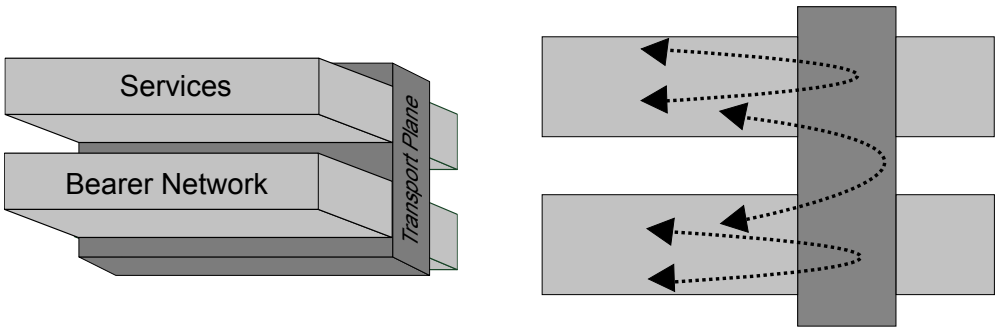
In the telecoms world, direct application-to-application signalling is a much more explicit concept, compared to that of the Internet, as it contrasts with traditional execution of service logic in the core network. It also contrasts with the more modern IMS approach to services which uses bearer network signalling as shown in figure 2.1(a) and consists of intermediate nodes which are involved in the service logic. Conversely, where direct application signalling is used, as is depicted in 2.1(b), a terminal signals directly to an AS for the majority of a service session, and bearer

network functionality is only invoked when a voice call (or any other media stream) is required. That is not too say the bearer network loses its importance. Telephony is the primary reason for a telecoms network and is the main source of a telco's revenue from customers. The modern bearer network is designed for bearing media streams from end-user to end-user, and as such, we do not wish to relieve it of its purpose – something that would anyway be difficult to do, since telcos invest a lot of money into ensuring that their bearer networks have high reliability and quality. The intention, rather, is to allow our proposed service environment to interface with existing bearer networks and treat the connectivity they provide as reusable functionality for services.

2.2 Separating the Bearer Network from Services

To further the discussion on direct application-to-application signalling we need to decouple conceptually the ideas of the bearer network and of services. The *bearer network* is made up of those entities which control calls (setup, maintenance, tear-down and handling of all events linked to calls), including entities that maintain call context and call state and provide the means for modifying calls (such as forwarding and call hold). Further it includes those entities which control logical association between two or more end-points in a call and ensure the availability of media streams between end-points. All this functionality is described in a combination of the *call control functional layer* and *bearer control functional layer* in the Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) specifications [20]. We call all signalling that occurs between these bearer network entities *"call/session signalling"*. Ideally we would not want services to use this signalling, or to even have knowledge of it, especially since it has been engineered specifically to support reliable bearer connectivity, and not to support service communication. Instead, all CS signalling should be abstracted away from services and accessed via an API.

One should not confuse the functions of the bearer network and that of the *transport network*. In comparison to the bearer network functions mentioned above, the transport network is responsible for transporting streams, signals or data (collectively called *information* in this case) from one end-point to another. The transport network does not consider the content of the information which it conveys (the *payload*), its purpose is just to ensure that the information arrives at its intended destination. An IP transport network, for example, is responsible for packets' correct



(a) (b)

Figure 2.2: The transport plane shown in (a), with which services and the bearer network, viewed as layers, intersect. The cross-section in (b) shows how all signalling (shown with dotted arrows), be it between or within layers, always traverses the transport network.

traversal of routers from one subnet to another. Similarly, in classical circuit-switched networks the actual voice circuits through various switches formed the transport network. Services, which we will discuss below, and the bearer network both make use of a transport network, whether it is shared between them as in the case of the IMS or if each has its own as is the case with the IN. For the remainder of this research report it is necessary to envision a transport plane with which both services and the bearer network separately intersect, as shown in figure 2.2(a). The technology implementation of the transport network will be assumed to be in place and its details will not be covered. The CS signals used by the bearer network and the service signals used by services, which we are about to discuss traverse this transport plane in the manner depicted in figure 2.2(b).

While the bearer network provides the infrastructure for operation of telephony, *services* incorporate all the value-added features for which customers pay additionally and that can be viewed as supplementary to the telco's core task of providing reliable bearer connectivity. In the modern telecoms network, the types of services are varied and require an adequately complex infrastructure to support a wide range of functionality. Services are also less tied to the telco domain and in the modern approach they are programmed in a software development environment that caters to the needs of a wide range of programmers who do not necessarily have a background in telecoms. Many services, such as instant messaging, also do not require the

support of the bearer network in their operation which is an important aspect of the case for ALS.

The boundary between services and bearer networks is formed when we note that the functions of the bearer network are associated with the traditional telephony oriented telecoms network whereas rich, modern services are associated with the Internet and IT views of software. Since they can be seen as very conceptually different, the level of abstraction of the bearer network to services is important if their decoupling is to be beneficial. The abstraction cannot be at a very low level, catering only for individual protocols such as in the case of an API into SIP which is presented to SIP servlets. Low level abstractions result in APIs which have function calls representing individual protocol messages thus increasing coupling to specific CS signalling. Rather, the abstraction should group units of functionality such that it can be generic and able to be mapped to multiple bearer network architectures. It should also remove the requirement that a service have knowledge of the bearer network structure. This kind of abstraction is achieved, for example, by Parlay and JAIN. Whatever technologies are used, we need to ensure that they enable services to be decoupled from the bearer network to allow for proper implementation of ALS.

2.3 Application Layer in a Technology Neutral Framework

Various telecoms standards consist of an application layer at the top of a layered architecture. However, the definitions of these application layers are dependent on context and on lower layers, resulting in different definitions. The application layer being discussed is that of the technology neutral NGN framework (from hereon referred to as the *"NGN framework"*) developed in [8, ch 2]. We use the layering system from this framework (shown in figure 2.3) to aid in differentiating between services and applications and to provide further context to ALS.

The NGN framework presents a system of layers, domains and planes which facilitate the analysis of complex telecoms systems. Figure 2.3 identifies the top two layers as the focus of the work in this research report. Of these, the application layer sits at the top of the layering system and encapsulates the logic of services. In the NGN, service logic is implemented in software applications developed using traditional IT techniques. This layer is supported by the Service Control Functionality (SCF) layer which provides an interface into the Resource Control and Management Function

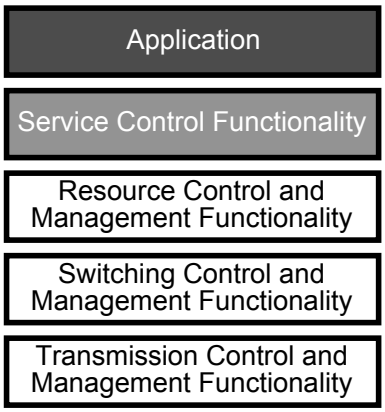


Figure 2.3: Layers of the NGN framework. Filled in layers are those which form the context of this research report.

(RCMF) layer of the bearer network. Section 2.2 explains why this interface should be abstracted adequately (ideally with an open standard API) so that application logic can be implemented irrespective of the bearer network technology.

All service logic existing both in ASs and in terminals is bound in the application layer. Signalling between these entities should not traverse layers – that is to say that no entities from lower layers should process signals travelling between application layer entities. This connects with and further enhances the reasoning behind the problem of incorrect usage of CS signalling laid out in section 1.2.1. We therefore use the NGN framework as a reference for ensuring that the final design for a service development environment based on ALS does not violate the requirement of keeping all inter-application signalling at the application layer.

2.4 Ideal Application Layer Signalling

In formalising ALS we need to visualise a target environment in which service development and operation is greatly enhanced by direct communication between terminals and ASs. In this section we identify the main components required to make ALS into an attractive mechanism for the development of modern services.

2.4.1 Service Operation Centralised in the Application Layer

Besides changing the method of service communication, ALS also affects the overall structure of services and the manner in which they are invoked. The ideal implement-

ation of ALS would result in a full inversion of the execution of services by shifting the locus of service control into the application layer. From the point of view of services, the bearer network would take on a supporting role, being called upon only when a service requires media streams to be setup between two or more terminals. This would mean that sessions would always be initiated in the application layer, which differs from the traditional approach where services are secondary to the processing of a call. The result being that service logic is only called on in response to the occurrence of specific events or criteria being met during the call. In this new service-centric approach even basic calls are initiated via the application layer, emphasising the concept of 3rd-party call initiation developed in [19]. This means that a basic call is treated as a service such that the process of invoking one would be initiated over ALS. Section 2.5.3 will cover how this necessitates the relocation of the overall view of bearer connectivity into the application layer.

2.4.2 Signalling Protocol and Structure

Our proposal for ALS has so far only described the route that signals take when they travel between terminals and ASs. So far no detail has been given on the actual structure of these signals or protocol which they would follow. Since the reduction of the reliance on CS signalling is one of the targets that ALS is aimed at, it is important that the signalling system be designed to be adequately lightweight to not result in the same overheads experienced by service architectures which rely on CS signalling. Further, the ALS protocol should be flexible enough to cater for any complex service. As will be seen in chapter 3, the software framework which will support ALS will be based on OO and will expose services to terminals as software objects. Similarly, terminals will expose their functionality as objects. The signalling should allow for both terminals and ASs to call methods on each other's exposed objects remotely, meaning that signals should contain the desired service object, the method to be called and any arguments to be passed into the method call. This is reminiscent of the CORBA and RMI mechanisms for accessing the functionality of remote objects, but as they are both known to be demanding systems, ALS calls for a much more efficient communications oriented system for remote object interaction – an idea taken further in section 3.2.4.

2.4.3 Appropriate Bearer Network Abstraction

We have up to this point emphasised the importance of the abstraction between the service environment and the facilities provided by the underlying bearer network. In section 1.2.1 CS signalling was identified as being an incorrect mechanism for interfacing between services and the bearer network. Then in section 2.2 we outlined the need to set the correct level of abstraction, for which the solution is open generic abstraction.

Bearer network abstraction is a topic on its own and has been part of the service environment for many years. Whilst it is technically separate from the proposal for ALS, it is closely linked in that it too acts towards centralising service logic in the application layer. To gain the full benefits of ALS, bearer network abstraction needs to be in place. Thus the software framework which will be developed in chapter 3 to support ALS will emphasise abstraction to ensure that software programmers are fully insulated from the detailed workings of underlying telecoms technologies, effectively and correctly confining them to software development in the application layer.

2.5 Important Clarifying Concepts

The changes which result from ALS call for various concepts to be clearly defined to enable correct implementation. These include the description of services based on the location of their invocation, the locus of a call's control and view and the types of signals which can be used. Many of the clarifying concepts covered here are taken from previous work on the same topic in [19, ch 3].

2.5.1 Atomic and Composite Services and Orchestration

Up to this point the term *service* has been used loosely. To avoid ambiguity as to what we mean when we refer to *service logic*, we define *atomic services* and *composite services*. An atomic service is one which performs very specific functions and does not reuse functionality provided by other services. For example, if an atomic service is acting as an interface into the bearer network, then upon execution of one of its functions a predefined set of signals traverse the bearer network according to the required functionality. Based on this, it is possible to view Parlay-X web services

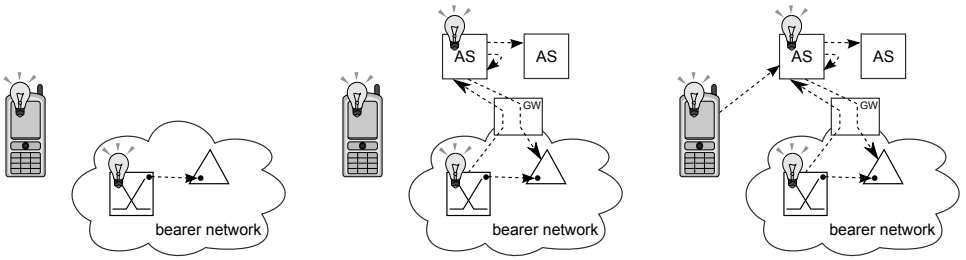
as atomic services. Atomic services are not used directly other than in composite services, where their atomic functionality is reused to build up complex functionality.

Where we have simply been discussing *services*, we have been specifically referring to composite services and the *orchestration* thereof. Orchestration is the control of the logical process flow of a composite service as it executes a specified function by reusing the functionality of atomic services and implementing its own logic. In the target service environment of this research report, all composite services exist in and are invoked in the application layer. This differs from the IN in which composite services (the target services in each capability set) resided in the bearer network and were built up from atomic Service Independent Building Blocks (SIB) which also existed in the bearer network.

In the proposed application layer approach, all composite services exist in ASs where orchestration also occurs. The composite services reuse atomic services which abstract both bearer network functionality and higher-level reusable logic existing in ASs. These atomic services will take on the term Reusable Building Blocks (RBB) when they are discussed in the context of the supporting service framework discussed in chapter 3. Composite services will continue to be referred to simply as *services* and *service logic* will denote service orchestration.

2.5.2 Types of Service Initiation and Invocation

The mechanisms with which service logic is called on can be categorised under two headings: the *initiation* of a service refers to the location where it is deemed necessary to request the service's execution; the *invocation* of a service is the actual process of calling on service logic. The changes to the network which precede new types of initiation and invocation are shown in figure 2.4. In the legacy network, before telcos opened up the functionality of the bearer network to 3rd-party developers, services could be initiated in the terminal or in the bearer network. They could, however, only be invoked by the bearer network using triggers which, when a specific condition was met, would execute service logic residing in SCPs in the bearer network. This is shown in figure 2.4(a). The opening up of the network resulted in the AS being the third location for initiation and the second source of invocation, as per figure 2.4(b). Service logic could now be hosted and initiated in ASs which in turn also allowed ASs to invoke their own service logic or service logic residing in other ASs or the bearer network.



(a) legacy (b) with open APIs (c) with ALS

Figure 2.4: Stages in the progression of service initiation and invocation. Lightbulbs show points of service initiation; dotted arrows show service invocation. The terminal is not shown as connected to the bearer network as the arrows only represent invocation. Also note the gateway (GW) entity required for enabling ASs to invoke bearer network services and vice versa.

The introduction of ALS adds a third type of invocation by providing terminals with the ability to invoke service logic directly, shown by the arrow from the terminal to the AS in figure 2.4(c). This is also aided by the fact that improved computational abilities of terminals allow for some service functionality to be shifted into them. This is a contrast to the legacy network, in which the initiation of a request for a service's functionality, such as for placing a call on hold, would result in the request being forwarded to a switch which in turn would invoke the service in an SCP on behalf of the terminal.

In the ideal environment all service logic is executed in the application layer. To that end ALS should result in a simplification of figure 2.4(c). If all service logic is located in the application layer, then there is no need for the application layer to invoke services in the bearer network (since service logic no longer exists in the bearer network) and similarly services are no longer initiated in the bearer network. Figure 2.5 represents this simplification which is also aided by relocating the view of bearer connectivity and its locus of control into the application layer which is discussed in section 2.5.3.

2.5.3 Bearer Connectivity Control and Monitoring

Section 2.4.1 presented the objective of inverting control of the entire telecoms service infrastructure. This is achieved when ALS is in place and the majority of service logic residing in ASs is either invoked from a terminal or from ASs themselves. Ideally, the bearer network only serves to provide bearer connectivity and is called

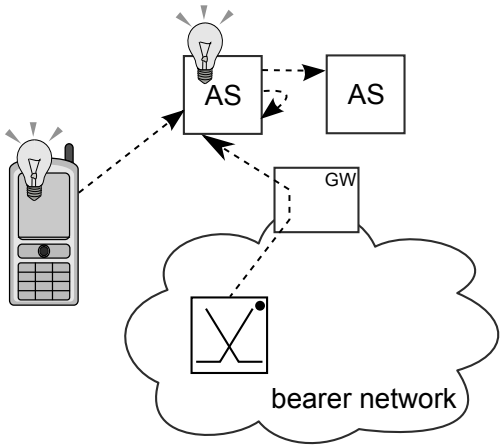


Figure 2.5: Simplified service initiation and invocation with ideal ALS. Lightbulbs show points of service initiation; dotted arrows show service invocation. Note that ASs no longer invoke service logic residing in the bearer network, and service logic is no longer initiated in the bearer network.

on only when required. The bearer network's functionality can then be treated as supplementary to applications and divided into atomic services which are exposed by generic call control abstractions. The result is that the application layer becomes the locus of service control.

Regardless of this, bearer connectivity remains telcos' main product offering especially as traditionally their main business is providing carrier grade telephony. Further, most telecoms services involve some form of telephony and thus require the support of the bearer network. Because of these two reasons, the issue then arises as to how bearer connectivity is monitored and controlled when services operate mainly in the application layer.

BCCM in the Application Layer

In a network configuration in which the bearer network is the primary means for call and service signalling, it is intuitive to have bearer connections both monitored and controlled from the bearer network. When service invocation and control is moved into the application layer as is the proposal for this research report, the location of bearer connectivity control and monitoring (BCCM) needs to be reconsidered. In designing an application layer focused network we have to determine whether it is feasible for BCCM to remain in the bearer network or if benefits can be gained from relocating it into the application layer. An important point weighing in favour of

shifting BCCM into the application layer is that in the proposed service environment all sessions are initiated and maintained in the application layer, to the extent that even a basic two party call is initiated in the application layer. This means that all bearer connections are associated to processes in the application layer and are effectively part of a service. In the ideal case of the proposed network design, all service logic is located in the application layer. Service programmers developing services which require bearer connectivity need to interact directly with a BCCM system. The logical approach would therefore be to have it reside in ASs.

Bearer Connectivity Control in the Application Layer

Moving both the monitoring and control of bearer connectivity can be seen as two separate, but connected issues. We have already discussed the tool which can enable bearer connectivity *control* in the application layer: abstraction using open and generic APIs into the bearer network. However, the main candidate API for our purposes, Parlay, is not completely in a form which would cater to IT software developers. This is because it has inadequate structure, does not implement software reuse and offers no guidance on application layer usage [21]. The approach we take is to further abstract Parlay's interfaces to produce an interface that is more IT-friendly. This is detailed when we outline the supporting application layer software framework in chapter 3. It is still important to acknowledge that Parlay capably performs the task of a generic interface, providing a robust starting point on which to build an application layer bearer connectivity control interface. We leave the topic here for further exploration in chapter 3.

Bearer Connectivity Monitoring in the Application Layer

Bearer connectivity *monitoring* in the application layer provides a newer challenge in the implementation of ALS. Bearer connectivity monitoring allows a service to determine whether it can invoke a particular operation in the bearer network without placing the bearer network into an invalid state and it also keeps a service updated of any changes to a bearer connection of which the service should be aware. In older network configurations in which application layer signalling is not formally in place we can differentiate between *network* and *application* views (we use the term *view* interchangeably with *monitoring*) of call processing where the network view is located in the bearer network and exists throughout the lifetime of a call and the application view is located in an application server and only lasts as long as it is

needed by a service [5, pp 53-54]. However these distinctions fall away when ALS is introduced. All service sessions are initiated and maintained in the application layer so an application view would also have to exist throughout the lifetime of a call, since in an ALS environment all calls are initiated in the application layer. From this we make the design choice of moving bearer connectivity monitoring completely into the application layer.

To cater for monitoring in the application layer, an AS should be equipped with models of all the bearer connections that have been started within the context of a service. The view associated to each model would always represent its associated bearer connection's current state. Before invoking an operation on a bearer connection, an application would first query that specific bearer connection's model to ascertain whether the operation is valid in the current context. Whilst it could still be possible to do this if bearer connection state monitoring was kept in the bearer network, this would be less efficient since in this case, a service application would have to send a query down to the bearer network before invoking any operation. Alternatively it would invoke an operation without any guarantee that it would execute successfully due to the possibility that the operation may put the bearer connection into an invalid state.

If the models of bearer connections are available in the application layer, services are offered a consistent view of the bearer network at all times. Further, these models, which are generic and abstracted, will complement the abstracted control of the bearer network discussed above – the argument from this being that if control is in the application layer, monitoring should be closely coupled to it and thus exist in the application layer too.

Bearer Connectivity Model

Together, control and monitoring are combined into a bearer connectivity model (in the literature this is referred to as a *call model*). The model forms part of a call control API, such as those presented by Parlay [22] and Java Call Control (JCC) [23]. The call control API which we will use to support our application layer centric service environment will integrate a finite state machine (FSM) originally designed to support a Parlay-X extended call control web service. This FSM is presented in [24] and shown in figure 2.6. Like that of other call models, this FSM shows the allowed transitions that can occur during the processing of a call. A model would disallow a call control operation if a particular transition would violate the

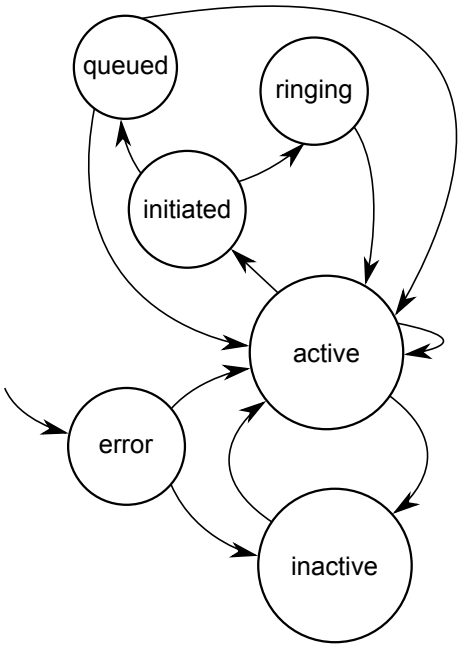


Figure 2.6: Bearer connectivity FSM

constraints set by the FSM.

The FSM in figure 2.6 is applicable to BCCM in the application layer as it was designed to provide state to a stateless web service interfaces into a bearer network. It therefore suits the decoupled and abstracted service environment of the supporting application layer framework to be detailed in chapter 3. Further, to support the simplified web service interface it was designed for, the model is much simpler than those used by the standardised, low-level and fine-grained call control APIs mentioned above. Simplification is a key goal in introducing ALS, making this FSM very appealing for use in the bearer connectivity model to be implemented by the framework.

Object-Orientation

The framework will be built up using OO and so too will the BCCM built into the framework. As shown by figure 2.7, BCCM lends itself well to being represented using OO [25] which provides extra motivation for it to be performed in the application layer[1]. The application layer which is a very software focused layer should ideally cater to IT methodologies and concepts. Thus OO's being very much an IT

[1]The OO model for a bearer connection that we actually use in examples in chapter 3 is much simpler than that of figure 2.7.

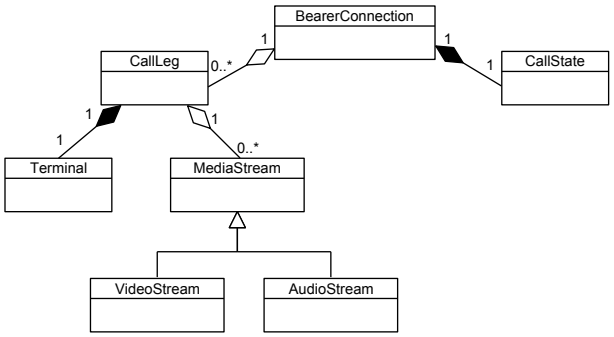


Figure 2.7: Example of BCCM represented using OO. The `CallState` object encapsulates the FSM in figure 2.6.

software concept means that since BCCM can be presented in an OO fashion it is virtually tailor-made for use in the application layer.

2.5.4 Signalling Requirements in the Application Layer

The technical requirements for direct signalling between terminals and application servers differs from those usually associated with CS signalling. The differences are revealed when considering the tasks of nodes involved in application logic and those involved in bearer connectivity. Obviously, ALS is designed to cater to the needs of distributing application logic. In the execution of this distributed logic, two applications signal to each other to request that the other perform some sort of computation, useful within the context of a service. Bearer network nodes signal to each other to discover a route for a media stream to traverse, to determine whether the stream is allowed to be established between two terminals and to maintain the stream. Therefore, bearer network nodes' computations are used to direct traffic based on QoS guarantees and AAA rules.

This distinction between bearer network CS signalling and ALS, highlights the fact that ALS is unique to the application layer and therefore needs to have a set of requirements defined based on its task of direct application to application signalling. Whilst this report does not fully detail a concrete ALS protocol, the requirements laid out below can be used to develop a protocol. Further, these requirements also affect the software framework which will support ALS (to be covered in chapter 3) and so in listing the requirements we also touch on the effects on the framework.

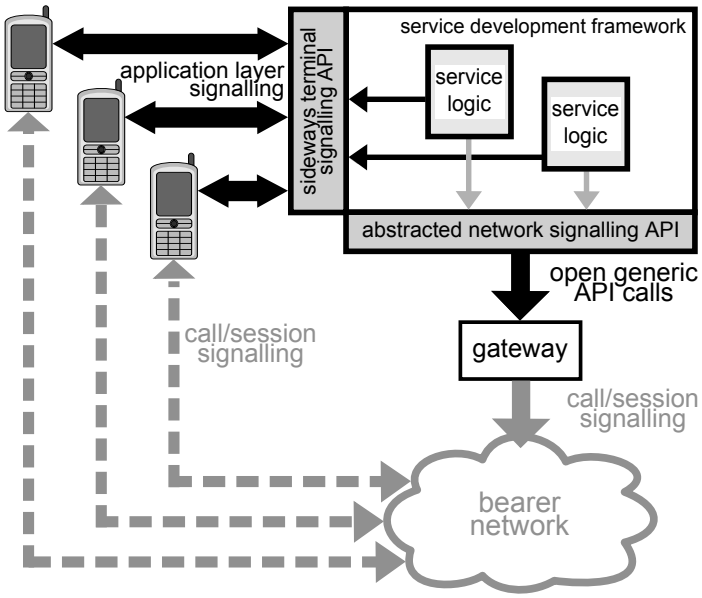


Figure 2.8: Sideways and downwards communication within the context of the framework. Note the clear distinction between the API for enabling communication with terminals and that for communications with the bearer network.

Sideways signalling

One characterising feature of ALS is that it is used exclusively for "sideways" communication. This means that it does not traverse layers, unlike CS signalling, which when used to support service functionality, acts in a "downwards" direction to provide communication between the bearer network and the gateways which provide services with access into the bearer network.

From figure 2.8 downwards signalling implies the interaction between two set systems, one of which relies on the other for supporting functionality. A Parlay gateway interfacing with an entity in the bearer network is an example. The important feature here is that these two entities stay associated to each other for a long period of time. Similarly, an AS will only be associated to a single Parlay gateway. On the other hand, also shown in figure 2.8, sideways communication is designed to allow communicating entities to interoperate with multiple separate entities. The relevant example here being ALS with which an AS interfaces simultaneously with multiple terminals.

The difference in downwards and sideways signalling also affects the way in which the software framework will present interfaces into the bearer network and into ALS. Most modern service environments present downwards interfaces, such as the various

RAs used by JSLEE [15]. Similarly, the OSE includes the *I2* interface which is a general name for any interface that is available to an *enabler* to access underlying network resources [18]. In section 3.3.1 the proposed software framework's RBBs are introduced, and they too perform a similar function. Downwards interfaces are those which provide the abstraction between layers that has been discussed in depth up to this point in the report. However, they are not used for communication within single layers.

Sideways signalling, which is constrained to a single layer, is not related to abstraction. Abstractions act to provide *isolation* between two dissimilar systems, where one of the systems needs to use the resources of another system in order to function. The aim of sideways signalling is to enable communication between two logical nodes. The two nodes can function independently of each other but have to *interoperate* to be useful. This can be likened to the difference between protocols and APIs discussed in [5, pp 24–27]. We cannot make the exact same clear distinction here because if the framework which is going to be developed in this chapter is going to be functional, then the protocol for enabling sideways communications is going to be presented as an OO API. Instead we rely on categorising communication as either sideways or downwards as a means of showing that the ALS software interface interacts with the software framework in a different way to that of downwards facing abstractions which will be encapsulated in RBBs. The software framework has to cater specifically to ALS as an integral part of the framework and not encapsulate it as an RBB.

Bidirectional Asynchronous Communication

Figure 2.8 shows that sideways communication, such as that of ALS, caters to a scenario in which one AS communicates with many terminals and in which these terminals send messages asynchronously to the AS and vice versa. This means that ALS has to support simultaneous receiving, handling and sending of messages. For this, the AS must also be able to initiate a message conversation by sending the first of a series of messages, and not only be able to respond to terminal requests.

This requirement of asynchronicity supports the simplification of the object-oriented message passing between distributed application layer entities. As we will see in section 3.2.4, the proposed ALS framework will use remote method calls as the format for message passing between ASs and terminals. Methods are capable of returning values, which calls for synchronous behaviour. However, supporting this

in a distributed environment adds complexity by requiring a means for enabling a node to identify the message containing the return value associated to a remote method it called. Instead, by specifying asynchronous functionality, we simplify the requirements of ALS by removing the ability to make synchronous method calls. This is covered further in section 3.2.4.

The bidirectional and asynchronous behaviour also affects the framework by forcing it to provide an interface both for sending and processing messages to and from multiple terminals simultaneously. The software mechanisms for this are presented in chapter 3.

Statefulness

State is a very important aspect of any telecoms system. We examined its importance in relation to BCCM in section 2.5.3. ALS is therefore specified to be stateful, in the sense that the action an application layer node takes in response to receiving a message depends on the state of that node. For example, if a terminal, say terminal C, sends a message to an AS requesting that a voice call be setup between itself and terminal A, but terminal A is already in a call with terminal B, the request will fail, whereas it would not if terminal A was available for calls. More formally, in the context of calls, messages that request a state transition unspecified by the FSM of figure 2.6 will fail. The idea being that certain messages passing between terminals and ASs are only relevant to certain states. The requirement of statefulness therefore mainly comes about by the fact that the ALS is affected by a dynamic bearer network whose state is constantly changing.

Statefulness places the onus on application logic to handle messages which arrive out of context (i.e. not appropriate to the current state). This adds complexity to ALS, however it is unavoidable given the telecoms context of the application layer. There are ways to simplify the programmer's task of handling state. For example, a mechanism involving programming exceptions can force a programmer to handle these exceptions if an attempt to perform particular operations fail. In this case, the AS can be programmed to reply to an out of context message with a relevant error which the terminal can handle. These exceptions can be built into any BCCM-supporting RBB, an implementation issue which we do not delve into further in this research report.

Types of Signals

There are five types of signals that can be sent from terminal to AS and vice versa:

1. *instruction*: a request that the receiver perform a function specified by the message.
2. *information request*: a request for information, such as the state of another terminal.
3. *notification request*: a request for the receiver to notify the sender of a particular event.
4. *acknowledgement*: confirms that a message has been received.
5. *return*: in response to a message that needs some information to be returned, this message contains that information. Note that this message is a separate asynchronous message as per the asynchronous message transmission requirement detailed above.

An ALS implementation does not have to cater explicitly to each of these messages, since it is up to the service developer to specify the different methods that can be called remotely. The list above is simply a categorisation that identifies the uses of ALS.

2.6 Conclusion

We have presented ALS as a novel approach to service development which ensures that service logic is correctly constrained to the application layer. "Sideways" ALS does not traverse layers, and so separates the bearer network from service logic distribution. This leaves terminals and ASs as the only entities that process service logic. In this situation, services are no longer reliant on the underlying bearer network infrastructure, making for an IT-friendly service development environment in which software programmers do not need knowledge of telecoms protocols to develop services which are distributed over a telecoms network.

ALS is not a protocol, and in fact at no point in this research report do we formally define a protocol. Rather, it is a methodology for allowing service logic end-points

to communicate in a similar way to that of the Internet whose infrastructure, which relies on keeping complexity at the network edge, is better suited to modern complex services. ALS is the crucial component of an application layer focused service development environment and from a software perspective is not treated as a reusable adapter to a signalling system (like a JSLEE RA).

We do not regard ALS as a replacement for bearer network architectures. The bearer network of any telecoms infrastructure is designed to facilitate high volumes of high quality calls – the primary source of revenue and hence the primary focus of any telco. Many bearer networks are in place, and have been for many years and telcos have made large investments into ensuring that they can offer a high grade and quality of service. What ALS attempts to achieve is to be complementary to the bearer network by removing the responsibility of service message communication from the bearer network and reposition the locus of service development in the application layer. To do this, all service sessions are initiated in the application layer and are kept there until a bearer stream is required by the session (which might not always be the case for some services). This focus on the application layer is performed to the extent of having BCCM in the application layer.

ALS also acts to simplify service development and calls for a new service development environment to be designed. The following chapter covers the requirements and design of a software framework which supports ALS and in doing so aims to simplify the process of implementing modern distributed telecoms services.

Chapter 3

Supporting Software Framework

ALS leads to a service environment which is a large step away from the traditional telco view of services. This is not just in the way that terminals communicate with service logic. ALS conceptually changes the way that services are developed and deployed. Service logic can be completely separate from the underlying bearer network, ideally being decoupled to the extent that it can be reused over any infrastructure.

Section 1.2.2 named a few of the supporting technologies for modern services. These are mature, standardised technologies which have been widely deployed to support telecoms services. If ALS is to be implemented, it would be sensible to look to these technologies to provide the necessary platform to support this new service paradigm. In the face of there being no service development environment which explicitly supports ALS, this chapter lays out the requirements of a supporting software framework (which will be referred to from hereon simply as *the framework*. The requirements are given in the form of a technology-neutral UML design.

3.1 The need for a Formalised Application Layer Signalling Framework

Whilst ALS provides the mechanism for a seemingly efficient and intuitive service development environment, it is not without its idiosyncrasies. Chapter 2 characterised ALS in terms of its service invocation and BCCM and how they differ from those of a traditional CS signalling based service infrastructure. Another main difference also discussed was that of the lifetime of AS processes and that in an ALS environment's service session, these processes exist for the length of the session — contrasting

with the traditional invocation of AS service logic only when required during a session. These differences highlight the fact that existing service architectures or frameworks may not cater to the needs of an ALS implementation. This is because they have been designed to host traditional telecoms services which are only executed to supplement the bearer network's handling of a call. Nevertheless, some existing technologies may actually be suitable, so there is a need to work out the requirements that implementations would have to meet. With these requirements in place, it would then be possible to map existing technologies to them and also to use these requirements as a guide for the development of new ALS supporting software frameworks and architectures.

3.1.1 Formalising The Level of Abstraction

ALS relies very heavily on service logic being correctly abstracted from bearer network operation. So far, three levels of abstraction have been covered. The first and lowest being that of simply abstracting a single CS signalling protocol into an API, such as is done with SIP servlets. At this level of abstraction a service programmer requires detailed knowledge of telecoms concepts.

The second level is open generic abstraction like that of Parlay. At this level of abstraction, programmers make use of a uniform interface which can be mapped to different bearer network implementations. Despite a service programmer not needing expertise in CS signalling protocols at this level of abstraction, one still has very fine grained control of bearer network resources, which to a lesser extent, requires that a service programmer be equipped with detailed telecoms knowledge. This follows on from the point made in section 2.5.3 about Parlay not offering guidance on application layer usage. This point also leads on to the third level of abstraction, which will be built into the framework and which we label as IT-friendly abstraction.

The idea of this level of abstraction is taken from the use of web services in telecoms. Looking to Parlay-X as an example, we see that it was created to further abstract out the fine grained call control offered by Parlay. Web services are "inherently simple and cannot express detailed network capability" [8, pp 369], yet for many cases they offer just the right level of control to programmers who come from an IT background and lack detailed telecoms knowledge. Web services provide the inspiration for this higher level of control, however we will not explicitly take the web service approach, instead replacing it with a more technology neutral and still highly abstracted OO interface into the bearer network.

This level of abstraction is applicable to the framework since the goal is to have the framework completely encapsulate service logic in the application layer. The justification for this comes from the fact the framework must be accessible to software programmers who are used to developing the computing applications associated with modern services which reside in the application layer. These programmers do not necessarily have a telecoms background and so for the sake of the telco and for their own it is more suitable to have them as far separated from network functionality as possible. Such an abstracted interface would make the framework a lot more appealing to wider range of programmers for the reason advanced in [26]: for each level of increased abstraction, the number of programmers with the skills required to program at that level increases by multiple orders of magnitude.

The principle here is that ALS is the catalyst for moving all service logic into the application layer. This, in turn, calls for service logic to be developed in a highly decoupled software development environment, meaning high abstraction from the bearer network to ensure that service logic does not cross the boundaries of the application layer. Despite abstraction from lower layers being a separate concept from ALS, it is nevertheless a necessary part of the framework.

3.1.2 Meeting The Needs of Direct App-to-App Signalling

The effect of ALS on service development is not a subtle one. It results in a unique requirement of a supporting mechanism for keeping signalling in the application layer (instead of adaptation to lower layer signalling, which is already implemented in various mature technologies). It also changes the way services are developed, as service logic is distributed between ASs and terminals and is executed only on them. Since they are both equally important in the computations involved in service logic there is a need to share the results of these computations easily. ALS must be an integral part of the framework whilst keeping the signalling mechanisms simple. This leads to some technical requirements such as multithreading, message context and dynamic method invocation (all covered in section 3.2).

It is clear that ALS needs a specialised software environment as no exisiting telecoms service development environments explicitly cater towards application logic in terminals and ASs communicating directly with each other to keep all signalling in the application layer. An ALS-based development environment is unique in that it must both cater for “downwards” bearer network abstraction and for the “sideways” signalling we introduced in section 2.5.4. This chapter therefore shows

how a software framework must be developed to cater specifically towards ALS.

3.1.3 Supporting Application Layer BCCM

In section 2.4.1 we advanced the idea of inversion of control, where in an ALS environment, bearer network connectivity is secondary to the operation of a service; in section 2.5.2 we discussed the reconfiguration of service invocation and initiation to a point where service initiation can only occur in the application layer and only ASs are able to receive service invocation requests. These two points together separate service control from bearer network entities in the SCF and RCMF layers and raise it into the application layer. In section 2.5.3 we went on to explain how having the application layer as the focal point of service development and operation would benefit from BCCM being located directly in the application layer as opposed to being presented as an API.

As was mentioned when BCCM was introduced, OO can intuitively represent bearer connections in a manner that is accessible to programmers who have limited CS signalling knowledge. This kind of BCCM demands a service development environment that is not based on various modules that adapt to different lower-level bearer network technologies. It instead calls for a interface into it that allows for the manipulation of a model that is a general representation of the state of the bearer network from a higher, less detailed viewpoint that is suitable for use by application logic that has no need for CS details.

We have already specified our framework as being designed for the application layer. Based on the positioning of BCCM in this layer, the framework is necessary to provision this higher level interface to service developers. This also fits in with the IT-friendly abstraction introduced above.

The various reasons given for the necessity of a dedicated ALS framework all centre around the encapsulation of service logic in the application layer and sideways signalling between service logic nodes. Having explained these reasons, we now move on to specifying various requirements to ensure a simple ALS implementation.

3.2 Guiding Principles and Design Choices

Beyond the main justifications for a formal ALS framework which were examined in section 3.1, the framework design will follow other important principles along with specific design choices. We will highlight them in this section before showing how they are used in and affect the design of the various framework components detailed in section 3.3.

3.2.1 Object-Orientation

OO is a powerful tool for reusable and extensible software design, and its benefits are useful to telecoms software development as well. Despite this, older procedural programming languages have been used for many years in telecoms systems and are seen as more computationally efficient. However, this fact rests with the quality of the software programmer — producing robust and efficient software at the lower level associated with procedural programming is a costly and time consuming task that can only be undertaken by skilled programmers. OO, on the other hand, opens up the door to a wider skill base and having a telecoms service environment based on it, enables programmers to make use of the many OO tools, programming languages and methodologies which are available to the IT industry.

The reasons for building up a service development environment using OO are manifold and are covered in [27]. We will give some reasons here which are more relevant to the ALS framework.

Expressive Power

The *expressive power* of a programming language refers to its ability to be able to describe in a human understandable form the solution to a problem that the software is attempting to solve. In the case of telecoms and ALS, the language needs to cater to solving various representational problems. The most pertinent problem is how would BCCM be represented in the application layer. Another is how to represent ALS so that a programmer has easy access to its functionality.

Regarding the first problem, section 2.5.3 already pointed out the applicability of OO to BCCM with an example in figure 2.7 of how it could be represented. The other problem can be handled by having terminals represented as objects. By

calling a `sendMessageToTerminal()` using a `Terminal` object, the framework will send a message to a physical terminal which this object represents. Further, a management object containing references to all terminal objects can allow terminals to be looked up according to specific attributes such as a username or telephone number associated with the terminal.

OO can also provide a strong representation of the overall framework. The various components of the framework along with their boundaries can be clearly defined and so too can their relationships. Structuring the framework in an easily understandable form makes the framework more accessible since instructions on the use of each of its components can be more easily contextualised and conveyed. This is more natural than the procedural approach which groups functions into reusable libraries which have much weaker relationships and structure in context of the problem being solved. We will see how OO accomplishes the task of representing the requirements of the framework in section 3.3.

Encapsulation

Abstraction has come up as a fundamental part of the supporting infrastructure for ALS. *Encapsulation* forms a critical part of this, since the objective is to hide telecoms complexity from the programmer so that no detailed telecoms implementation issues leak into the application layer. Complex implementation details can be encapsulated within objects so that a programmer can utilise an abstracted interface. For example, a `makeCall()` method of a `CallManager` object would encapsulate all the Parlay method calls required to setup a bearer connection, thereby simplifying a programmer's interaction with underlying network functionality by encapsulating multiple complex calls into a single method. The encapsulation which OO achieves is, in a sense, the main means by which bearer network abstraction is implemented.

Encapsulation also enables the reusability of service logic. Since bearer network implementation details are encapsulated within the framework's objects, any service logic using these objects can function irrespective of the encapsulated technology's specific bearer network logic — once again pointing towards one of the ideals that abstraction tries to achieve.

Object Distribution as a Protocol

Telecoms services are inherently distributed. This fact is very relevant to ALS in that the framework will rely on having some service logic reside in terminals. Technologies like RMI, CORBA and Distributed Component Object Model (DCOM), have shown the usefulness of having application logic distributed as objects in multiple computing nodes. Whilst not a defining feature of ALS, its signalling protocol can be structured using OO.

As will be shown, within the framework services will be presented as objects with public methods that can be called from other services or from terminals. Since remote terminals will see these service objects as exactly what they are — objects with methods — they will be able to interact with them using OO message passing. OO will therefore also serve the framework as the protocol between terminals and ASs. This will take the form of allowing a terminal or AS to call a method on a remote service object, have this call marshalled into a form which can be carried by the transport network, and then at the destination the message will be unmarshalled so that the local method on the desired service object can be called. This sequence of events generally describes the operation of all the distributed object technologies listed above. The difference being that the above technologies have overheads which are not suitable for use by lower power terminals. Thus we will look to reuse their concepts but in a simplified manner.

3.2.2 TINA

The use of OO in telecoms is not a new concept. Amongst its other accomplishments, the Telecommunication Information Networking Architecture (TINA) initiative brought to light, emphasised and formalised OO for service development.

A legacy approach to the modern telecoms environment, TINA provided a basis for future telecoms service technologies. It was made up standardised architectures which provided guidance on the development of services and the composition of transport networks to cater to the distribution of the logic of these services. We briefly mention it here to show how ALS follows on from the TINA vision of modern services and also formalises a novel way in which TINA's concepts can be realised.

TINA was never fully accepted in the telecoms industry as a standard, however its four main principles define modern telecoms services [28] and support the concept

of ALS:

1. object-oriented analysis and design
2. distribution
3. decoupling of software components
4. separation of concerns

The applicability of the first three of these principles to ALS have been covered. Contextualised within the ALS framework, the last concerns the need to have separation between the different components of a telecoms network. ALS enhances this by providing a means for creating a clear separation between services and bearer network resources.

As the framework's details are laid out in the remainder of this chapter, TINA's relevance will become more apparent. It is important to keep TINA in mind when discussing ALS as it formalised a lot of what ALS aims to accomplish.

3.2.3 AAA

AAA is a critical part of any service infrastructure. The framework should provide mechanisms for this to implemented. The manner in which they are implemented is only covered very basically in this research report. We only indicate the points where AAA functionality appears without detailing the implementation. This section covers the AAA functionality that the framework needs to cater for and section 3.3, which describes the framework, presents a few of the technical details.

Authentication

All terminals connecting to an AS have to authenticate themselves to ensure that they actually do have access rights to the AS and then to be associated to the AS in a way which ensures that they can be identified for all service requests.

When a terminal first connects to an AS, it is challenged for credentials that uniquely identify it and for a password to authenticate it properly. The technical security details (such as encryption) are not covered, but this report does indicate how the

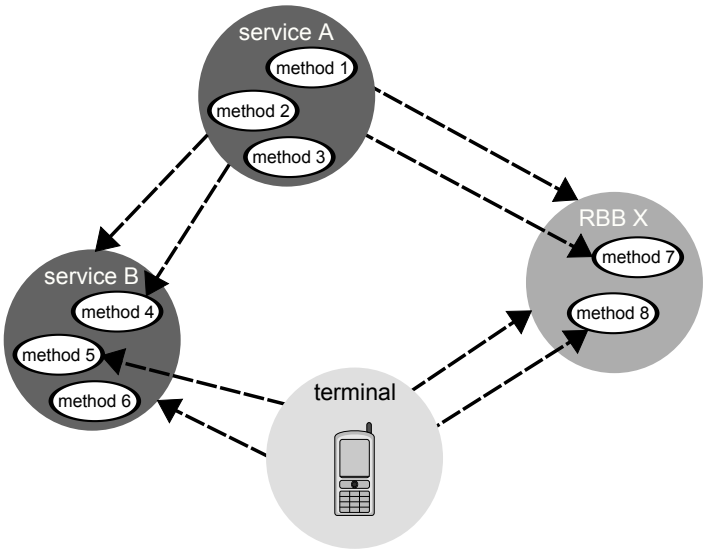


Figure 3.1: Access restriction types. Arrows show locations where access restrictions can be enforced.

framework only allows a terminal to be associated once it is correctly associated. In fact, the credentials provided during the login phase are used to identify the terminal during service sessions.

Inter-entity Access Control

From the perspective of the framework there are three types of components that cooperate in service logic: terminals, services and RBBs. For the purpose of this discussion we will provide a general term for these components: *entity*. Since entities are presented using OO, access between them can be restricted on a per-object and per-method basis. On a per-object basis, access is granted to all the methods of an object; on a per-method basis, access can be granted to individual methods.

Figure 3.1 shows the types of access control that can be enforced. Terminals may require access rights to invoke functionality of service objects or their methods. Access control can also enforce restrictions on terminals accessing services which use specific RBBs or their methods. Further, services may have to gain access rights to use RBBs or other services or their methods. As a clarifying point, control on RBBs accessing functionality outside of themselves is not required, since RBBs cannot invoke service methods or other RBBs' methods.

Whilst not all of these types of access restrictions have to be implemented, the

telco (or any other host of the framework) has a lot of flexibility in the type of access that can be granted. This flexibility gives the telco control over the way that both service developers and end-users interact with its resources. The service-to-service, and service-to-RBB restrictions allow the telco to limit service developers by specifying what underlying bearer functionality or existing service logic they can reuse in their own service applications. The terminal-to-service and terminal-to-RBB restrictions can ensure that users have to register their terminals for specific service functionality. This has to be supplemented with authentication functionality discussed above, so that terminals can be associated with unique credentials that can represent a user.

Accounting

Telcos require a lot of flexibility in accounting to provide as much flexibility in charging for added value as they do for the technical functionality of services. To cater to this, every call to any service and RBB method can be charged in whatever way the telcos wishes. Before the actual method is executed, the relevant charging is performed. We cannot detail the charging functionality, but this report does indicate where accounting does hook into the method calls performed within the framework. The remainder is left as an implementation detail.

3.2.4 Signalling Protocol

Seeing as the main attribute of the framework is that it supports and is based on ALS, the actual application layer communication protocol forms part of the framework's design. Section 3.2.1 already covered this from the point of view of treating message passing between objects as the main communication mechanism. We touched on the requirement of having method calls marshalled into a format which can be carried by the transport network. We now detail this requirement.

When two local objects interact the messages passing between them are in the form of a method call with specified arguments. In remote interaction between objects, the name of the object would also have to be included to assist the ASs in which the objects reside in identifying the object for which the method call is intended. Overall, in the ALS context, this would mean that messages between nodes would contain the following information:

- object name
- called method
- list of arguments for method

Obviously, more information could be included, such as the package into which the object is grouped (assuming the implemented framework makes use of a package mechanism like Java's `package` or Python's modules) and the types of each of the arguments. However, here we are only looking for essential elements and so it is from the above list that we identify the requirements of the marshalled method calls which act as the ALS protocol.

A protocol has two identifying features. The first is the actual format or structure of the messages and deals with issues of encoding and readability. The second is the mechanism with which they are sent, specifying the sequence of events which occurs to send a message from one node to the next. Provided that the protocol is capable of transporting method invocations from one node to the next, one could argue that the detail of its mechanisms and message structure are irrelevant. This argument is not completely untrue, especially since the real advantages of ALS lie in the direct communication between terminals and ASs and not in the details of the signalling that provides this. On the other hand, one could make the same argument about any service architecture, be it the IMS or the IN. Yet these two telecoms infrastructures both have rigorous definitions of their various protocols for the purposes of standardisation, interoperability and ensuring the use of best practice techniques. Whilst we are not aiming to standardise on any protocols within the scope of this research report, recommendations are put forward based on the grander purpose of ALS.

Message Structure

The structure of messages should be able to represent the information describing the call to a method on a remote object along with arguments for that method. We are designing for the simplest case in which argument types are not included and thus all arguments can be represented as strings. This is important in that the protocol will be text-based like many other modern application layer protocols that are in common use in telecoms services such as SIP and the Hypertext Transfer Protocol (HTTP). Then, we look to the Extensible Markup Language (XML) to provide the text-based

```
<?xml version="1.0">
<message>
        <service-object>InstantMessaging</service-object>
        <method>sendMessage</method>
        <arguments>
                <argument>+27825551212</argument>
                <argument>I dont know where I am but the food
                    in the fridge is awesome! LOL!</argument>
        <arguments>
</message>
```

Listing 3.1: XML formatting of a remote method invocation of a service object

structure. XML is adept at representing objects in a textual manner and is also in common use by telecoms web services and most modern programming languages also have built-in support for it. XML's structure makes for human-readability which is advantageous in that it makes for efficient debugging and ensures that the meaning of messages can be comprehended more easily by programmers.

Listing 3.1 presents an example of a message structured using XML. The schema is very simple but conveys adequate information to invoke a remote service method. The message does not explicitly associate the arguments with specific method parameters, instead the schema relies on the position of the arguments much like how methods are normally called in most languages. That and any other extraneous information which is not included here, can be included in an actual implementation at the cost of lessening simplicity, but for clarity only the bare necessities are shown.

The schema also does not include a place for a destination address. ALS deals with *direct* application-to-application signalling, and so, unlike SIP which uses proxies, does not have to consider the routing of messages. Instead ALS can rely on the transport network for addressing functionality. The manner in which this is performed is part of the signalling mechanism of the protocol.

Signalling Mechanism

The actual mechanism for sending messages is built up around a list of requirements of which some have already been covered; the following should be supported:

- remote method calls
- asynchronous message sending and receiving

- direct application-to-application signalling
- peer-to-peer relationships

Remote method calls This is more of an overriding requirement which leads to the other three: the signalling mechanism should support the transportation of the marshalled method calls in the format of the XML structure above. The protocol's transmission mechanism should therefore provide an applicable "channel" over which the message can travel according to the following three requirements.

Asynchronous message sending and receiving The asynchronous nature of the framework affects the protocol on two levels. The first is that a specific method of a service object should be able to be accessed by multiple terminals simultaneously (a topic covered in regards to message context in section 3.2.6). The second relates to the returned value of a method. Within a single program thread of a non-distributed application, method calls are synchronous: the thread's logical flow waits for a method to return with a value. Doing this in a remote multithreaded situation is more complex because it relies on each method call being transported with a unique identifier enabling the returned value to be associated with the original method call. This results in extra overhead which can be overcome by specifying that the return of values is not supported. If a method has to send information back to the calling node (be it the AS or terminal), this can be performed inside that method by calling a method on a remote object of the calling node. This is better supported in an asynchronous environment and does not require any extra supporting functionality. We summarise by stating that the protocol does not support synchronous communication.

Direct application-to-application signalling This is a requirement which links to the addressing of terminals. The protocol does not have to cater to addressing because nodes operating at the application layer do not perform any kind of relaying of messages (unlike that which is performed by SIP proxies). Instead a direct channel is established from origin to destination and the transport network can be relied on for addressing and message routing as service messages are not processed by any intermediate transport network nodes. The form of this direct channel is not specified and left as a transport network issue, but we can propose some ideas. For example, a terminal can open up a connection in the form of a *socket* to keep the channel continuously open while the terminal is associated with the AS. This kind

of continuous connection is often blocked by firewalls. Thus another example would be to use a web service type of connection to overcome firewall issues, but it would not be a web service in the true form as both the terminal and the AS would have to act as HTTP servers, which would blur the client-server relationship of web services. However, we will not delve deeper into this transport network issue.

Peer-to-peer relationships OO interactions are inherently peer-to-peer in that no one object can assume a server role. In a client-server model clients can only make requests to servers and not vice versa. The closest a client-server model can come to a server invoking client functionality is using a notification pattern. Whereas servers can never make requests to clients, objects all have the capability to invoke methods on other objects (provided they have the permission to do so). Since the protocol is built up using OO concepts, a peer-to-peer relationship between terminals and ASs is more applicable. It enables the asynchronous communication discussed above by allowing both terminals and ASs to invoke the methods of each other's objects. At the protocol level this means that message transmission is *symmetric* in that there is no telling apart remote method invocations from terminals or ASs.

3.2.5 Dynamic Object Loading and Method Invocation

The consequence of objects interacting remotely by having method calls marshalled into a form suitable for network transportation, is that dynamic object manipulation has to be introduced. Traditional non-distributed OO source code makes reference to class names in a static manner. That is, when an object is to be created from a class, the actual name of the class appears in the source code. When one of this object's methods is called, the actual name of the method also appears in the source code — more formally, the specific class and its methods are referred to at *compile time*. Although this is the most intuitive way to create objects and invoke their methods, objects do not necessarily have to be manipulated in this way, and for some cases, such as in the case of making remote calls to methods, they cannot be manipulated in this way.

Most modern, industry-standard languages include the ability to load objects dynamically and call their methods. Objects can be created from classes whose names have been specified at *runtime* and similarly, method names can also be specified at runtime for invocation. This is useful to the framework in that when a method call

is unmarshalled, the method on the remote terminal or AS can be called despite its name and its associated object's name not being "hardcoded" at compile time.

Further benefits of this dynamic manipulation of objects result from the fact that this decouples the framework's source code from that of service and RBB logic. These benefits include enabling access control on objects and also allowing new services and RBBs to be installed without having to restart the framework process on the host AS. The former is useful in the context of the inter-entity access control of the framework (see section 3.2.3). Instead of the traditional direct method calling on objects, methods are invoked by calling a framework method of the form `invokeMethod(objectName,methodName,arguments)`[1]. This `invokeMethod()` method can implement the above-mentioned access control, before actually invoking the method. The latter benefit is critical in a telecoms environment with a 99.999% uptime requirement, since it ensures an AS can continue to operate, and not be shutdown to install new services.

For the framework design, dynamic object manipulation will not be explicitly shown, but it will be assumed to be in place. Any SDs will show remote objects being manipulated as if they were co-located with local objects, meaning that an interaction between two remote objects will be shown by direct method calls by one object on the other. The reader must be aware that dynamic method calls are actually being used in this case, and are being relayed to the called object via a suitable manager object to be introduced in section 3.3.4.

3.2.6 Message Context

Communication between terminals and ASs occurs using methods calls on remote objects. Consequently, consideration must be given to the mechanism with which a programmer can identify a terminal which called a specific method. Service logic is invoked as per section 2.5.4, leading to a situation in which two or more terminals could simultaneously call the same method on the same AS. In the majority of cases the AS would have to reply to each terminal with some specific contextualised message, which may be different for each of the terminals.

As a rudimentary example, say there is a service object called `ShortMessage` which allows a terminal to send a short message to another terminal. The object has

[1] In the framework this method is actually called *callServiceMethod()* and is called in the context of invoking service methods.

a `sendMessage(dest,msg)` method which, when called, sends a message from the requesting terminal to the destination specified as a method argument. The logic of this method needs to be able to signal back to the requesting terminal indicating the success of sending the short message. If two terminals simultaneously request a short message to be sent and one is successful while the other is not, the service logic must be able to send back a different signal to each of the terminals.

There are two options for accomplishing this. The first would require the programmer to include a reference to the terminal at the beginning of each method definition (from the example above: `sendMessage(terminal,dest,msg)`). The other option relies on the fact that due to the asynchronous nature of the framework, multithreading would be used to spawn a new thread for each service logic invocation. Programmatically, this would mean that an invoking terminal could be associated with a thread in a lookup table. Within the flow of a particular thread, multithreaded languages allow for a reference to that particular current thread to be obtained (for example, Java's `Thread.currentThread()`). This reference then provides the index of the associated terminal in the lookup table.

The second option is simpler from a programming point-of-view, as it means that a programmer does not have to include the terminal in the list of each method's parameters. Yet, it does introduce the overhead of a lookup table albeit hidden from the programmer. This means that both options are viable but have their drawbacks. We do not standardise the mechanism for identifying message context, but we select the second mechanism in this report for describing the operation of the framework. In an actual implementation, whichever way message context functionality is implemented, care must be taken to ensure that the interface into ALS is kept as simple as possible.

3.2.7 Reusable Functionality

Reusability is defined as "the extent to which a program (or parts thereof) can be reused in other applications" [29, pp 111]. Reusability is not a new concept in the telecommunication service development industry. Services in the IN were built up from SIBs which encapsulated functionality that was common to multiple services. The idea being that these SIBs contained flows of program logic that could be connected to each other to form the logic of a full service. A more contemporary example comes indirectly from the Parlay SOA approach. In standardizing the interfaces into the bearer network, Parlay allowed for easier reuse of software modules. Any

module that complied with Parlay could be slotted into a service application that made use of the Parlay standard and required that specific module's functionality. By pushing abstraction as an important concept for telecoms, Parlay made a good case for proper software reusability in telecoms and persuaded telecoms software developers and vendors to develop software in a reusable manner.

In the IT industry, reusability is a more prevalent concept. Any experienced software developer will look to reuse as much functionality as possible. High quality reusable software modules are widely (and often freely) available. The obvious benefit being that this allows software to be developed rapidly. The other benefit being that software developers can focus on building high quality application logic (i.e. the application logic that is unique to their software) whilst relying on 3rd party developers to maintain the quality of the reusable functionality. Nonetheless, even with IT demonstrating the benefits of software reuse, telecoms has had a much slower uptake of it. Software modules, more often than not, are proprietary, make no use of open interface standards and are not considered for use in other applications.

The software framework of this research report will include software reusability as a formal concept and as such it will be one of the core features. We have already considered it in terms of its integration with access control (see section 3.2.3) and since abstraction has been made into a very important aspect of the framework so far in this report, reusability, which arises naturally out of abstraction, is already inherently part of the framework. Its integration into the overall framework and its use within service logic will also be covered in section 3.3.1. Whilst reusabilitiy will be emphasised, the functionality, structure and interfaces of specific reusable components will not. We have recognized that standardised interfaces are important, but specifying the exact form of these interfaces is beyond the scope of the framework and the work of this research report. Instead we will show how reusable modules' (or RBBs') functionality will be invoked, how they will provide the abstracted "downwards" interfaces into the bearer network and how new RBBs can be integrated.

3.2.8 Location of Essential Value-added Functionality

Certain reusable functionality is essential to the value, but not necessarily the structure, of the framework. For example, BCCM provides the abstracted interface into the bearer network and is thus essential to the telecoms context of the framework. However, not all services require its functionality, thus making it auxiliary to the structure of the service framework. This is also true since the actual BCCM

implementation can be changed depending on the technology of the underlying bearer network.

The fact that some important value-added functionality is not crucial to the functioning of some services means that this functionality should be encapsulated within RBBs or as services as opposed to being presented as dedicated components of the overall framework . This point will be clarified with other examples when the service framework is presented in section 3.3.

3.2.9 User Identity

Within traditional telecommunications users are represented by their E.164 numbers. If one party wishes to contact another be it by a basic voice call or by short message service (SMS) message, the E.164 number is used to identify the destination party. This applies to the framework as well. Every terminal which will connect to the application server will have to be identified in some way. We are not aiming to replace the user experience of telecommunications functionality. So for example, as above, when one party wishes to call another they should still be able to use the E.164 numbering system. When a basic two party call is performed using ALS, the caller dials the called party's E.164 number which identifies the called terminal. The AS signals, over ALS, to the called terminal that there is an incoming call, and if the terminal is not engaged, the bearer network is invoked to setup the call.

The framework is therefore designed to challenge a terminal with user credentials when it first connects to the AS. On a successful authentication, the terminal is associated with the user credentials, including an E.164 number. The number is then used to contact the terminal for any service session. We note that the E.164 numbering system does not have to be used and can instead be replaced with a username system. However, telecoms end-users are accustomed to E.164 numbering and thus it remains as an identifying mechanism.

3.3 Main Framework Components

The overall design of the framework is presented in OO form in figure 3.2. The design draws on that proposed in [19]. It is presented at a high-level and does not show some of the important features of the framework which have so far been discussed:

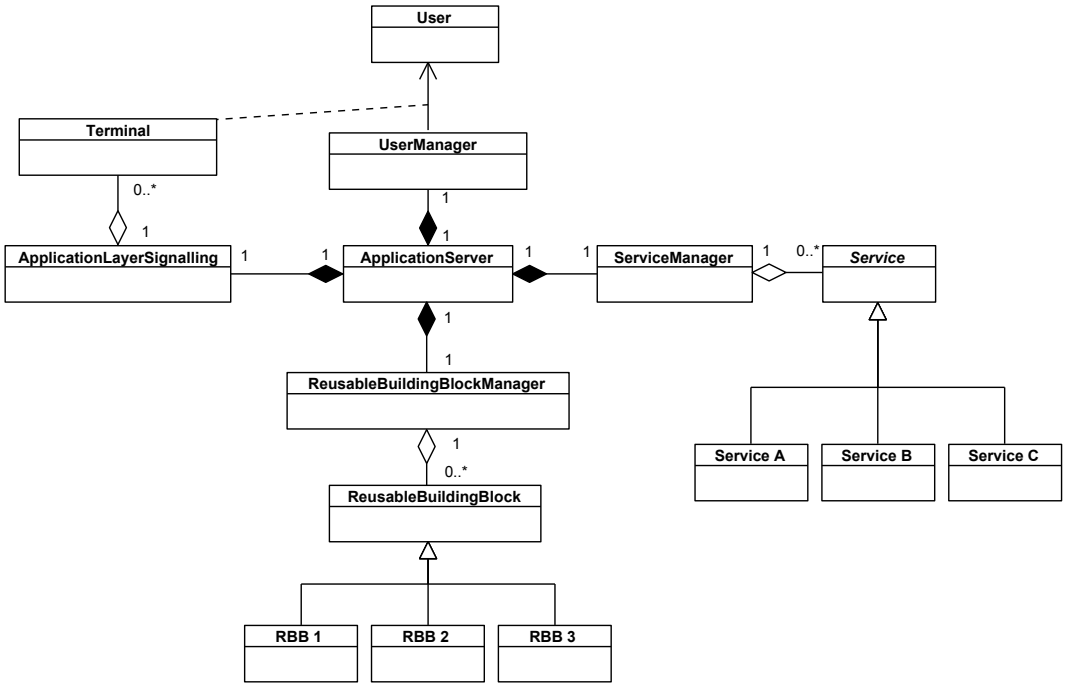


Figure 3.2: UML class diagram of overall framework

BCCM and AAA. Section 3.2.8 explained how BCCM is a feature best captured as an RBB, and not as a component of the overall framework. Similarly, AAA is not part of the high-level framework, but rather will be presented as a formal *service* for reasons to be stated later in section 3.3.3.

The framework is made up of five discernible parts. Two of the parts, *services* and *RBBs*, directly concern programmers who are developing software within the framework. The other three, being the *ALS interface*, *user profiles* and the various *managers* are abstracted from service programmers, but are instrumental in the operation of the framework. The five major framework components are now covered.

3.3.1 Reusable Building Blocks

RBBs have been contextualised within the topic of atomic and composite services (section 2.5.1) and within the software engineering technique of reusability (section 3.2.7). One more inferred purpose of RBBs is their provision of one of the framework's hallmarks: abstraction. In the same vain as JAIN's *RAs*, RBBs also provide reusable functionality of which some includes adaptation to lower-level telecoms network resources.

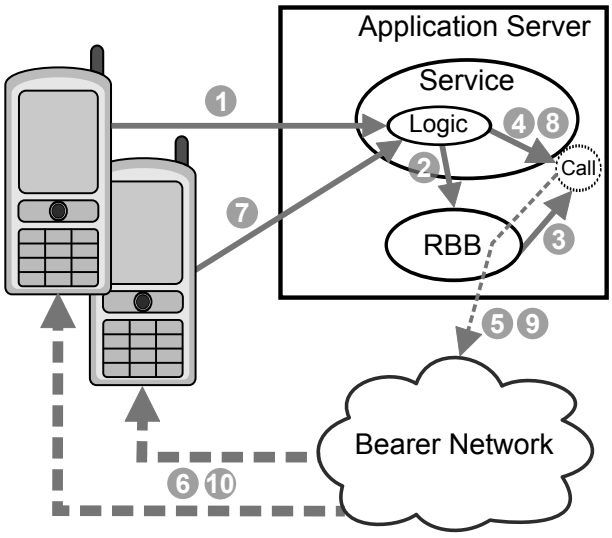


Figure 3.3: Basic operation of the two party call RBB

The purpose of RBBs is to encapsulate that functionality that will be reused in different services, whether the functionality be the interfacing with other systems or simply some useful reusable logic. This report will not actually standardise the framework's RBBs. Instead, by way of example, show how they are integrated and their functionality called on.

Example RBB: Two Party Call

We will use a BCCM example to convey various concepts of RBBs. The *two party call RBB* is an essential element of many telecoms services and so is a fitting example. This RBB encapsulates basic call control between two parties. It offers three functions: making a call, hanging up a call and notifications of call events. Its operation is explained with reference to figure 3.3.

In context of a basic call service, this RBB is called on when a terminal sends an ALS request to an AS (**1**) to setup a call with a specified destination terminal. The service logic invokes the RBB (**2**) to create a reference to the call (**3**). The service logic uses this call reference (**4**) to request from the bearer network (**5**) that it setup a call using CS signalling (**6**). When either participating party wishes to hangup the call, a request is sent from that party's terminal to the AS (**7**). Using the object reference to the call, the RBB's hangup operation is called (**8**) which requests from the bearer network that it disconnect the call (**9, 10**). Not shown in figure 3.3 are the notifications which the RBB provides. These notifications are used to couple

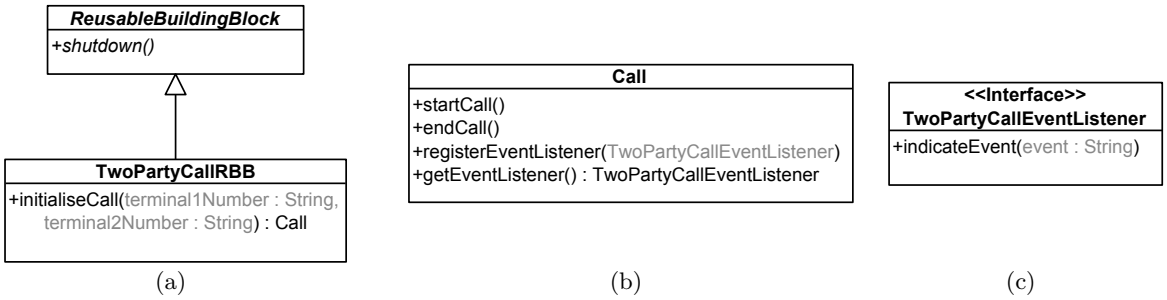


Figure 3.4: Classes of the two party call RBB

the call state model of section 2.5.3 to the BCCM provided by the framework.

We now detail the structure and operation of RBBs using the two party call RBB just specified.

Structure and Operation

RBBs are made up of multiple objects which cooperate to provide reusable functionality. The first point of contact to each RBB is a single object with the same name as the RBB (such as `TwoPartyCallRBB`). This object constrains the location of access-control to a single point, making for simpler access management. It also presents the RBB with a minimalistic interface. Each RBB has one of these "first-contact" objects which is implemented using the *singleton* design pattern [9, pp 127-134]; these objects exist from the point that they are first used until the time that the framework is shutdown, meaning that the same object is always invoked, regardless where the invocation originates.

The "first-contact" objects inherit from the `ReusableBuildingBlock` abstract class shown in figure 3.4(a). In doing so, they must override the `shutdown()` abstract method. This method is used when the AS is shutdown. When this happens, `shutdown()` is called on all RBBs that have been instantiated. This gives the RBBs an opportunity to perform any cleanup that needs to occur before the AS is shutdown.

The entire RBB is not just made up of a single "first-contact" object. Rather this single object can be seen as a kind of factory, which produces the objects which the service logic manipulates in order to interact with the RBB. In the two party call example, calling an `initialiseCall()` method on the `TwoPartyCallRBB` object

produces a `Call` object with which the service logic interfaces to control the call. More formally, the `TwoPartyCallRBB`'s single method is defined as in figure 3.4(a).

The `Call` object which the `initialiseCall()` method returns has four methods which map onto the RBB's three main operations shown in figure 3.4(b). The first two methods, `startCall()` and `endCall()`, are self explanatory. The last two are critical to ensuring that there is a mechanism for the RBB to signal the service logic that particular events have occurred. The mechanism relies on a listener pattern to implement notifications. The listener, which is passed into the `registerEventListener()` method, is an implementation of a formal programming *interface* (for example, an `interface` in Java or a class with only pure virtual functions in C++) in figure 3.4(c). The interface specifies various methods which the RBB calls in response to specific events. In the two party call RBB, notifications exist for each step in a call setup, for failed bearer connection setup, and for expected or unexpected termination of the bearer connection. All these events can be mapped on to the FSM of figure 2.6.

Notifications are critical to the implementation of the call state model of the BCCM of the framework, which uses the above mentioned FSM. The call state model is implemented as a separate RBB since it is used in the same form for multiple bearer connectivity RBBs. For the two party call RBB, the service logic would register a concrete implementation of `TwoPartyCallListener` that updated the call state model in response to events which mapped to relevant state transitions. This is one example of the use of notifications which will be further detailed in section 3.3.6.

From the above details we can extract the following from RBBs:

- The first point of contact of an RBB is a single object which
  - is a singleton
  - is named with the same name as the whole RBB module
  - acts as a factory to produce the other objects with which service logic interacts to perform desired tasks.
- RBBs never signal directly to service logic; their functionality is only ever called-on and they never operate independent of service logic.
- An RBB can signal back to service logic indirectly using a notification pattern. The service logic must create a concrete implementation of a listener with predefined methods which the RBB calls on in response to defined events.

- An RBB only notifies of events occurring in resources which it encapsulates. However, if the event occurs in a terminal, the notification is rather sent using direct ALS, even if the event occurs within the context of one of the RBB's operations.

3.3.2 Application Layer Signalling Interface

In the JAIN standard all interfaces to external systems are presented as resource adapters which perform that exact task: enabling the application server to adapt to independent resources. Implicitly this also conveys that this adaptation performs abstraction between layers. We raise this point to contrast the implementation of ALS within the framework to the way one would attempt to implement it in other frameworks. If JAIN were to be mapped to our framework, RAs would fill the role of RBBs. Thus, using JAIN, ALS would simply be presented as another RA. This contradicts a number of points about RBBs and ALS. Firstly, ALS is constrained only to the application layer (section 2.5.4) whereas RAs interface with lower layer functions. Secondly, RBBs do not call on service logic directly. This means that the lower layer entities with which these RBBs interface, are not able to invoke services within the framework. On the other hand, ALS allows terminals and ASs to signal directly and asynchronously to each other and the relationship between them can be seen more as a peer-to-peer (section 3.2.4). Thirdly, RBBs are interchangeable and the framework can technically operate without them, albeit with reduced functionality. This differs from ALS which was the catalyst for the development of the framework in the first place and thus is ingrained in it and critical to the moving of service logic to the application layer. Lastly, RBBs encapsulate the functionality of resources that contain no orchestrating service logic whereas ALS enables service logic to be distributed between ASs and terminals - resulting in the requirements of ALS being different to those of RBBs.

Structure and Operation

The above discussion contains the reasoning for the framework's ALS interface being a completely separate component of the framework to RBBs as per figure 3.2. This separation assists service developers in differentiating between reusable functionality and service logic. The ALS API is presented as distinct from that used to invoke RBB functionality. It is made up of the main `ApplicationLayerSignalling` object

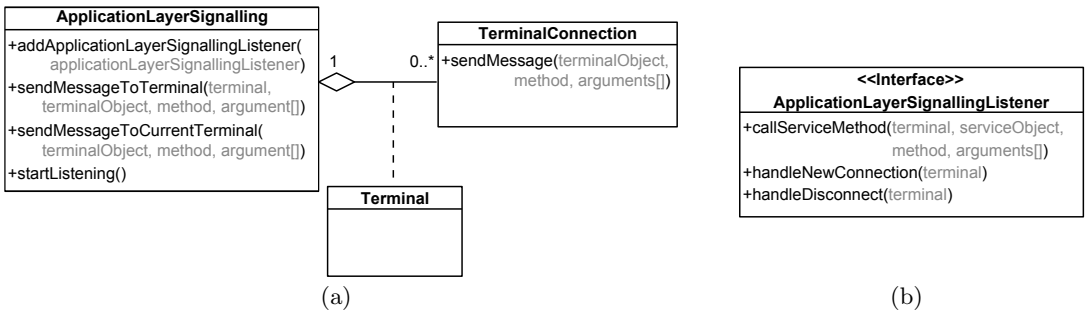


Figure 3.5: ALS Classes. The association class in figure (a) represents the mapping between `Terminal`s and `TerminalConnection`s

(figure 3.5(a)) which encapsulates the ALS mechanisms and protocols and presents a clean and simple interface for sending messages to terminals.

Before the ALS component begins listening for connections from terminals, an `ApplicationLayerSignallingListener` implementation is registered with it by calling `addApplicationLayerSignallingListener()`. This listener's methods, shown in figure 3.5(b), are called when a new terminal connects or disconnects and when a terminal sends a message to the AS. Once the listener is registered, the framework calls `startListening()`. Then, on a terminal's connection to the AS, `handleNewConnection()` receives a new `Terminal` object (figure 3.5(a)) which represents the terminal's association to the framework. The method also calls on AAA logic to challenge the terminal to login with correct credentials, the operational details of which we leave for section 3.3.3. Every `Terminal` is associated to a unique `TerminalConnection` object which has a single method, `sendMessage()`, that the ALS component calls to send a message to the physical terminal represented by the object. This method does not block, due the specification of only asynchronous message sending and receiving in section 3.2.4.

Service logic never calls `sendMessage()` directly and, in fact, has no direct communication with `TerminalConnection` objects at all. User identities are not tied to specific terminals, meaning that messages that an AS sends to a terminal are within the context of a *user*'s session. Service logic cannot signal directly to terminals as there needs to be a way to decouple users from terminals. As we will see in the next subsection, once a terminal has been authenticated using a user's login details, the terminal is associated with the user and service logic can now send messages to terminals using a user's identity, instead of directly to the terminal. To further this decoupling, as mentioned above, every `TerminalConnection` is mapped to a `Terminal` object which is nothing more than a mapping key that the service logic

and the `ServiceManager` (also introduced in section 3.3.4) use to identify a terminal uniquely. The `TerminalConnection` to `Terminal` mapping is performed when a new terminal connects to the AS and is stored in the `ApplicationLayerSignalling` object. To send a message to a terminal, `sendMessageToTerminal(terminal, ...)` can be called (i.e. using the `Terminal` key to specify the terminal to which the message should be sent), which in turn calls the relevant `TerminalConnection`'s `sendMessage()` method.

To describe communication in the direction of a terminal to an AS we return to the `ApplicationLayerSignallingListener` and its `callServiceMethod()` method. The method is defined with the following parameters:

- `callServiceMethod(terminal, serviceObject, method, arguments[])`

When a terminal sends a message, the ALS component unmarshalls the message, creates a new thread of execution for the remote method call and finally calls the above `callServiceMethod()` within this thread. The `terminal` parameter identifies the source of the message and the framework uses this identifying parameter to associate the message source to the thread of execution by passing it on to `ServiceManager`'s `sendMessageToService()` method — providing a means for the service logic to signal back to the message source if need be by effectively creating a "message context" (according to section 3.2.6) and enabling `ApplicationLayerSignalling`'s `sendMessageToCurrentTerminal()` method. The detail of this is more relevant to the service manager component and so is left to section 3.3.4.

`ApplicationLayerSignallingListener` also caters for notifications of terminal disconnection, using the `handleDisconnect()` method. This method is called when the terminal physically disconnects from the AS. There are two scenarios for this: an expected or unexpected disconnect. An expected disconnect is a graceful one in which the terminal first logs off and then physically closes the ALS channel between it and the AS. In this scenario, the terminal sends a logoff request to the AS, which then sets off a process of ending all service sessions in which the terminal may be involved. The AS then acknowledges this logoff request, in response to which, the terminal physically closes its connection to the AS. This in turn triggers the `handleTerminalDisconnect()` method which is called as a notification that the terminal's connection has been physically terminated indicating that the terminal's association to the framework which was created when the terminal initially connected has been removed.

In the other scenario, the physical connection between the terminal and the AS may end unexpectedly, due to some environmental reason. At this point, a similar sequence of events as the expected disconnection occurs, in the opposite order: first `handleTerminalDisconnect()` is called, and then the logoff procedure occurs, minus the acknowledgement.

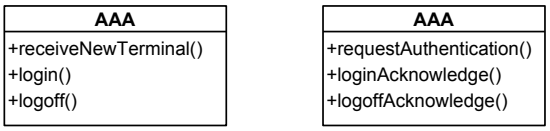
In the above two scenarios we indicated that all service sessions are ended when a terminal disconnects from the application server. To accommodate this, all services have to implement a method which is called in response to a terminal disconnecting in the above scenarios. This ensures that services are not left in an invalid state in the case of a service session not ending completely before a terminal's association to an AS ends. We look at this implementation detail in section 3.3.4.

3.3.3 User Profiles

User identity within the framework was introduced in section 3.2.9. A user can be identified with any unique feature, however we selected legacy E.164 numbering to keep the user experience of "dialing" the same, as a further means of showing that an ALS-based solution to services can be used in contemporary telecommunication networks.

User profiles cannot be introduced without covering some implementation details of AAA. Although AAA exists in the framework as a formal service, it is critical to the framework's operation. From the perspective of user profiles, it provides the mechanisms that the framework uses to identify a user initially and to provide the necessary authentication. The AAA service challenges the terminal when it attempts to login, and then orchestrates the process of looking up the user's credentials, authenticating the user and then associating the user's profile with the terminal.

The detailing of how services signal to terminals or how the framework itself makes a request to a service is left to sections 3.3.4 and 3.3.6. Nevertheless, AAA is implemented as a service, so the reader should be aware that services have access to features of the framework, and that it is not just terminals that can invoke service functionality but also the framework itself. Another design detail also relevant to user profiles, is that service logic in terminals is also presented using objects with methods that can be called remotely. Figure 3.6 shows the service objects that exist for AAA in both ASs and terminals.



(a) in AS (b) in terminal

Figure 3.6: AAA service objects showing only those methods which assist the implementation of user profiles.

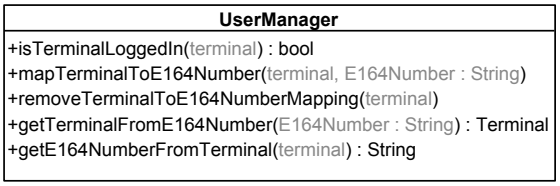


Figure 3.7: User profile class

One more design issue relevant to user profile functionality is that in the description of the operation of user profile functionality we make mention of a "database". User profile information has permanent storage in this database, the interface to which is implemented as an RBB. For this discussion, the reader can assume that this database functionality is in place and that whatever user profile information is required can be retrieved using a suitably abstracted database RBB interface.

Structure and Operation

The user profile functionality operates in the following way (restating some operational details from section 3.3.2). In response to a new terminal connection, the ALS interface signals to the framework to call the `receiveNewTerminal()` method of the `AAA` service object, shown in figure 3.6(a). By calling `requestAuthentication()` on the terminal's `AAA` service object in figure 3.6(b), the terminal is challenged to authenticate itself to the AS. To meet this challenge, the terminal sends its credentials as arguments to the `login()` method. At this point processing is handed over to the user profile component, which looks up and authenticates the user's credentials. On success, the terminal is associated to user information retrieved from the database.

For the purposes of this report, we look to the bare essentials of user profile

functionality. The E.164 number can function as a unique identifier for a user. Thus authentication can occur using only this number and a password. Further, the E.164 number can be used as an identifier for the purposes of targeting a particular terminal during a service session. Therefore, once the user is authenticated by verifying the E.164 number and password supplied to the `login()` method, the terminal is bidirectionally mapped to the E.164 number within the `UserManager` during a call to `mapTerminalToE164Number()`. This enables the operation of the functionality of the `getTerminalFromE164Number()`, `getE164NumberFromTerminal()` and `isTerminalLoggedIn()` methods in figure 3.7.

It is acknowledged that more than just an E.164 number would be required for full user profile functionality of an actual implementation of the framework. For example, a user's access rights to particular services and RBBs might be stored along with other information about the user. We leave this as an implementation detail.

Once the `Terminal` has been associated to the relevant user information, a login acknowledgement (the `loginAcknowledge()` method) is sent to the physical terminal, ending the initial connection and login process. At this point the user is associated to the terminal in a way that would provide all necessary support to services to enable them to target specific terminals based on user information and also to obtain user information if required by a specific message context.

A terminal's logoff process also involves the AAA. In the graceful disconnection scenario covered in section 3.3.2 above, the terminal calls `logoff()` on the AS's `AAA` service object. The AS performs a formal logoff process (covered in section 3.3.5) and then acknowledges this by calling `logoffAcknowledge()` on the terminal. During the logoff process, the terminal-to-E.164 number is removed by calling the aptly named `removeTerminalToE164NumberMapping()` method.

3.3.4 Managers

The main task of manager objects is to facilitate communication between the four main components of the framework: services, RBBs, user profiles and the ALS interface. Whilst technically these components could communicate directly, a telco is given more flexibility in access control and accounting if all communication traverses manager objects. We cover these managers — `ServiceManager`, `RBBManager` and `ApplicationServer` — below.

Service and RBB Managers

Service and RBB managers operate similarly so we discuss them together. To introduce their operation we use the example of a terminal attempting to invoke a method on a service object. Once the remote method call is unmarshalled, the invocation travels via the service manager which then performs three functions:

1. determines if the terminal has access rights to either the service object or to the method
2. performs any accounting that should result from that method call
3. performs dynamic method invocation

The same sequence of events occurs when a service makes a call to an RBB method. In this research report the actual details of the mechanisms for implementing the first two operations above are not covered. All that is considered with regards to them is that the managers first perform access control and then accounting before the actual method is invoked. The third operation has already been introduced in section 3.2.5. We supplement this with a description of the actual process of invoking a method and a few more details with regards to the instantiation of services and RBBs.

Message context plays a critical role here. As we specified in section 3.2.6 that a new thread is started up every time a method is called on a service object and that the service object can use this thread to identify the terminal which called the method, this means that only a single service object per service need be instantiated. If the framework did not cater for message context a new service object would have to be instantiated for each service session. Since this is not the case, each service object becomes a singleton, only being instantiated the first time it is used and then staying in existence until the application server is shut down.

Whilst message context does not apply directly to RBBs, individual RBB methods are called within service message contexts, meaning that the same specification of singleton objects applies to RBBs. Section 3.3.1 covered the detail that each RBB has a single object as an initial point of contact, matching up with the message context requirement of singletons.

This entire discussion of single objects leads to a design detail of dynamic method invocation: when calling a method dynamically, only the name of the service or RBB

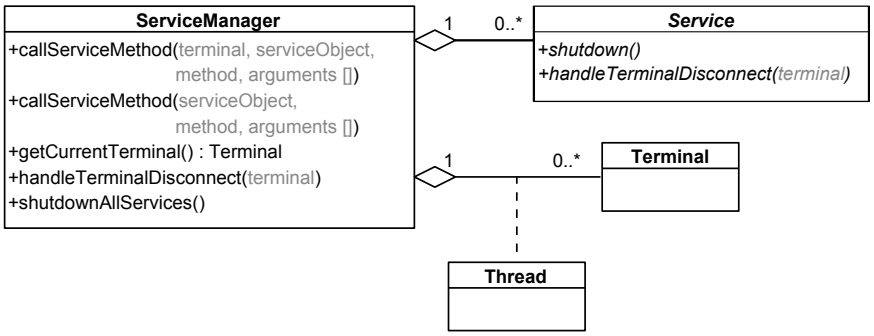


Figure 3.8: Service manager classes. Note the message context association between `Terminal`s and `Thread`s.

has to be specified and not the name of a specific instance of the object. It is up to the programmer to ensure that any service or RBB is designed with this in mind. For example in an instant messaging type service where two users can be engaged in a conversation, each conversation might have to be represented by a unique object. In this case, the main instant messaging singleton service object would have to create these conversation objects to manage the state of the conversations, as opposed to creating a new instant messaging service object for each conversation. This also applies to RBBs and in fact corresponds to the `Call` objects of the `TwoPartyCallRBB` and the "first-contact" objects touched on in section 3.3.1. This single object setup makes for simpler service logic invocation, removing the need for terminal service logic to obtain a reference to a specific object while also removing the need to have this reference included in any protocol messages.

Service Manager Structure and Operation

The `ServiceManager` object is a singleton with three main purposes: the first is to instantiate services when their functionality is invoked, the second is to relay messages (which are effectively method calls) to services and the third is to maintain message context when a terminal invokes a method on the service. Figure 3.8 shows how these functions are statically structured in the framework.

With regards to service instantiation, when either form of the `callServiceMethod()` method is called, the service manager first checks if this service has been used since the AS was started up, by checking if the service's singleton is stored in a collection of instantiated services in the `ServiceManager`. If not, the service's singleton is instantiated and stored in the collection. Once this process is completed, the `ServiceManager` performs dynamic method invocation to call the method specified

in the message sent from the terminal — which is the message relaying functionality mentioned above. Before relaying the message, the `ServiceManager` maps the terminal, represented by a `Terminal` object, to the current thread of execution created by the ALS component when the message was originally received by the AS. This enables the message context functionality of the framework. With this thread-to-terminal association in place, a call to `getCurrentTerminal()` will return a `Terminal` object representing that terminal that invoked the current thread of execution.

The message context association only occurs when the `callServiceMethod()` method which takes a `Terminal` as an argument is called. The other version of the method does not require a `Terminal` as its purpose is simply to facilitate one service invoking the functionality of another, as opposed to the invocation being received from a terminal.

The `callServiceMethod()` methods also perform the access control of section 3.2.3 before invoking any service method. The mechanisms involved in this are not considered in this research report, but the point is raised here to show where the access-control functionality hooks into the framework.

In section 3.2.4 the ALS protocol was specified using XML, a text-based representation of messages. For that reason, `callServiceMethod()` takes an array of strings as its `argument` argument, indicating that service methods can only accept arguments in the form of strings. The protocol also affects `callServiceMethod()` by specifying that this method does not return a value — a side-effect of the protocol's asynchronous behaviour.

The process of a terminal disconnecting from the framework was covered in section 3.3.2. During this process, all service sessions are ended, which explains the presence of the `handleTerminalDisconnect()` method. This method iterates through all services with which the disconnecting terminal is engaged in a session and alerts them to the fact that the terminal is disconnecting. This provides an opportunity to service logic to end its session with the terminal gracefully by having its overridden `handleTerminalDisconnect()` method called.

The `shutdownAllServices()` method, which is called when the AS is shutdown, iteratively calls the overridden abstract `shutdown()` method on all services. During this process all services must perform any cleanup required to ensure that the AS shuts down gracefully.

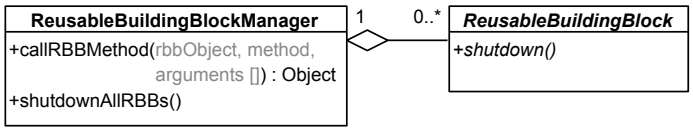


Figure 3.9: RBB manager classes

RBB Manager Structure and Operation

The static structure of the RBB manager component is very similar to the service manager. However, it lacks message context functionality and the ability to enable RBBs to signal to terminals. The `ReusableBuildingBlockManager`, shown in figure 3.9, has a `callRBBMethod()` method which performs the same object initialisation and dynamic method invocation that the service manager does. Because of this, RBBs are built up using the same singleton, first-contact object approach as services — a topic which was already covered in section 3.3.1.

RBBs are capable of signalling back to services using a notification mechanism. Section 3.3.1 listed that this involves creating a concrete implementation of a listener interface and passing it to the relevant RBB via a `registerEventListener()` type of method. Therefore, unlike service methods, RBB methods have to be able to accept objects, and not just strings, as arguments to their methods. The `sendMessageToRBB()` takes an array of objects as its `arguments` argument to cater for this. Moreover, since RBBs exist locally to service logic, their methods do not necessarily operate asynchronously, meaning that, unlike service methods, it is possible for them to return a value in the form of general objects (such as an `Object` in Java) which can be casted into whatever type required for their use.

Main Application Server Manager

The main `ApplicationServer` object is the first object that is created when the AS is started up. It contains singletons of each of the framework's main objects which represent the four main components (`ApplicationLayerSignalling`, `ServiceManager`, `ReusableBuildingBlockManager` and `UserManager`). As shown by figure 3.10 it provides four `get...()` methods to retrieve each of these components from anywhere in the framework. The `ApplicationServer` singleton object itself is retrieved by calling its own `getApplicationServerInstance()` static method.

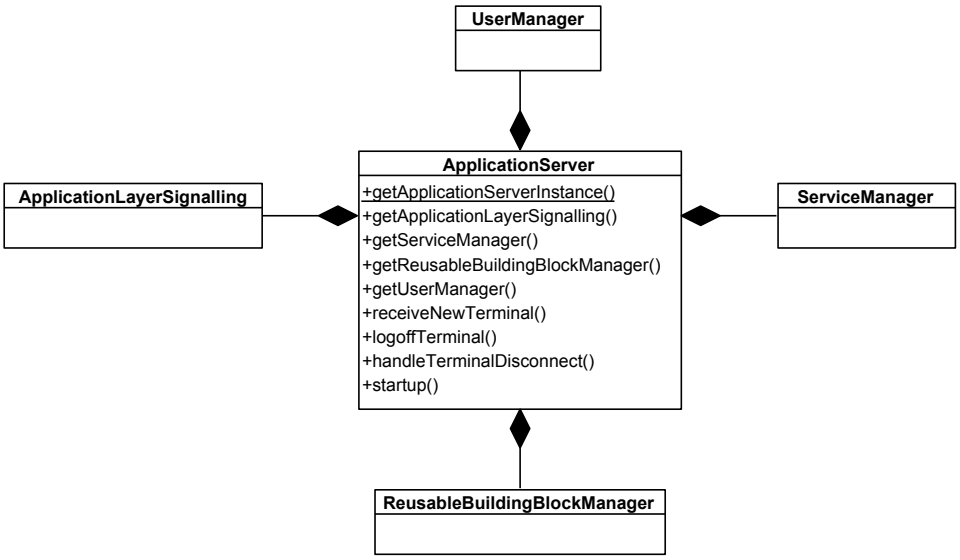


Figure 3.10: Application server class

The `startup()` method initialises the AS by instantiating each component and performing any necessary initialisation on them. For example, once initialised, this method calls the `startListening()` method of `ApplicationLayerSignalling`. `startup()` effectively provides the hook-in point for any initialisation that the framework's components require. Once this method has completed, the AS is ready to handle service logic.

`ApplicationServer` also takes part in the orchestration of the login and logoff processes that occur when a terminal connects to and disconnects from the AS respectively. `receiveNewTerminal()` is called in response to a new physical connection to terminal being formed. In the simplest case presented in this research report, the only operation which this method performs is to challenge the terminal for user credentials. `logoffTerminal()` and `handleTerminalDisconnect()` reflect the two parts to a terminal disconnecting from the AS: logging off and physical disconnection. In the case of a terminal formally logging off before physically disconnecting, `logoffTerminal()` is called before `handleTerminalDisconnect()` and ensures that the terminal's user's current association to the AS is removed first. The methods are called in the opposite order if the terminal disconnects unexpectedly and the AS is forced to clean up the association. These processes are all detailed in section 3.3.5.

3.3.5 Dynamic Behaviour of Supporting Infrastructure

Up to this point we have described various components of the framework that support service logic. We have presented their static structure and the functionality they provide. As each component is dependent on each of the other components, a detailed description the dynamic behaviour of each one requires an understanding of each of the other components. We have adequately covered each to now be able to present their dynamic operation and the manner in which they co-operate in the form of SDs.

The following scenarios are covered:

- a terminal connecting to and logging into an AS
- a terminal remotely calling a method on an AS
- a service invoking another service's method
- an AS calling a method on a terminal (either a terminal that is already involved in a service session or one that is not)
- a service invoking an RBB's method
- a terminal disconnecting (gracefully and non-gracefully) from an AS

Terminal Connection and Login

The terminal connection and login procedure in figure 3.11 begins when a terminal forms a physical connection with the AS. For example, this could be the form of a socket being opened between the terminal and AS. When this occurs, the `ApplicationLayerSignallingListener`'s `handleTerminalConnect()` method is called with a `TerminalConnection` representing the physical terminal as an argument (**1**). A new `Terminal` object is created and uniquely mapped to the `TerminalConnection` (**2**). This `Terminal` is passed on to the `ApplicationServer` object (**3**, **4**), which then instructs the AAA service to challenge the terminal for its credentials (**5**, **6**). It must be noted here that whenever a method is called on a service, this request always goes through the `ServiceManager` (be it from a terminal, a service or from another location within the frame), however we do not always show this in order to simplify the SDs. In the case being described, what is not shown is that the `ServiceManager`'s `callServiceMethod()` which takes a

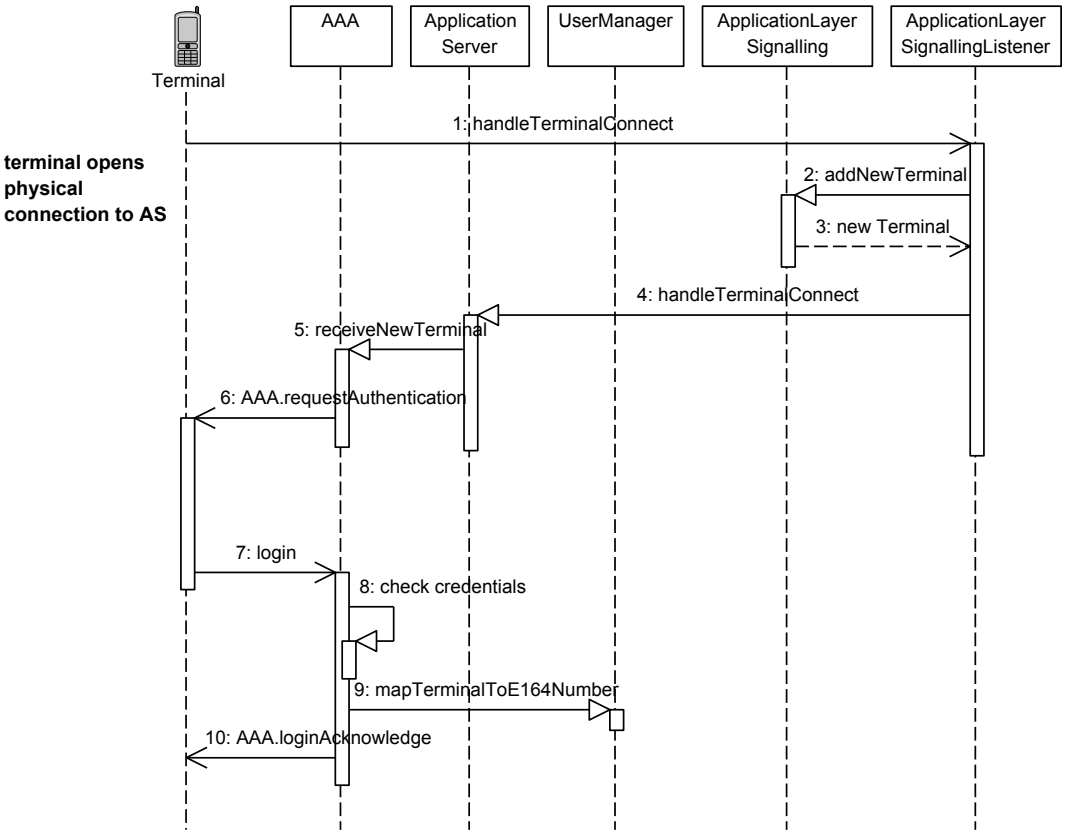


Figure 3.11: Terminal connection and login SD

`Terminal` argument is used, allowing `AAA` to use `ApplicationLayerSignalling`'s `sendMessageToCurrentTerminal()` method to send the challenge to the correct terminal. It should also be noted that the SDs will also always show service objects signalling directly to terminals, hiding the fact that `ApplicationLayerSignalling`'s `sendMessageToTerminal()` or `sendMessageToCurrentTerminal()` has to be called to facilitate this.

In response to the challenge for its credentials, the terminal sends them in a call to `AAA`'s `login()` method (**7**). The credentials are checked according to the implementation's authentication logic (**8**). The terminal's credentials, in this case, represented by an E.164 number, are then mapped to the terminal's representative `Terminal` object within the `UserManager` object (**9**) allowing the `Terminal` to be later retrieved by service logic. `AAA` then acknowledges to the terminal that the login was successful (**10**), completing the connection and login process.

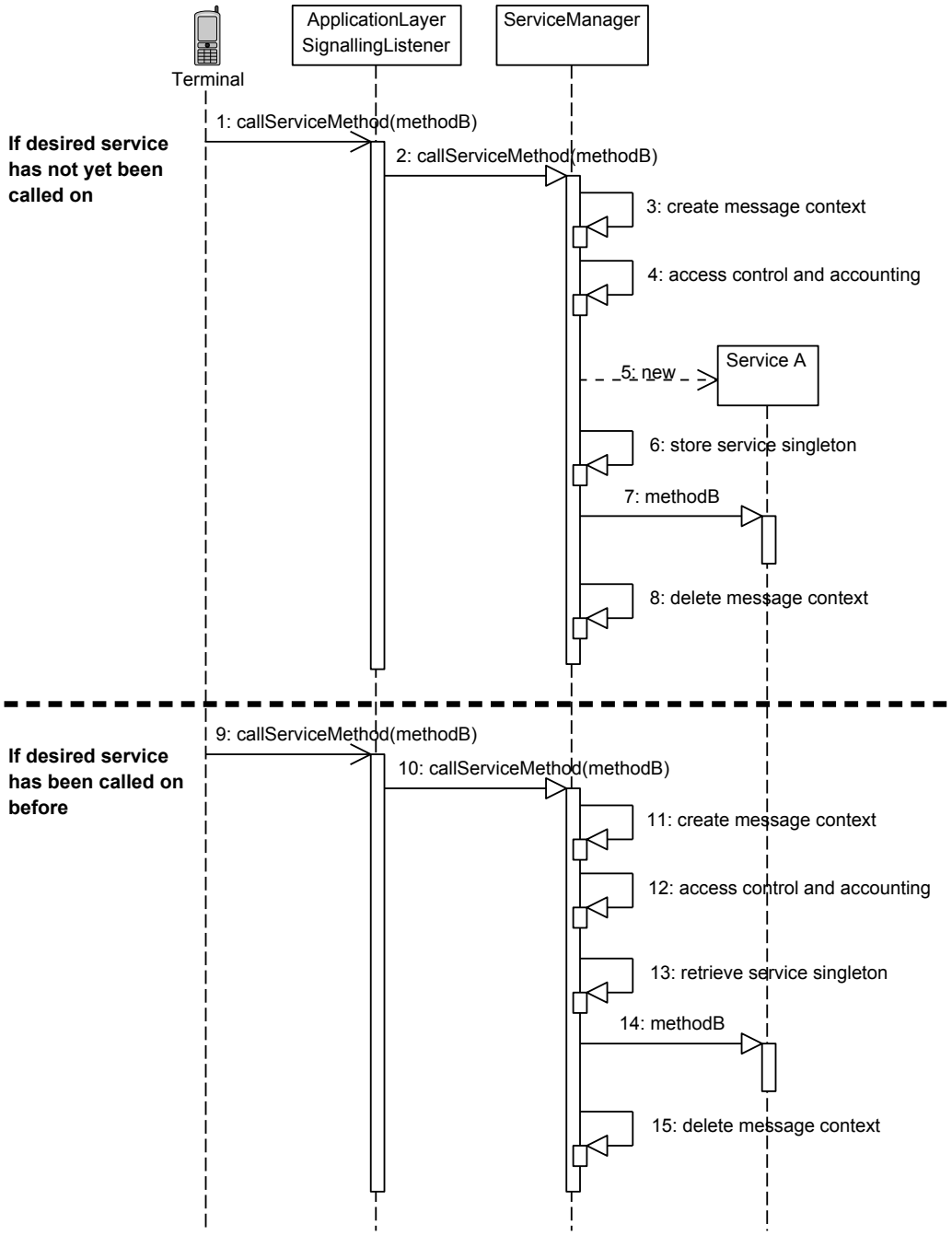


Figure 3.12: SD of remote method call from terminal to AS

Terminal to AS Remote Method Call

The SD of figure 3.12 encompasses all the operations that occur when a method call request arrives from a remote terminal. The entire process is abstracted from the service developer, so for simplicity's sake, other SDs will only show this as an arrow directly from the terminal to the service object. However, this SD breaks the whole

process down into the individual critical operations to show what actually occurs to facilitate a successful remote method call.

The SD shows two scenarios which result in a slightly different set of operations being executed. The difference lies in the fact that the first scenario covers what occurs when specific service logic is being used for the first time. In this case, the representative service object singleton has to be instantiated and stored before the desired method can be called. In the other scenario, the service object has already been instantiated and so just has to be retrieved. We now describe both scenarios.

When a method call arrives from a terminal a new thread is created (not shown in the SD) and `ApplicationLayerSignallingListener`'s `callServiceMethod()` method receives a `Terminal` object (representing the originating physical terminal) along with strings of the names of the desired service object and method name and a list of arguments destined for the desired method in the format of strings (**1** and **9**). A call to the `ServiceManager`'s `callServiceMethod()` method then begins the process of dynamic method invocation (**2** and **10**). Firstly, a message context is created by mapping an object representing the current thread of execution to the `Terminal` object (**3** and **11**). Then access control and accounting functionality determines whether the current terminal has access to the desired service object and method and performs any necessary charging (**4** and **12**) — the details of which are not covered in this research report.

If the desired service object singleton has not yet been instantiated, a new instance is created and stored inside the `ServiceManager` (**5, 6**). If it has already been instantiated since the AS was started up, the service object singleton is retrieved (**13**). After this, the actual service method is called synchronously to the current thread with the arguments that were originally sent to `callServiceMethod()` (**7** and **14**) (note that no value is returned from the method). At the end of the remote method call procedure, a cleanup operation removes the message context (**8** and **15**).

It is also possible for one service to call on another's logic. In this case, the sequence of operations is almost the same as above except that the originating service directly calls the `callServiceMethod()` overloaded method without passing in a `Terminal` object. Since it is a service initiating this method call, and not a remote terminal, no message context is created. Otherwise, the service method call proceeds in the same manner.

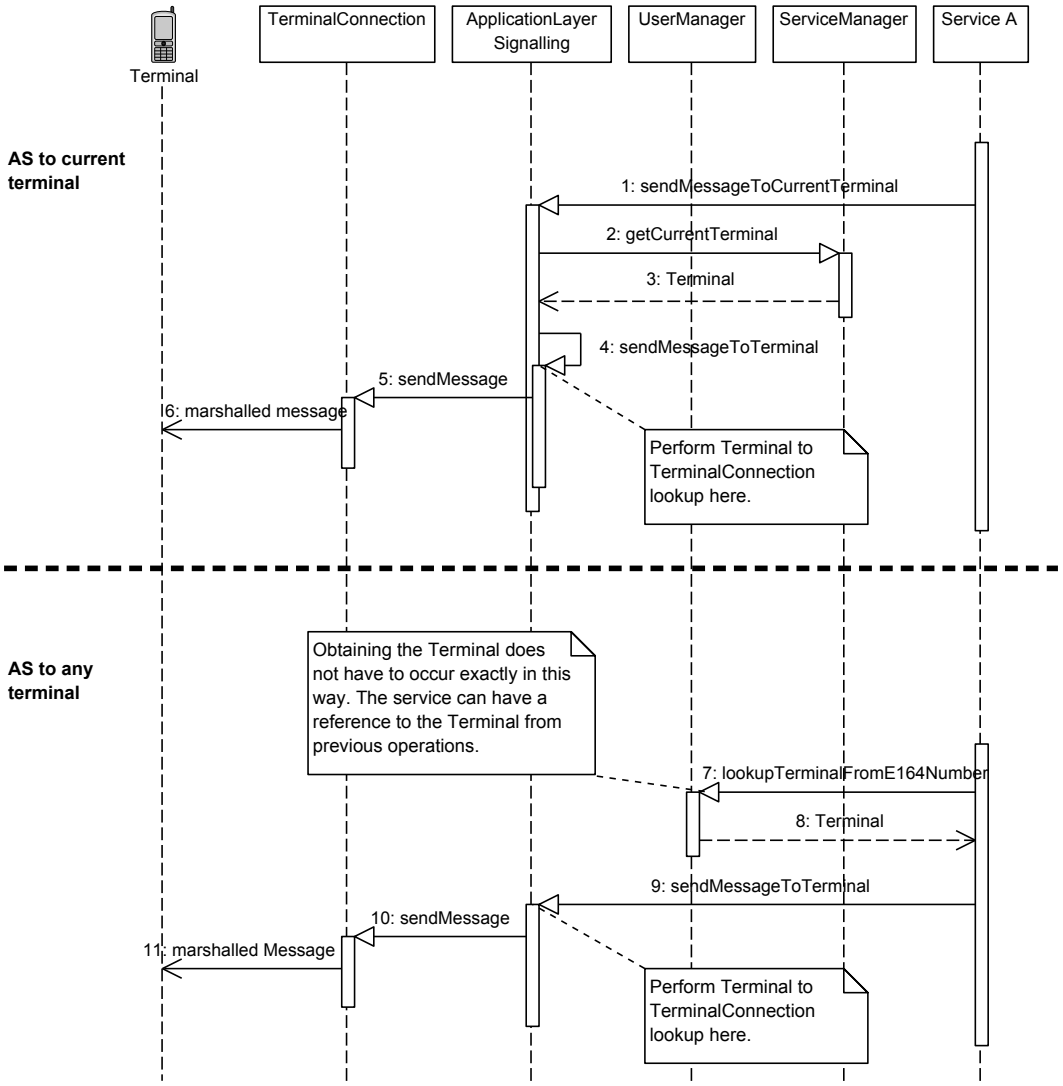


Figure 3.13: SD of remote method call from AS to terminal

AS to Terminal Remote Method Call

Similar to terminal to AS signalling, when describing service logic in SDs we show the services signalling directly to terminals. Whilst a programmer is presented with very simple `sendMessageToTerminal()` or `sendMessageToCurrentTerminal()` methods, the message arrows from ASs to terminals encapsulate a number of operations which have to be performed to transfer the messages successfully. These are shown in figure 3.13 and detailed below.

There are two cases for transferring messages, which arise from the two methods available to programmers for sending messages to terminals. In the first case, the service logic has to send a message to the terminal which remotely called a method

on the AS resulting in the current message context. The second case occurs when service logic has to signal to a terminal which did not make the remote method call resulting in the current thread of execution. In the second case, service logic may or may not have a reference to that terminal yet.

In the first case, service logic calls `sendMessageToCurrentTerminal()` on the `ApplicationLayerSignalling` object, passing into this method the desired object and method existing in the terminal and arguments in string format (**1**). At this point the current terminal has to be retrieved. In the description of figure 3.12 we explained how in creating a message context the object representing the current terminal is mapped to an object representing the current thread. Therefore, using a mechanism described in section 3.2.6, the object representing the current thread is retrieved and used to lookup the `Terminal` involved in the current message context (**2**, **3**). `ApplicationLayerSignalling` then calls its own `sendMessageToTerminal()`, passing in the retrieved `Terminal` as an argument (**4**). This method performs a `Terminal`-to-`TerminalConnection` lookup, retrieving the `TerminalConnection` which was originally created when the physical terminal first connected to the AS. Using this object and a call to its `sendMessage()` method (**5**), the ALS component ensures that the message is marshalled and sent to the correct terminal (**6**).

The second case of service logic signalling to a terminal starts with the service logic obtaining a reference to a representative `Terminal` object (**7**, **8**), since the terminal was not originally involved in the current message context. One way to obtain this reference is for the service logic to send the E.164 number associated to the terminal to the `UserManager`. This E.164 number is that which was associated to the terminal when it originally logged in. As is noted in the SD, the service logic may already have a `Terminal` reference to the terminal and so does not actually have to request a lookup from the `UserManager`. To send the message, the service logic calls `ApplicationLayerSignalling`'s `sendMessageToTerminal()` and passes in the `Terminal` object along with the desired remote terminal object, method and arguments (**9**). In the same way as the first case detailed above, this method then looks up the relevant `TerminalConnection` and calls its `sendMessage()` (**10**) method to put the actual marshalled message onto the wire (**11**).

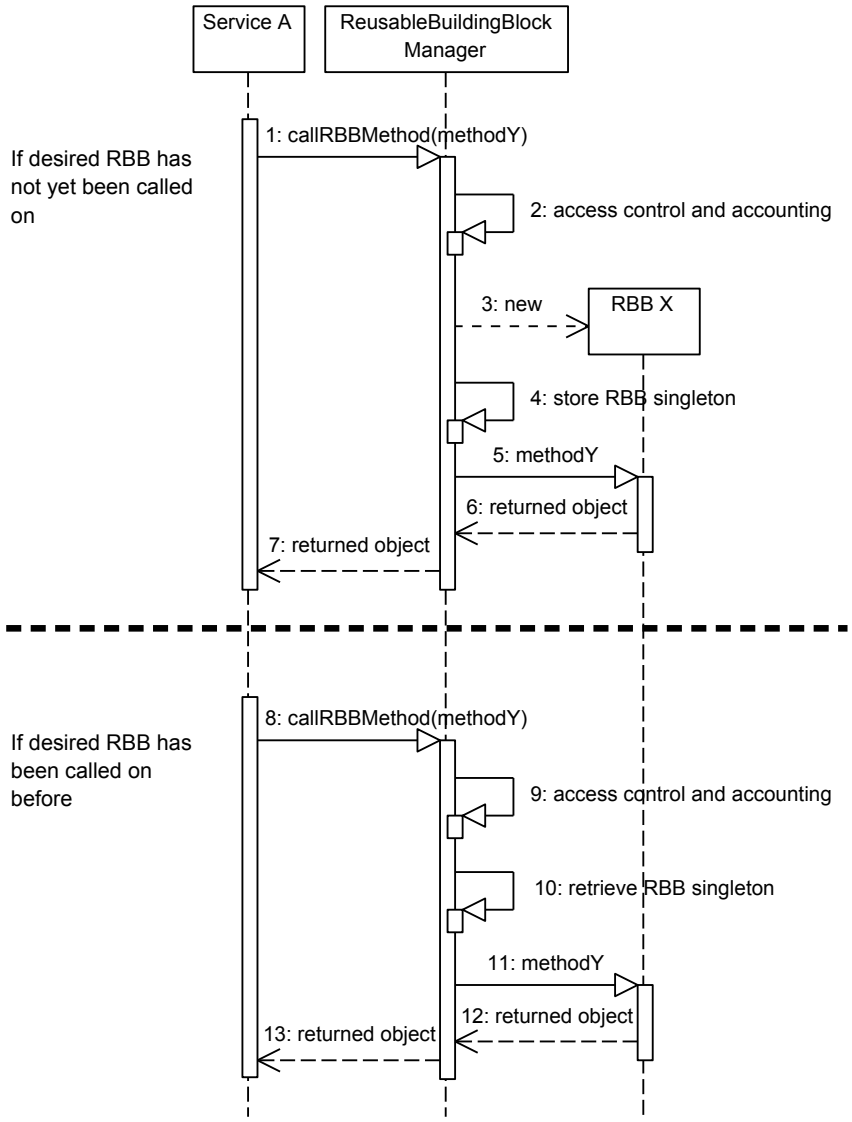


Figure 3.14: SD of a call to an RBB's method

RBB Method Invocation

Invoking RBB functionality is reminiscent of a terminal signalling to a service object in an AS. This is because RBBs are designed and managed using the same first-contact singleton object pattern as services. The SD for RBB method calls, shown in figure 3.14, is therefore similar to remote method calls to AS service objects except for three differences. Firstly, message context is not considered since RBBs cannot signal back directly to service logic. Secondly, RBB method calls originate only from service logic and not from terminals or other RBBs. Lastly, unlike service object methods, RBB methods can return a value since they exist locally to service logic. This last difference also results in method calls being synchronous from the point of

the view of the caller (in this case, service logic).

RBB method invocation also has two of the same scenarios as terminal-to-AS method calls. The first scenario occurs when the desired RBB has not been called on since the AS was started up. The other scenario covers what occurs when the RBB has been used before. Both scenarios are now detailed.

In both of the above RBB invocation scenarios, a method call request arrives at the `ReusableBuildingBlockManager` from some service logic (**1** and **8**). The request contains the targeted RBB object and method and a list of arguments for the method. The `ReusableBuildingBlockManager` determines whether the calling service logic has access rights to the desired RBB and method and performs any required accounting for the method call (**2** and **9**). At this point the sequence differs depending on the scenario. If the RBB has not yet been used, then its singleton is instantiated (**3**) and stored in the `ReusableBuildingBlockManager` (**4**). If it has been used since the AS was started up, then the singleton is retrieved (**10**). In both scenarios the method is then called synchronously using the arguments that were originally sent to the `ReusableBuildingBlockManager` (**5** and **11**). Once the method call completes and a resulting value is returned (**6** and **12**), this resulting value is relayed to the calling service logic (**7** and **13**) completing the dynamic RBB method invocation.

In other SDs, when service logic calls on RBB functionality, the method call will be shown simply as a single arrow directly from the service logic lifeline directly to the RBB's singleton object lifeline. The reader can now be aware that this single arrow encompasses all of the operations in the above SD.

Terminal Disconnection and Logoff

When discussing the ALS interface of the framework in section 3.3.2, we covered the fact that the disconnection of a terminal from an AS can occur in two ways: gracefully with a formal logoff before closing the physical connection and non-gracefully without logging off first. In both cases, the AS must ensure that a terminal's association to it is completely removed and any service session in which the terminal was involved is not left in an unstable state.

In the SD of figure 3.15 both scenarios are described and we show how even for a non-graceful disconnect the terminal's association is still fully removed and all services

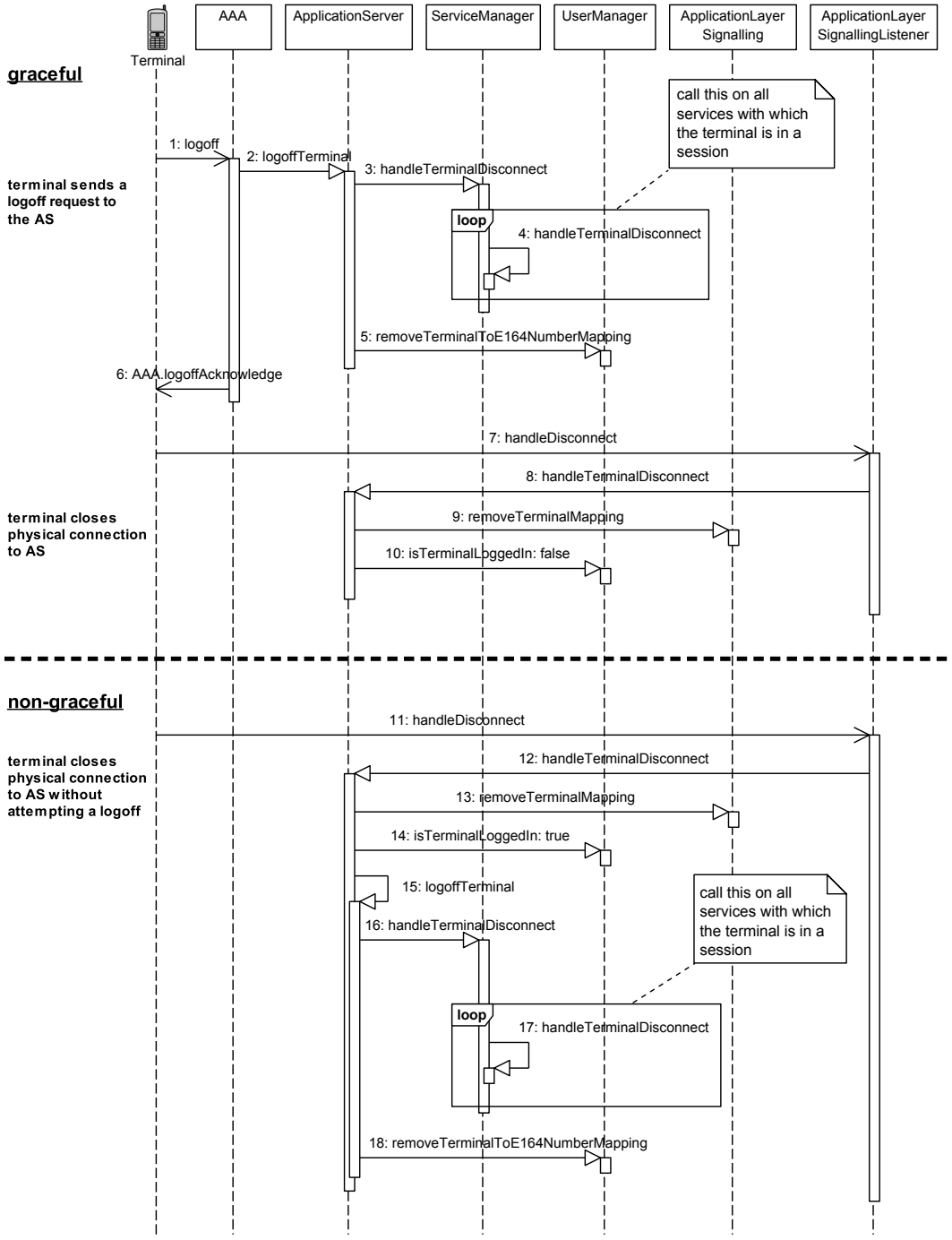


Figure 3.15: Terminal disconnection and logoff SD

are still given an opportunity to end their session with the terminal in question.

In the case of a graceful disconnect, the terminal first remotely calls `logoff()` on the `AAA` service object (**1**). The `ApplicationServer` object is then invoked to orchestrate the logoff process (**2**) by first calling `handleTerminalDisconnect()` on the `ServiceManager` (**3**). This causes the `ServiceManager` to iterate through all the service objects with which the terminal is involved in a session and calling `handleTerminalDisconnect()` on each of them (**4**), providing them with an opportunity to remove any reference to the terminal cleanly. Once this process is complete, a call to `removeTerminalToE164NumberMapping()` on the `UserManager` (**5**) removes the association of the terminal to the AS, indicating to the framework that the terminal is no longer logged in. The logoff process is then completed by remotely calling `logoffAcknowledge()` on the terminal's `AAA` object (**6**).

Once the terminal is aware that the logoff procedure is complete it physically disconnects itself from the AS, which invokes a `handleDisconnect()` on the `ApplicationLayerSignallingListener` (**7**). The `ApplicationServer` is then requested to orchestrate the removal of physical association of the terminal (**8**) which in this case simply means removing the `Terminal`-to-`TerminalConnection` mapping within the `ApplicationSignalling` object (**9**). The last method call of the process checks if the terminal has been logged off. It will always return a value of false in this scenario since the terminal has been logged off before the physical disconnection (**10**). The point of checking if the terminal is logged off is for the second scenario in which the terminal is physically disconnected *before* it is logged off of the AS.

The second scenario begins when a terminal unexpectedly physically disconnects itself from the AS. This triggers a `handleDisconnect()` event (**11**) which indicates that the `ApplicationServer` now has to orchestrate both a removal of the physical association to the terminal and the logging off of the terminal from the AS (**12**). The `Terminal`-to-`TerminalConnection` mapping is first removed (**13**) and then a check as to whether the terminal is logged in this circumstance indicates that the terminal is still logged in (**14**). This means that the process of logging off the terminal has to be performed in a similar way to the first scenario and is set off by calling `logoffTerminal()` (**15**).

First a call to `handleTerminalDisconnect()` on the `ServiceManager` (**16**) iteratively ends all service sessions with which the terminal was involved (**17**). To end the process, the terminal's association in the `UserManager` is removed (**18**). The logoff

process is obviously not acknowledged to the terminal since the physical connection to the terminal no longer exists.

3.3.6 Services

In describing the framework so far, we have covered much with regards to its facilities which support the operation of services. We now focus directly on services and in doing so give an outline of how service developers use the framework to implement service logic. To do this, we introduce an example service which integrates the functionality of the framework which has been covered so far.

Along with this functionality, the aim is also to show other essential functionality including demonstrating how service logic brings a terminal that was not originally involved in a service session, into this service session and giving an example of how application layer BCCM is actually implemented in service logic. In doing the latter we show how remote method calls are actually used to replace certain CS signals.

This essential service is presented from end-to-end, including the static (class diagram) and dynamic (SD) designs which dictate its operation.

Example Service: Two Party Call

We have already covered that the basic two party calling functionality of a terminal is implemented as a service in the ALS environment. As a means of demonstrating the implementation of services in the framework, we therefore use the relevant example of a two party call service.

This example service reuses the example two party call RBB from section 3.3.1 and also introduces a call state model RBB which encapsulates the FSM of figure 2.6. The FSM has to represent the current state of the two party call by having the implementation of the `TwoPartyCallEventListener` send transition requests to the call state model RBB. This means that the call model will only be used to show the current state of the call. For this example service the call model will not be used to check whether an operation on a call is valid or not, however it is important to note that in an actual implementation an operation would be disallowed if it would cause an invalid transition to occur.

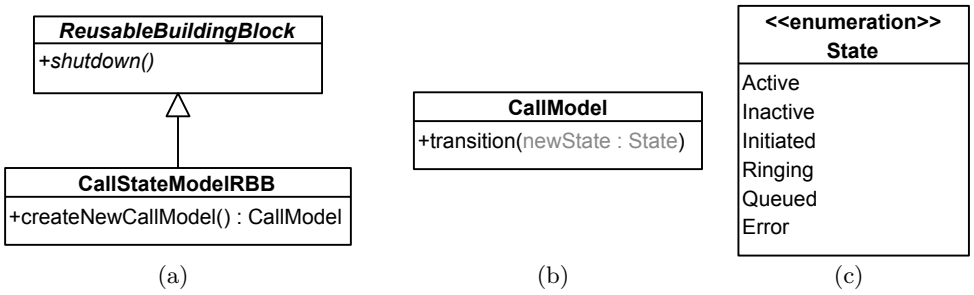


Figure 3.16: Classes of the call state model RBB

This example service also demonstrates another aspect of application layer BCCM. Traditionally, in an abstracted environment for a 3rd-party call service all signalling related to the setup of a call goes through the bearer network and the service logic only gets notified of the correct operation of the CS signalling as opposed to being involved in it. Throughout this report we have emphasised ALS which we take as far using it to replace CS signalling for certain parts of bearer connectivity. This whole aspect of ALS comes to the fore in this example where we show how a terminal signals in the application layer to indicate the fact that it is either ringing or has been answered, instead of the AS being alerted to these events via an RBB which acts as interface into underlying bearer network functionality.

Static Structure of The Two Party Call Service

Figure 3.16 is a representation of the classes involved in the call state model RBB. The RBB's main singleton (figure 3.16(a)) has a `createNewCallModel()` method, which is a factory method used to manufacture a `CallModel` object (figure 3.16(b)) for manipulation in service logic. The manufactured object can transition to one of the six states specified by the FSM implemented by the RBB by calling `transition()` and passing into it a `State` enumeration (figure 3.16(c)) value. In an actual implementation, this method would use exceptions to indicate an invalid state transition, a detail which we leave out here for the sake of simplicity.

Besides orchestrating two party calls, this two party call service facilitates the communication between the call state model and the two party call RBBs. Since the specification was laid down that RBBs cannot communicate directly with each other, an implementation of `TwoPartyCallEventListener` (figure 3.4(c)) is used to commuicate events in the two party call RBB to the call state model RBB by translating these events into state transitions which are then passed to the

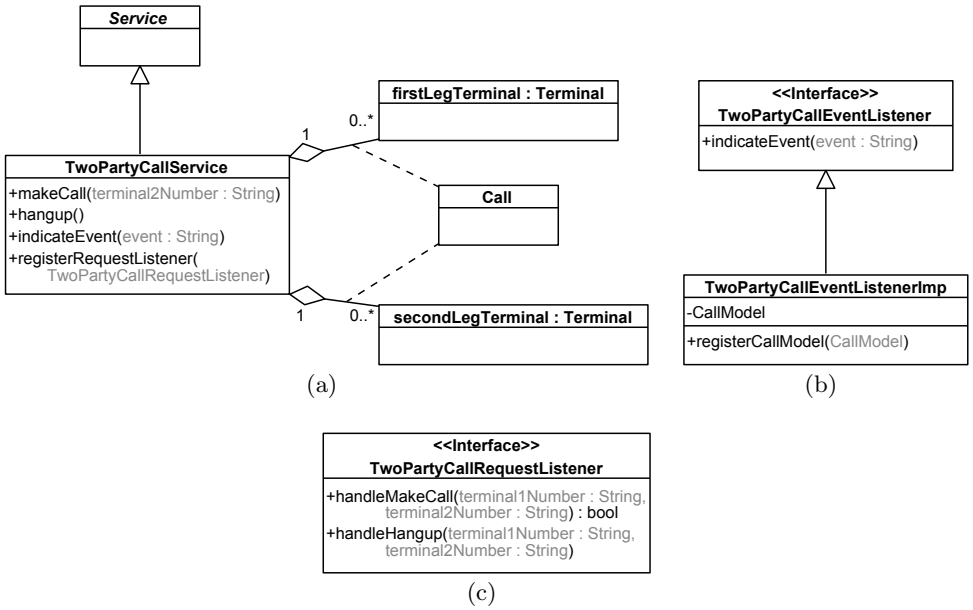


Figure 3.17: Classes of the two party call service. The `TwoPartyCallEventListener` interface and `Call` class are from the *two party call* RBB in figure 3.4.

`CallModel`'s `transition()` function.

The classes which actually make up the service are very simple. The service's first-contact class (figure 3.17(a)) contains the two methods representing the operations which a user would invoke on the service — `makeCall()` and `hangup()` — neither of which take arguments representing the originating terminal since its information can be retrieved using the framework's message context functionality. The third method, `indicateEvent()`, is that remotely accessible method of the service which allows for CS signalling to be replaced by ALS. This method takes as its argument a string representing an event that has occurred on the terminal as part of creating, maintaining or tearing down a two party call. This method passes on the event information to the `TwoPartyCallEventListener` implementation by calling `indicateEvent()` on that object. This means that the listener can be notified from two locations, since its `indicateEvent()` method can be called from both service logic and in response to an event occurring in the bearer network. The latter would occur, for example, after a failed attempt to contact a terminal through the bearer network.

The last method, `registerRequestListener()`, takes an implementation of the `TwoPartyCallRequestListener` (as shown in figure 3.17(c)) interface as an argument. The use of this method and the associated interface will be explained in section 4.2.

The `TwoPartyCallEventListenerImp` in figure 3.17(b) integrates the two RBBs. The class has a `registerCallModel()` method that allows the listener implementation to store a call model whose state represents the current state of the call of which the listener is being notified of events. This enables the `indicateEvent()` service logic above.

When a call is created, the two terminals involved in this call have to be bidirectionally mapped to it. The `Terminal`-to-`Call` mappings are required to support message context functionality such that the call associated to the terminal invoking a service method can be retrieved. `Call`-to-`Terminal` mappings assist in determining the other terminal involved in a call when one of the terminals makes a two party call service request. Figure 3.17(a) shows these mappings as two separate association class UML relationships.

Dynamic Operation of The Two Party Call Service

Two SDs demonstrate the main functions available to the two party call service: making and hanging up a call. Figure 3.18 shows the operation of requesting the setup of a call until the point where the bearer connection is actually setup. Figure 3.20 shows the service logic that gets executed when either terminal makes a request to hangup the call. Along with showing how service logic operates within the framework, these SDs clearly shows how ALS replaces certain parts of CS.

Making a call The operation of setting up a call in figure 3.18 is set in motion when terminal 1 makes a `makeCall()` remote method call (**1**). This method takes terminal 2's E.164 number as an argument. The `TwoPartyCallService` now needs to get two items which it does not yet have but which it requires for later use: terminal 1's E.164 number (to be passed to the `TwoPartyCallRBB` in order to setup the call) and terminal 2's object reference (which the service logic maps to a `Call` and also uses to notify terminal 2 to prepare itself for an incoming call). To obtain the latter the service makes a call to `UserManager`'s `getTerminalFromE164Number()` and passes in terminal 2's number that was sent from terminal 1 (**2, 3**). To obtain the former, the service uses the `getCurrentTerminal()` message context method to obtain the `Terminal` object representing terminal 1 (**4**) and then passes this object on to the `UserManager` to lookup terminal 1's E.164 number (**5**). The service then uses the reference to terminal 2 to signal to terminal 2 that it should ready itself to receive a phonecall (**6**).

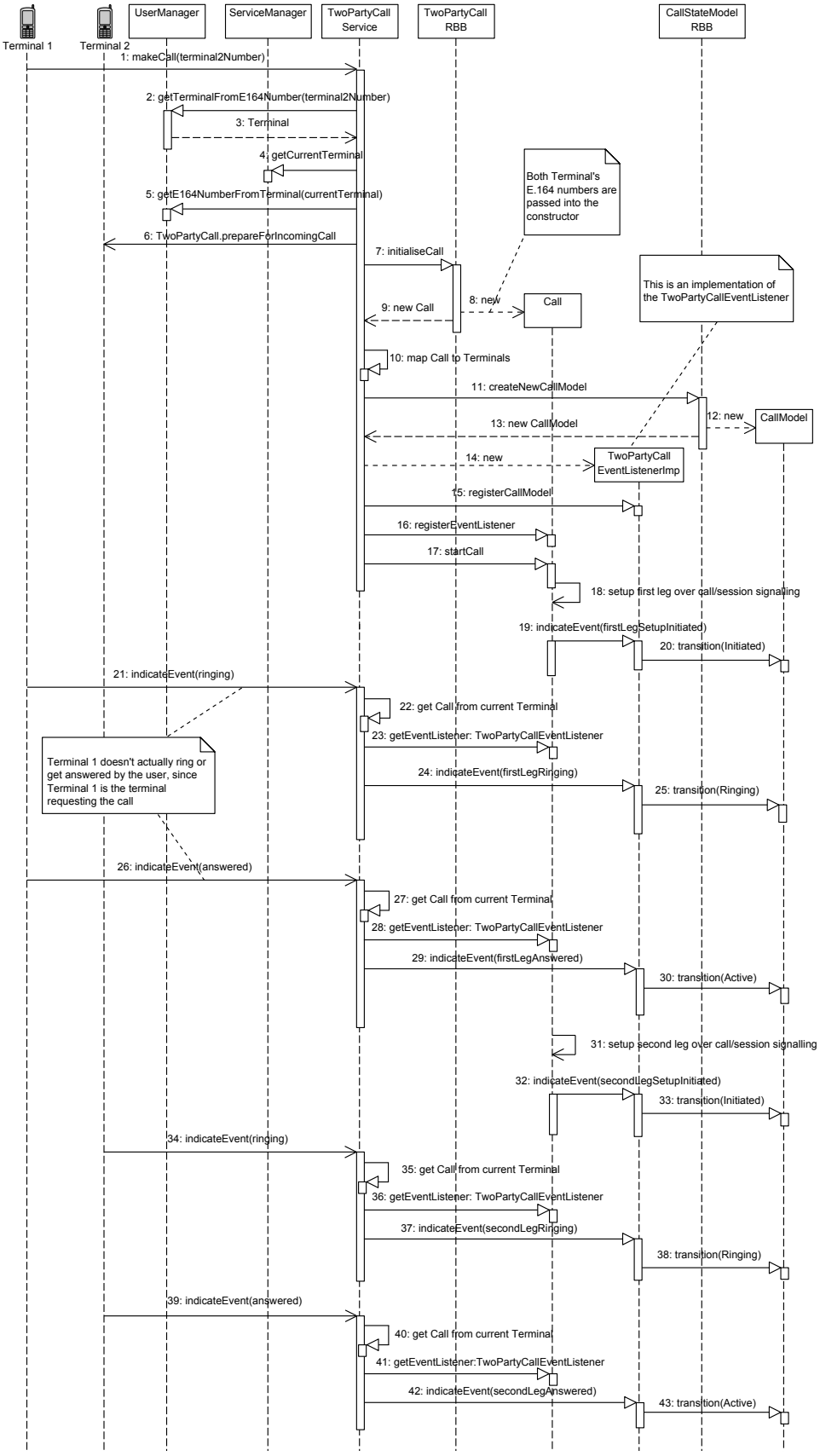


Figure 3.18: SD for the setup of a two party call

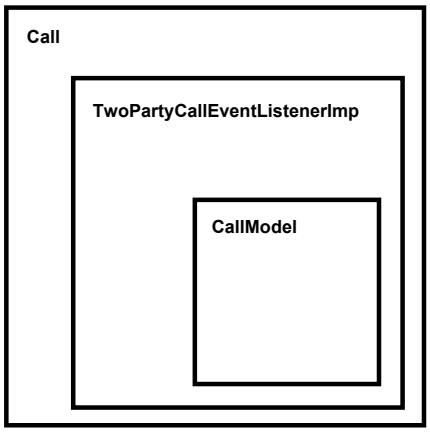


Figure 3.19: `CallModel` association to `Call`

The two party call RBB's functionality is now invoked. A `Call` object is created by passing the two terminals' numbers to the `initialiseCall()` RBB method (**7**–**9**). This object is then mapped for later retrieval to each terminals' E.164 numbers (**10**) as per the mapping of figure 3.17(a). Following this, a new `CallModel` object is created by the `CallStateModelRBB` (**11**–**13**). This model is associated to the `Call` by first instantiating the `TwoPartyCallEventListener` implementation (**14**) which allows for a `CallModel` to be registered with it (**15**). The event listener in turn is registered with the `Call` (**16**). This results in the setup shown in figure 3.19 where the `CallModel` is contained within the `TwoPartyCallEventListener` which is contained within the `Call`. With this setup, any events occurring during the call will trigger an event on the `TwoPartyCallEventListener` implementation which will update the `CallModel` accordingly.

At this point, the service is now ready to invoke the `TwoPartyCallRBB` to setup the call via the bearer network using a 3rd-party call setup. A call to the `Call`'s `startCall()` method starts this process (**17**). We do not show the details of the low-level CS signalling which occurs to setup the call (**18**), but the fact that the call setup has been initiated is indicated by firing `firstLegSetupInitiated` event on the `TwoPartyCallEventListener` (**19**) which updates the `CallModel` to be in the `Initiated` state (**20**).

Traditionally in a 3rd-party call, the originating terminal rings, then is answered by the calling party and then the call is routed to the destination terminal (while the calling party waits). When the called party answers, the call is fully setup. This situation changes in the case of using ALS to setup a call as it involves a 3rd-party call being setup using a 1st-party mechanism: the caller makes a call request from the terminal that is going to receive the call, but the request is directed towards

the AS which acts as a 3rd-party in the setup of the call between the two terminals. Despite 3rd-party call control being used, there is no reason for the originating terminal to ring or be answered. On the SD, the fact that the originating terminal indicates that it is ringing or answered is just to show the AS that it has been contacted via the bearer network and a bearer stream has been setup and also to ensure that the call model does not make any invalid state transitions. Therefore, once the terminal has been contacted over CS signalling it indicates a `ringing` event (**21**). The service then updates the call model by retrieving the `Call` object from the `Call`-to-`Terminal` mapping (**22**), obtaining the `Call`'s event listener (**23**) and then telling the event listener that the terminal of the first leg of the call has been contacted (indicated as a `firstLegRinging` event) (**24**) so that it knows to update the `CallModel` to the `Ringing` state (**25**). Once the bearer stream has been setup on the originating terminal, it sends an `answered` event to the AS (**26**). This makes the service follow the same steps preceding the `ringing` event above, but this time to make the `CallModel` transition to the `Active` state (**27**–**30**).

Using CS signalling, the two party call RBB continues to setup the second leg of the call (**31**). The setup of the second leg does not need to be explicitly requested by the service since the earlier `startCall()` request (**17**) is a request to setup the entire call and not just the first leg. The setup of the second leg over CS signalling is therefore shown as an asynchronous message to indicate that it occurs independently of the service logic. Once this second leg setup has begun, a `secondLegSetupInitiated` event is signalled to the `TwoPartyCallListener` (**32**) so that the `CallModel` can transition to the `Initiated` state (**33**).

For the second leg of the call a similar set of events occurs as for the first in order to keep the `CallModel` updated. However, in this case, the destination terminal does actually ring and is actually answered by the called party. Thus, when the terminal rings, it indicates a `ringing` event over ALS (**34**) which results in all the steps being taken to update the `CallModel` to a `Ringing` state (**35**–**38**). When the terminal is answered it indicates an `answered` event (**39**) which results in the `CallModel` transitioning into the `Active` state (**40**–**43**). At this point the call is completely setup and the voice call can proceed.

Hanging Up a Call In response to a hangup request from either party in the call, a `hangupCall()` remote method invocation on the `TwoPartyCallService` object (**1**) begins the process of hanging up a call shown in figure 3.20 (in the case of this SD, *Terminal 2* is the requesting terminal and thus referred to as the *first leg* in

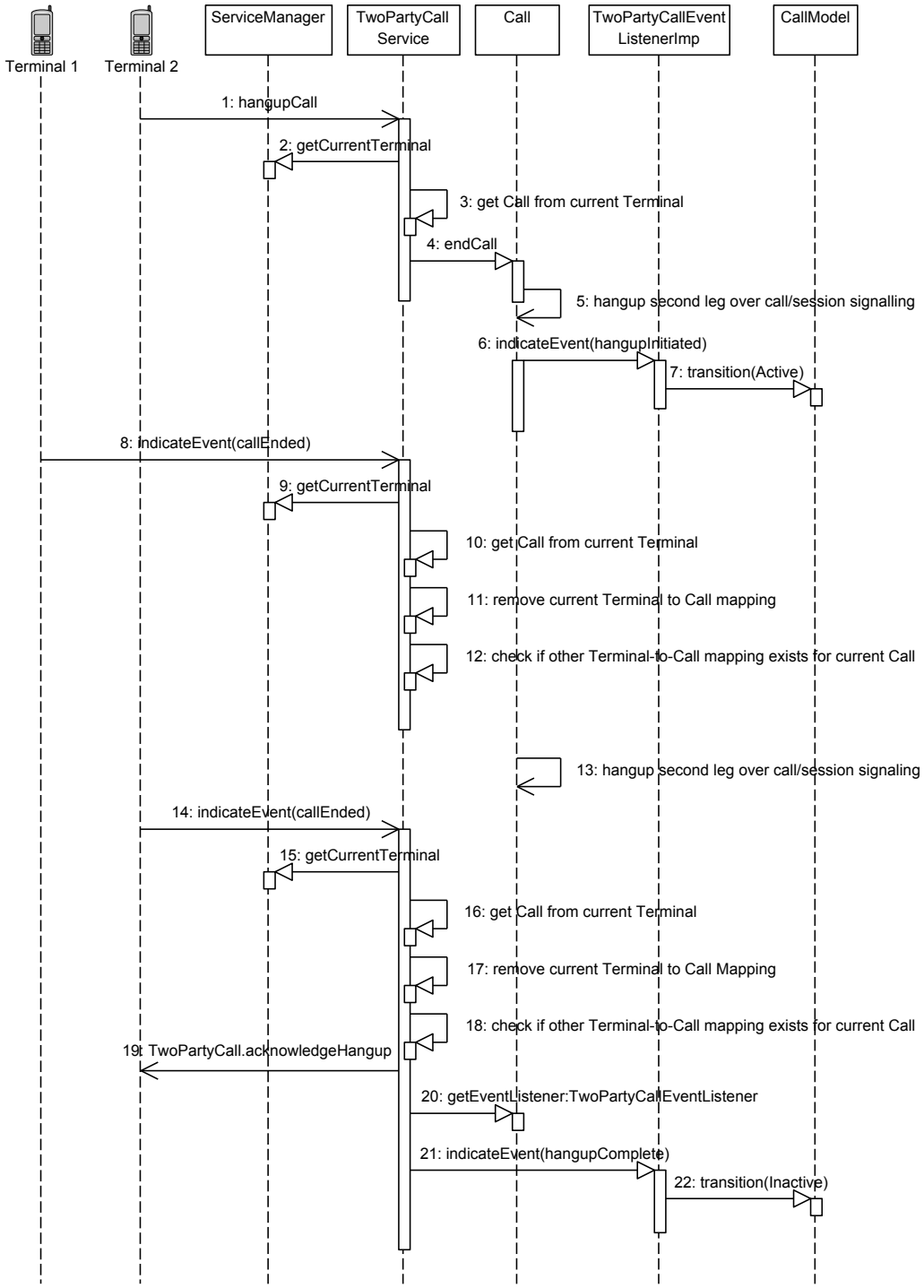


Figure 3.20: SD for hanging up a two party call

the following description). The `TwoPartyCall` service firstly obtains the current message context `Terminal` (**2**) and uses this to retrieve the `Call` object representing the call in which the requesting terminal is involved (**3**). By calling `endCall()` on the retrieved `Call` object, the service then requests from the two party call RBB that it terminate the bearer connection (**4**). Underlying bearer network functionality to disconnect the call is invoked (shown by an asynchronous arrow in the SD), starting with disconnecting the second leg of the call (**5**). The `CallModel` is then updated by indicating a `hangupInitiated` event on the `TwoPartyCallEventListener` (**6**) which in turns tells the `CallModel` to transition to the `Active` state (**7**).

Once the second leg's bearer connection has been ended, it indicates this as a `callEnded` event (**8**). The service now needs to determine if the call has been ended completely so that it can update the `CallModel` accordingly and delete any objects which are no longer required. To do this, the service obtains the current reference object of the terminal which just indicated that its bearer connection ended (**9**). The `Call` object representing the current call is retrieved from the `Terminal`-to-`Call` mapping (**10**) before the mapping to the `Terminal` of the first leg is removed (**11**). This `Call` object is then used to determine if there are other call legs involved in the call (represented by the other `Terminal`-to-`Call`) (**12**), which at this point is not the case, since the first leg has not yet been disconnected, so no further service logic occurs at this point.

The RBB then continues to invoke bearer network functionality to disconnect the first leg of the call, which is shown simply as a single asynchronous message (**13**). When the first leg's bearer connection ends, it sends a `callEnded` event message (**14**). In the same manner as the second leg, the service logic obtains the current message context `Terminal` (**15**), its associated `Call` (**16**) and then removes the association (**17**). This time, however, the association is the last to the current `Call`, so the service logic knows that the call has ended completely. In response to this, it sends an hangup acknowledgement to the `TwoPartyCall` object of the terminal that originally requested the hangup (**19**). The service logic then puts the `CallModel` into the `Inactive` state by retrieving the `TwoPartyCallEventListener` (**20**), sending it a `hangupComplete` event (**21**) so that it in turn tells the `CallModel` to perform the transition (**22**).

3.4 Conclusion

Both the static structure and dynamic operation of the framework have been presented, giving a thorough description of the manner in which ALS enhances and simplifies service development. Very simple method calls allow sending messages to terminals and an intuitive inheritance hierarchy ensures that messages received from terminals can be processed within service logic. This functionality is supported by dynamic method invocation and message context, ensuring that service logic can be invoked without having previously been compiled into the system and allowing service logic to identify an invoking terminal respectively.

To add to this, the framework was developed to support those features necessary for fully fledged services including RBBs, user identity and AAA functionality. Further, the details of various manager components revealed how the framework manages the co-operation of the various components of the framework and ensures that service logic programmed by 3rd-party developers is appropriately decoupled from the remainder of the framework by having various details of the framework abstracted but still ensuring that service logic has sufficient access to operate correctly.

The framework was presented in UML form to support its OO design. Along with class diagrams, various SDs gave the dynamic operation of the main processes which occur in the system, including terminal login and logoff, remote method invocation from the terminal to the AS and dynamic method invocation. Finally, two SDs presented a two party call service, combining all the elements of the framework and also revealing how the framework supports BCCM in the application layer.

All the above elements bring together the various principles and design choices which were selected to guide the design of the framework. This framework confirmed OO as very applicable to telecommunications services, whilst the importance of reusability was realised as RBBs. As mentioned above, dynamic method invocation and message context were pivotal to simplifying the mechanisms provided to programmers for asynchronously accessing remote service logic. Whilst the actual signalling protocol supporting this was not finely detailed, a higher-level overview was adequate to demonstrate the manner in which remote method invocations are sent between terminal and AS allowing for direct application-to-application signalling. The framework was also defined to cater for lower-level network functionality abstraction by showing how RBBs are used to interface with the bearer network and give programmers intuitive control over its functionality.

The framework has therefore been presented in its entirety. A proof of concept implementation of it is discussed in appendix A. Chapter 4 then builds on it, by presenting three services which show how the framework deals with non-call related services as well demonstrating how a call-based service interacts with other services.

Chapter 4

Example Services

With the framework adequately defined, its utility in the development of services can be explored. This is done by developing three example services within the framework. The intention of this chapter is to make full use of all the features of the framework to show how it acts as a foundation for simple service development, whilst providing flexibility in the design of these services and supporting them with a robust platform on which to execute.

Different kinds of services are presented. A simple presence service demonstrates a service operating only in the application layer and also how the framework is applicable to modern services such as presence. This presence service is then combined with the *two party call* service of section 3.3.6. Finally, a conference call service is presented as a more complex BCCM-based service.

4.1 Presence Service

A *presence service* can be defined as a "software system whose role is to collect and disseminate Presence Information, subject to a wide variety of controls" [30]. *Presence information* is the "dynamic set of information pertaining to a presentity that may include Presence Information Elements such as the status, reachability, willingness, and capabilities of that Presentity" [30]. In this case, *presence information elements* are the basic unit of information in a presence service and a *presentity* is a "logical entity that has Presence Information associated with it" [30]. In summary, *presence* is "a set of characteristics describing the context in which the user, or an agent representing the user, exists" [8, pp 364] and a *presence system* "allows users to subscribe to each other and be notified of changes in state" [31].

Presence state information can be static or dynamic – terminal capabilities and availability being an example of each respectively. For this service, only *availability* is considered as a presence information element, realising that if the service operates correctly then it can be easily extended to incorporate other information elements like location and reachability.

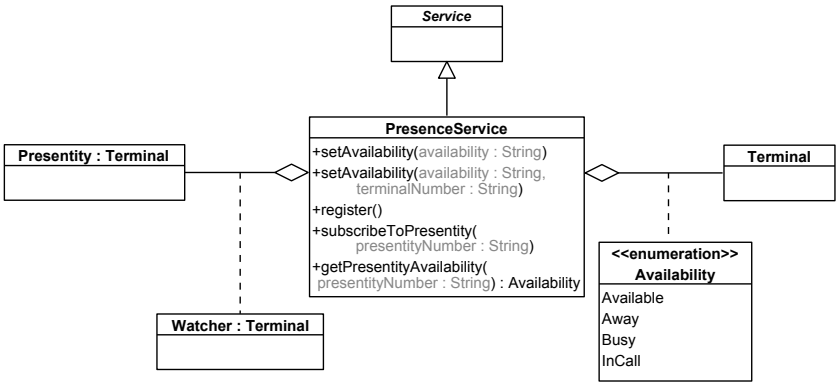
In the context of this service, certain terms need to be defined. Two definitions here are based on those in [31]. A *presentity* is an entity that provides presence information about itself that other entities may be interested in. A *watcher* is an entity that is interested in presence information of other entities and either subscribes to presence updates or when required, fetches current presence states from these entities. For this service we are only dealing with subscription-based presence updates, which leads to two other terms, defined specifically for the purposes of this report: *subscriber* and *subscription*. We define a *subscriber* as a watcher that is registered to receive updates from presentities. A *subscription* is an actual presentity from which a watcher will receive updates.

4.1.1 Functionality and Structure

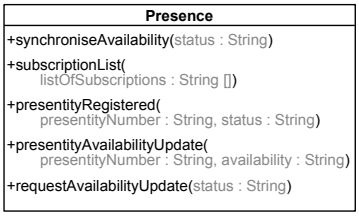
The presence service presented here offers the following functions:

- description of a presentity's current availability as either *available*, *away*, *busy* or *in call* (the last one being reserved for use when the two party call service is integrated with this service in section 4.2)
- subscription to presentity status updates
- database persistence of status and subscriptions (by involving a database RBB)
- alerts to subscribers when a presentity (in which they have interest) registers to the system

The service, with its static structure shown in figure 4.1, is made up of a `PresenceService` first-contact service object which stores two mappings: between subscriber and subscription terminals and between terminals and current availability state. Both of the mappings are represented by the association classes shown in figure 4.1(a). The former mapping is bi-directional in that it allows the service to determine all subscriptions for a particular watcher and all subscribers to a particular presentity. An example of this mapping is shown in figure 4.2. The latter mapping is



(a) main AS classes



(b) terminal classes

Figure 4.1: Presence service classes

simply to associate a terminal to a particular availability state. In a presence service which catered for more detailed state descriptions, terminals would be associated to an object that would encapsulate various presence information elements. For this service only availability is catered for in the form of an `Availability` enumeration class which represents each of the states which the presence service makes available to a terminal.

The main AS `PresenceService` object, has four methods which support the main features of the framework: allowing service logic to update the presence status of a terminal, allowing a presentity to update its own presence state to the service, registering to the service and subscribing to a presentity's updates. Each of these functions will be expanded out into SDs. The fifth method which is not callable from a remote terminal, `getPresentityAvailability()`, returns a value and is used by other services to determine a specific terminal's presence state. Its use will become clear in section 4.2 when the presence service is combined with the two party call service. The SDs below also show that the service calls remote methods on an object existing in the terminal. The use of each of the methods of the `Presence` terminal object in figure 4.1(b) will be clarified in the following section.

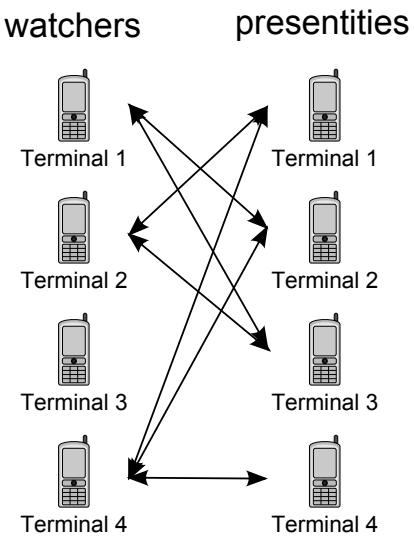


Figure 4.2: Example subscription mappings. Bi-directionality of arrows shows how subscribers can be retrieved from a subscription and vice versa. Note that a terminal can be both a watcher and presentity.

4.1.2 Dynamic Operation

When a terminal associates itself to the presence service this fact is broadcasted to all the terminals who are subscribed to its presence updates. On association it must also be mapped to all of its subscriptions as described in figure 4.3. These and the service's other main functions are detailed in the following SDs.

As shown in the SDs, the terminal's subscriptions and associated subscribers are not kept in memory while the terminal is not registered to the service. Instead these are stored in a database using a database RBB, which will not be formally defined in this report. Instead, methods that are invoked on this RBB simply represent the database functionality that should be catered for and so should not be considered as actual methods for implementation. A proper database RBB would be more general purpose and not tied only to presence functionality as it is here. The SDs below use this arbitrary database RBB to show at what points the database is accessed.

Registering

The process of registering to the service begins in figure 4.3 when a terminal (in this case, Terminal 1) calls the `register()` remote method (**1**). In response to this, the service must first obtain a reference to Terminal 1 so that it can communicate back to

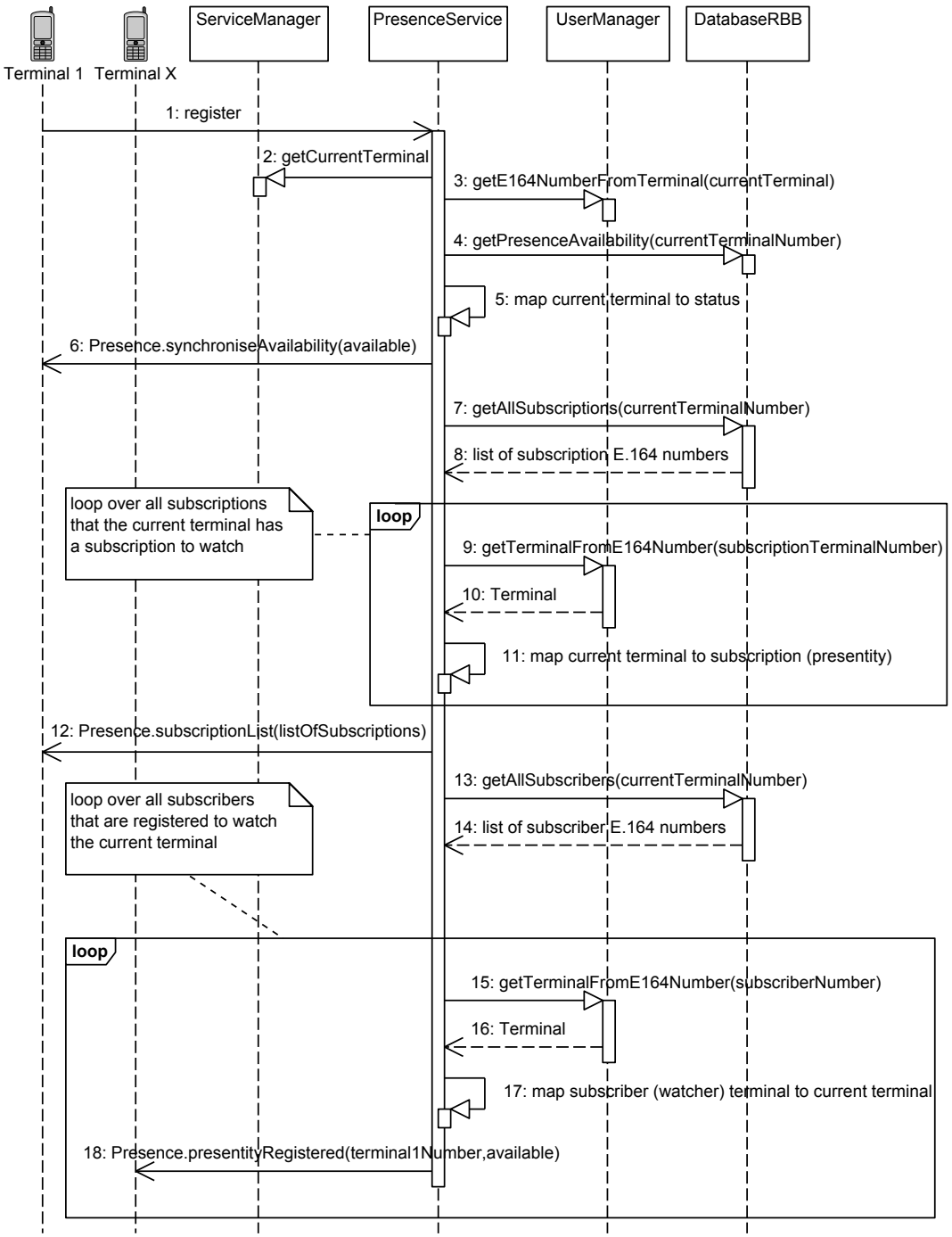


Figure 4.3: Presence service registration SD. Terminal X represents each of the subscriber terminals to Terminal 1.

the terminal and also perform the various mappings crucial to the operation of this service (**2**). All information relevant to the service which can be associated to specific users is persisted in the database using E.164 numbers as an index value. The current terminal's E.164 number is therefore at this point retrieved from the `UserManager` (**3**) since this registration process will involve interaction with the database. The service is capable of storing a user's last availability status and since the terminal may have been previously registered to the service, this last availability state is retrieved from the database (**4**) (the point where the last availability was actually persisted in the database is indicated later in the SD which describes a terminal updating its availability). The availability value retrieved from the database is mapped to the current terminal (**5**) and sent to the terminal so that it can synchronise its status with the service (**6**).

Two tasks are now undertaken in the registration process. The first is to map the registering terminal to all its subscriptions and then send the list of these subscriptions to the terminal. The second is to map all of the terminal's subscribers to itself and then broadcast the fact that this terminal has just registered along with its availability status to each of the subscribers.

The first task begins when the `PresenceService` requests a list of all the current terminal's subscriptions from the database RBB (**7, 8**). The service logic then iterates through this list, retrieving the associated `Terminal` object from the `UserManager` for each E.164 number in the list (**9, 10**) and then storing a subscription mapping of this `Terminal` to the representative object for the current terminal (**11**). Then the list of E.164 numbers is relayed to the current terminal (**12**) so that it too will have knowledge of the states of all its subscriptions.

The second task similarly retrieves a list of the all the current terminal's subscribers which have previously been persisted in the database (**13, 14**). By iterating through this list, the service logic retrieves each terminal associated to the E.164 numbers in the list (**15, 16**) and then maps each as a subscriber to the current terminal (**17**). Finally, a remote method call on each subscriber terminal (represented by Terminal X in figure 4.3) indicates that the current terminal (Terminal 1) has just registered to the presence service (**18**).

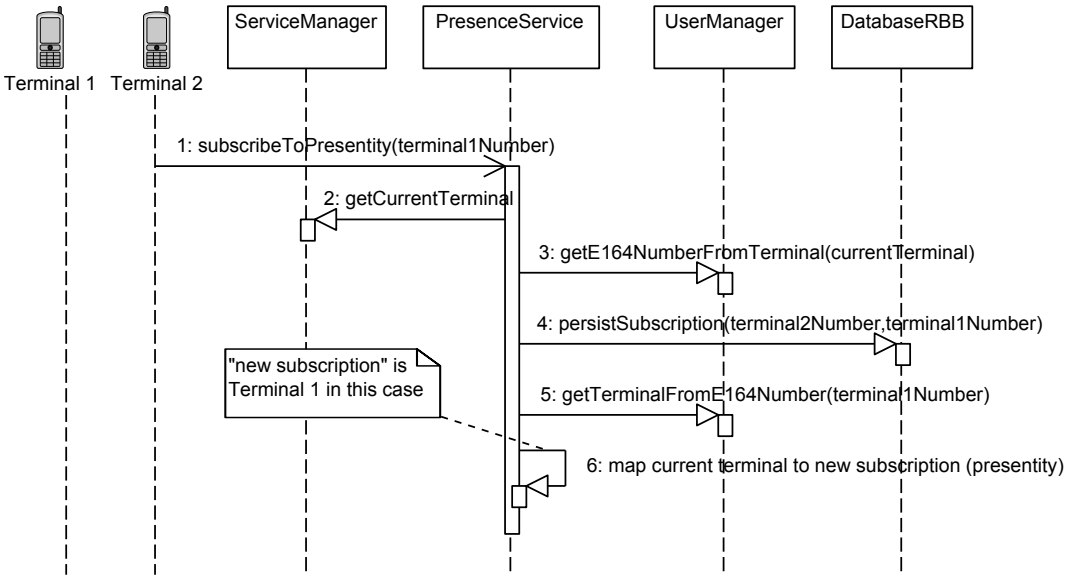


Figure 4.4: SD describing the subscription to a terminal's presence updates.

Subscribe to a Presentity

Subscription to a terminal's status updates is a simple process that involves storing the subscription in the database and mapping the requesting terminal as a subscriber to the terminal which it is interested in. Figure 4.4 details these steps.

After the requesting terminal remotely calls the `subscribeToPresentity()` method of the `PresenceService` object (**1**), expressing its interest in the terminal represented by the E.164 number which it sends as the `presentityNumber` parameter, the service logic retrieves the reference object of the requesting terminal (**2**). The reference object is used to retrieve the requesting terminal's E.164 number (**3**) which in turn is persisted along with the subscription terminal's E.164 number. These two pieces of data together represent the new subscription relationship in the database (**4**). The subscription terminal's representative object is then retrieved from the `UserManager` (**5**) allowing for a mapping to be created between the subscriber and subscription (**6**).

Update Presence Availability Status

When a terminal user wishes to update a terminal's current service, he or she executes a command on the terminal which initiates a request from the terminal to the AS to effect this update. The request is made in the form of a `setAvailability()`

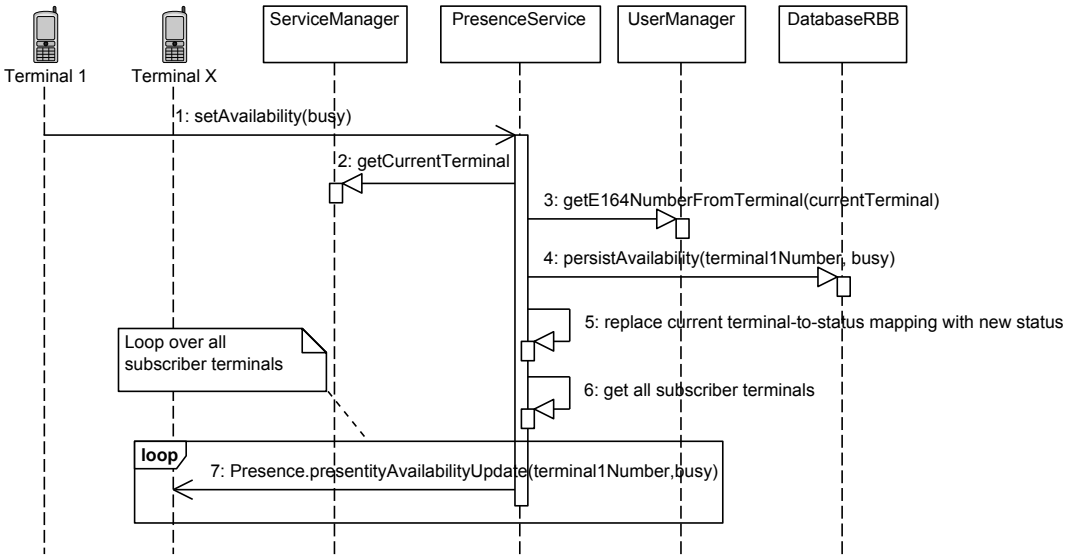


Figure 4.5: SD describing the process of a terminal updating its availability status

remote method call on the `PresenceService` object (**1**) as per figure 4.5. This request is made with one of the possible availability statuses (specified by the `Availabilty` enumeration) included as an argument.

The current message context terminal is retrieved (**2**) and used to obtain its E.164 number from the `UserManager` (**3**). The service logic then interfaces with the database RBB to persist the new availability status to the database (**4**). The current `Availability`-to-`Terminal` mapping is updated to represent the new status (**5**).

All subscribers to the current terminal have to be notified of the status update. This is achieved by retrieving the list of all the current terminal's subscribers (**6**) and iterating through the list, calling the `presentityAvailabilityUpdate()` method on each of them (**7**).

The service also supports the updating of presence from other service logic. The `setAvailability()` method has been overloaded to also take a E.164 number as an argument. Calling this version of the method from within the application server results in the server making a request to the terminal to update its presence so that the logic of the SD of figure 4.5 can be reused. This is described in figure 4.6. The utility of this method will become apparent in section 4.2, where the presence service is integrated with another service.

When service logic requires that a terminal update its presence status, it calls `setAvailability()` with the terminal's E.164 number as an argument (**1**). This

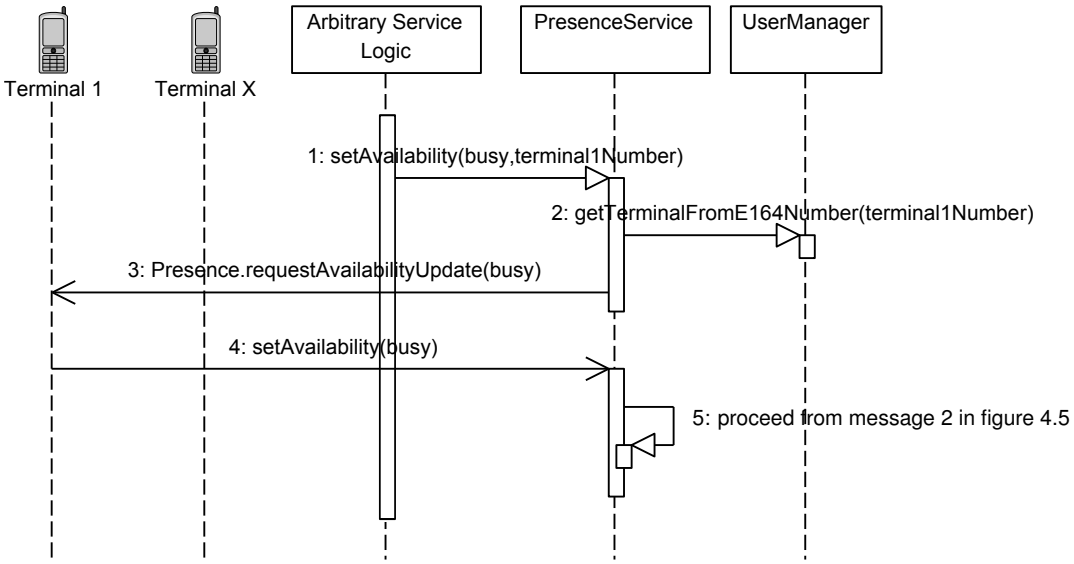


Figure 4.6: SD describing the updating process for a service requesting that a terminal update its availability status

makes the service retrieve the terminal's representative object (**2**) and send a `requestAvailabilityUpdate()` with the desired status as an argument (**3**). The terminal responds to this by remotely calling the `setAvailability()` method (that does not take a E.164 number as an argument) (**4**). This makes the `PresenceService` orchestrate the same presence updating process as figure 4.5 (**5**), resulting in the terminal and AS having consistent availability information about the terminal's new status.

4.2 Two Party Call Service with Presence

Two services have so far been designed for implementation within the framework, namely a two party call service and a presence service. The aim is now to join these two services to build up complex service functionality. The purpose of this is twofold. Firstly we want to demonstrate how complex service functionality is actually implemented very simply. Secondly, services are not as useful in isolation and so we wish to show how services are combined.

In the legacy IN, service logic was executed at specific points (detection points (DP)) in the processing of a call. These DPs represented state transitions of the BCSM, and allowed service logic to either only be notified of the state change, or actually be requested to perform some function before call processing would continue [32].

In integrating the two services to build up a more complex service, we take a similar approach since we are integrating a call service. Whilst not formally implementing detection points, hooks, which serve the same purpose, are introduced into the two party call service. These hooks act in the same way as event detection points (EDP) in the IN. They may function like an EDP-R (request type), that is, call processing stops until these triggers have been handled and call processing may be affected by the outcome of the invoked service logic. They may also function like an EDP-N (notification type) in which external service logic is only notified of a state transition in the call model but does not affect call processing.

In the implementation in this section the number and types of hooks certainly do not match those of the IN's BCSM, but we are only aiming to show how two services co-operate, and not to provide the same generalised functionality as the IN call processing. It is also important to note that the presence service can still operate in isolation. The fact that we are able to combine two or more services shows the flexibility of the framework and how it can be used to orchestrate complex functionality.

4.2.1 Functionality and Structure

Section 3.3.6 mentioned the `TwoPartyCallRequestListener` interface along with the `registerRequestListener()` method of the `TwoPartyCallService` class. Their use is specifically for integrating the two party call service with other service logic. `TwoPartyCallRequestListener` specifies the methods which act as the "hooks" that were mentioned above. These methods get called on soon after a terminal makes a request to either make or hangup a call. The methods serve two functions, firstly to perform some sort of service logic before the call processing continues and then, in the case of making a call, to return a boolean value indicating whether call processing should continue.

The high-level functionality of the combined service logic is to allow the presence service to determine whether a call can be made or not. If a user has set his or her terminal's presence availability to `Busy`, this should indicate to other users that this user is not able to accept calls currently. By combining the two party call service to the presence service, call restrictions can be enforced based on presence availability. Thus, if a user tries to call another whose availability is `Busy`, the call will not be setup. This is taken further by specifying that when a two party call is setup, its availability is automatically set to `InCall`. Then if another user tries to initiate a

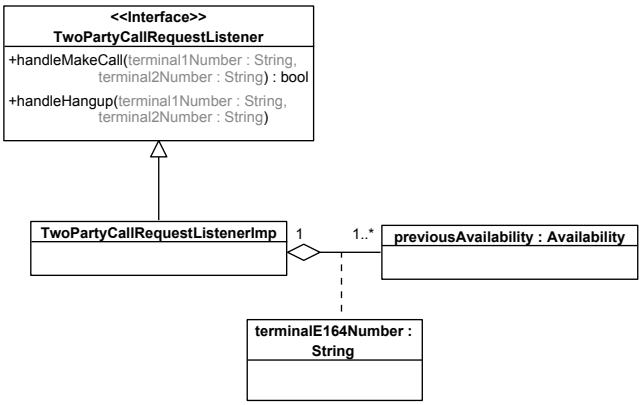


Figure 4.7: Classes required for the integration of the two party call and presence services

call with a user that is already in a call, the call setup will fail immediately, reducing the amount of signalling required to determine that the call cannot be setup. When the call is hungup, both terminals return to the availability state in which they were before the call began.

The combination of the two services which facilitates the above functionality relies on the classes shown in figure 4.7. The `TwoPartyCallRequestListener` implementation overrides the `handleMakeCall()` and `handleHangup()` methods, allowing it to execute some service logic when a request to make or hangup a call is made. These two methods take the E.164 numbers of the originating and destination terminals as arguments which can be used by the service logic invoked by these hooks. The `TwoPartyCallRequestListener` implementation also stores a mapping between terminal E.164 numbers (which the presence service uses to identify terminals uniquely) and previous availability states to provide a means for the terminals to return to their previous states once a call is complete. The purpose of this mapping will be elucidated in the following section.

4.2.2 Dynamic Operation

The integration requires modifications to the SDs of figures 3.18 (`makeCall()`) and 3.20 (`hangupCall()`). The change to the `makeCall()` SD involves inserting an invocation of `handleMakeCall()` just before initialising the call in the bearer network (via the `TwoPartyCallRBB`). Thus, this invocation is inserted after message **5** of figure 3.18. The changes along with the invocation of the `PresenceService` functionality are shown in figure 4.8 and we now run through each of the messages

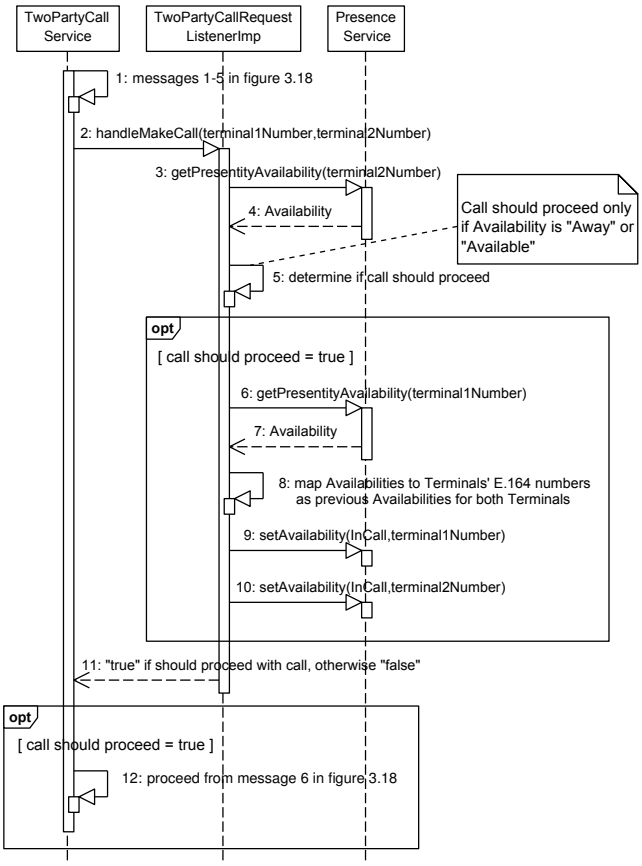


Figure 4.8: Using `handleMakeCall()` hook to integrate two party call and presence services

in this figure.

In response to receiving a `makeCall()` invocation from the terminal, the E.164 numbers of both terminals are retrieved (**1**). At this point the SD of figure 3.18 is modified, by inserting an invocation of the `handleMakeCall()` hook (**2**). By the programmed behaviour of the `TwoPartyCallRequestListener` implementation, this gets the availability of the destination terminal (**3, 4**) and then determines if the call setup should proceed according to the functionality of the service stated in section 4.2.1. That is, it should proceed only if the terminal's availability is not `Busy` or `InCall` (**5**). Based on this information, if the service logic determines that the call should proceed, it obtains the availability of the originating terminal as well (**6, 7**) and then stores the current availabilities of both terminals as previous availabilities in the terminal E.164 number to previous `Availability` mapping shown in figure 4.7 (**8**). Having backed up the previous availability states, both terminal's availabilities are now set to `InCall` (**9, 10**), preventing other terminals from contacting them. The `setAvailability()` method used is that version of the overloaded method which

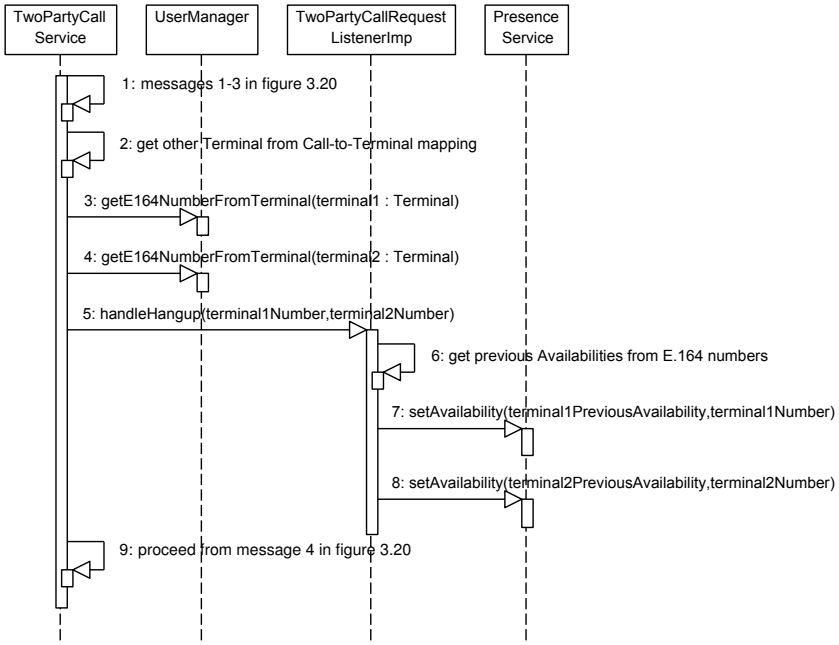


Figure 4.9: Using `handleHangup()` hook to integrate two party call and presence services

can be called from within an AS from other service logic as discussed in section 4.1.2.

If it was determined in message **5** that call processing should proceed, the `handleMakeCall()` hook returns a boolean value of *true* (**11**), indicating to the `TwoPartyCallService` that it should continue call processing as per figure 3.18 (**12**). Otherwise, a *false* value is returned and the call processing ends, preventing the call from being setup.

The `hangupCall()` SD of figure 3.20 is also modified just before it invokes bearer network functionality via the `TwoPartyCallRBB`. This modification and the integration point with the `PresenceService` at the point where a call is hung up is shown in figure 4.9. As is similar to the `TwoPartyCallService` in isolation, hanging up a call begins with the terminal invoking a `hangupCall()` remote method call and the service logic obtaining a reference to the current `Terminal` and the `Call` object representing the two party call in which the terminal is involved (**1**). Since, the service logic now requires the E.164 numbers of both terminals to interface with the `PresenceService`, but in the original SD these are not required, the modification is made at this point. Here, the `Call` is used to retrieve the other `Terminal` involved in the call (**2**). Then the E.164 numbers are retrieved using both `Terminal` objects (**3, 4**). The `handleHangup()` hook is invoked (**5**) to orchestrate the service logic of

returning both terminals to the availability state they were in before the call was setup. The previous availabilities are retrieved from the terminal E.164 number to the previous `Availability` mapping (**6**) and then set as the current availabilities for each terminal (**7, 8**). Call processing for hanging up a call then proceeds as before in the original `hangupCall()` SD (**9**). Note that this hook does not return a value, and merely sends a notification to the service logic which it invokes, thus call processing will continue irrespective of the invoked service logic.

The two services could be integrated further, for example, by allowing a user to request that a call automatically be setup with another user when that user's availability changes to `Available`. However, we have adequately shown how two services co-operate (whilst making sure that they are not too closely coupled) and so do not pursue further integration possibilities.

One last point to note is that, unlike in the IN where triggers are executed based on transitions of the BCSM, service logic external to the two party call service is not executed on transitions of a `CallModel` object. The integration could have been done in this way, but we have chosen to keep the `CallModel` as exactly what it is, a model, and not as a means for triggering other service logic. More so, the `CallStateModelRBB` does not cater for triggering external service logic and so it was simpler to add hooks into the `TwoPartyCallService` directly.

4.3 Multiparty Call Control RBB

Before presenting the design of a conference call service, a multiparty call RBB will be introduced to serve a similar supporting function that the two party call RBB does for the two party call service. That is, it will provide a simplified interface into the bearer network that will allow for the setup of calls involving more than two parties. With this in place, the conference call service will reuse its functionality and also provide more complex features that are required of conference calls. This is done in the same way that the OSA/Parlay conference call control service capability feature (SCF) [33] is supported by and extends the multi-party call control SCF [34].

The multiparty call RBB will enable service logic to create a call in the bearer network and then add multiple legs to this call. Further it will provide the same listener functionality as that of the two party call RBB, in that each leg of the call

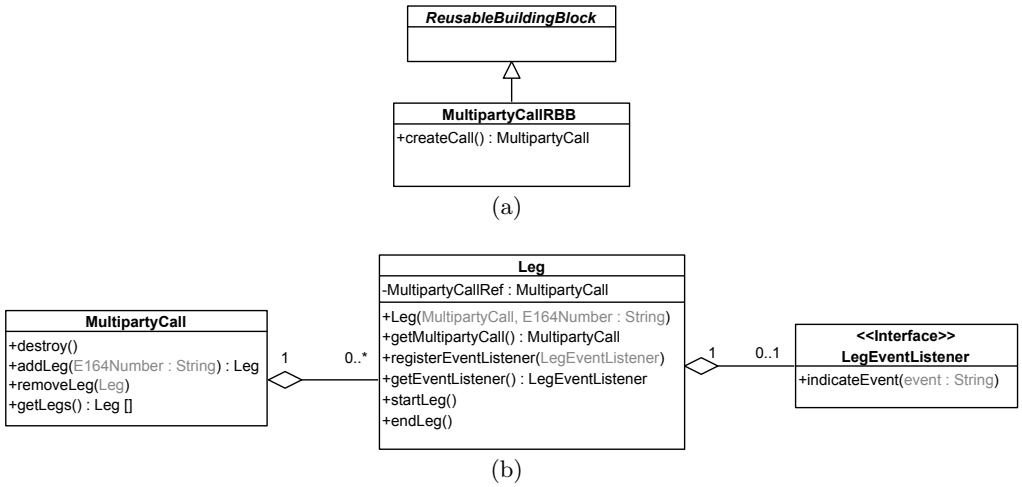


Figure 4.10: Multiparty call RBB classes

will be able to have an event listener registered with it. This can and will be used to integrate this RBB with the call state model RBB allowing for each leg of the call to maintain its own call model.

4.3.1 Functionality and Structure

The RBB is made up of a `MultipartyCallRBB` first-contact object which creates `MultipartyCall` objects in response to an invocation of `createCall()`. These objects represent multiparty calls in the bearer network. When they are first created they are empty, such that no parties are involved in the call. In order to add legs representing parties, service logic calls `addLeg()` on the `MultipartyCall` object which adds the leg to the call within the bearer network and then returns a `Leg` object representing this leg. The `Leg` object is then manipulated in order to connect the terminal to a bearer network stream, remove the terminal from the call or to add event listeners which respond to specific bearer network events.

The classes making up this RBB are shown in figure 4.10. The main singleton, `MultipartyCallRBB` (figure 4.10(a)), is a formal `ReusableBuildingBlock` that acts in the same way as other RBBs in that it operates as a factory class that produces other objects that can be manipulated by service logic. The objects which it produces, `MultipartyCall`s, can store multiple `Leg` objects which in turn can each have a `LegEventListener` object registered with it, all of which is represented in the class diagram of figure 4.10(b).

A `MultipartyCall` object, which is created in response to an invocation of

`createCall()` on the `MultipartyCallRBB` object has methods relating to the basic functionality required of a multiparty call in the bearer network. Firstly, the object's constructor automatically creates a multiparty call in the bearer network and ensures that there is an association between it and the constructed object. The `addLeg()` and `removeLeg()` methods allow legs to be added and removed from the multiparty call respectively. The former constructs and returns `Leg` object which is associated to the relevant leg in the bearer network. The latter allows one of these `Leg` objects to be passed into it, signalling to the RBB that it destroy the associated call leg in the bearer network.

This `MultipartyCall` object can also be completely destroyed by calling its `destroy()` method. This firstly removes all legs (and associated `Leg`s) from the multiparty call and then destroys this multiparty call in the bearer network. The last method of this object, `getLegs()` simply returns a collection of all `Leg` objects currently associated with this call.

`Leg` objects provide the very basic functionality required for manipulating a participant in a multiparty call. The `startLeg()` and `endLeg()` methods connect and disconnect the associated terminals from the bearer stream of the call. The `registerEventListener()` method allows for an implementation of the `LegEventListener` interface to execute service logic external to the RBB in response to specific events within the network. This event listener can also be retrieved by service logic using the `getEventListener()` method, since it is also possible for service logic to indicate the occurrence of a call-related event (such as a terminal signalling over ALS that a call leg has been setup or disconnected).

The `Leg` class' constructor takes a `MultipartyCall` as an argument. This `MultipartyCall` instance is actually that object that has called the constructor of the `Leg` object currently being constructed. `Leg` objects are never constructed in isolation but rather in response to calls to `addLeg()` on a `MultipartyCall` object. The reason that a `MultipartyCall` is passed into the constructor and then stored in the `MultipartyCallRef` attribute, is to enable the `getMultipartyCall()` method of the `Leg` class. This method allows service logic to determine the `MultipartyCall` to which a specific `Leg` belongs. This method is required since service logic would intuitively have `Leg` objects associated to `Terminal` objects, such that when a `Terminal` is retrieved using the message context functionality of the framework, the `Leg` object to which it is associated can also be retrieved. In turn, this method would allow the associated `MultipartyCall` to also be retrieved.

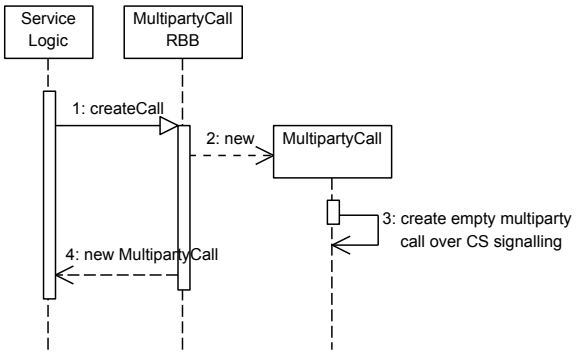


Figure 4.11: SD for creating a multiparty call in initial state (with no attached legs)

With the static structure in place, we now look to develop some reference sequences that service logic may implement to make use of the functionality that this RBB provides. These reference sequences relate to the most common way that the RBB can be used: allowing a terminal to request that it join a multiparty call and then having the AS instruct the bearer network to contact the terminal and join it to the multiparty call. This also includes the integration of call monitoring functionality by means of the `CallModelRBB`.

4.3.2 Dynamic Operation

Creating an Empty Multiparty Call

The most trivial message sequence is related to creating a multiparty call in its initial state as shown in figure 4.11. A call to `createCall()` (**1**) instructs the `MultipartyCallRBB` to construct a new `MultipartyCall` object (**2**). In the constructor, the RBB interfaces with the bearer network, signalling to it to initiate a multiparty call (**3**). Once this is complete, the new `MultipartyCall` object is returned to the service logic for manipulation (**4**).

Adding a Leg to a Multiparty Call

Once the `MultipartyCall` is created it is now possible to add legs to it. To do this, we follow the sequence in figure 4.12, which is similar to that of the sequence that occurs in response to a `makeCall()` invocation on the `TwoPartyCallService` as per figure 3.18. Adding a leg to a multiparty call is also similar to setting up a two party call in that it also relies on ALS to indicate terminal-side call setup events.

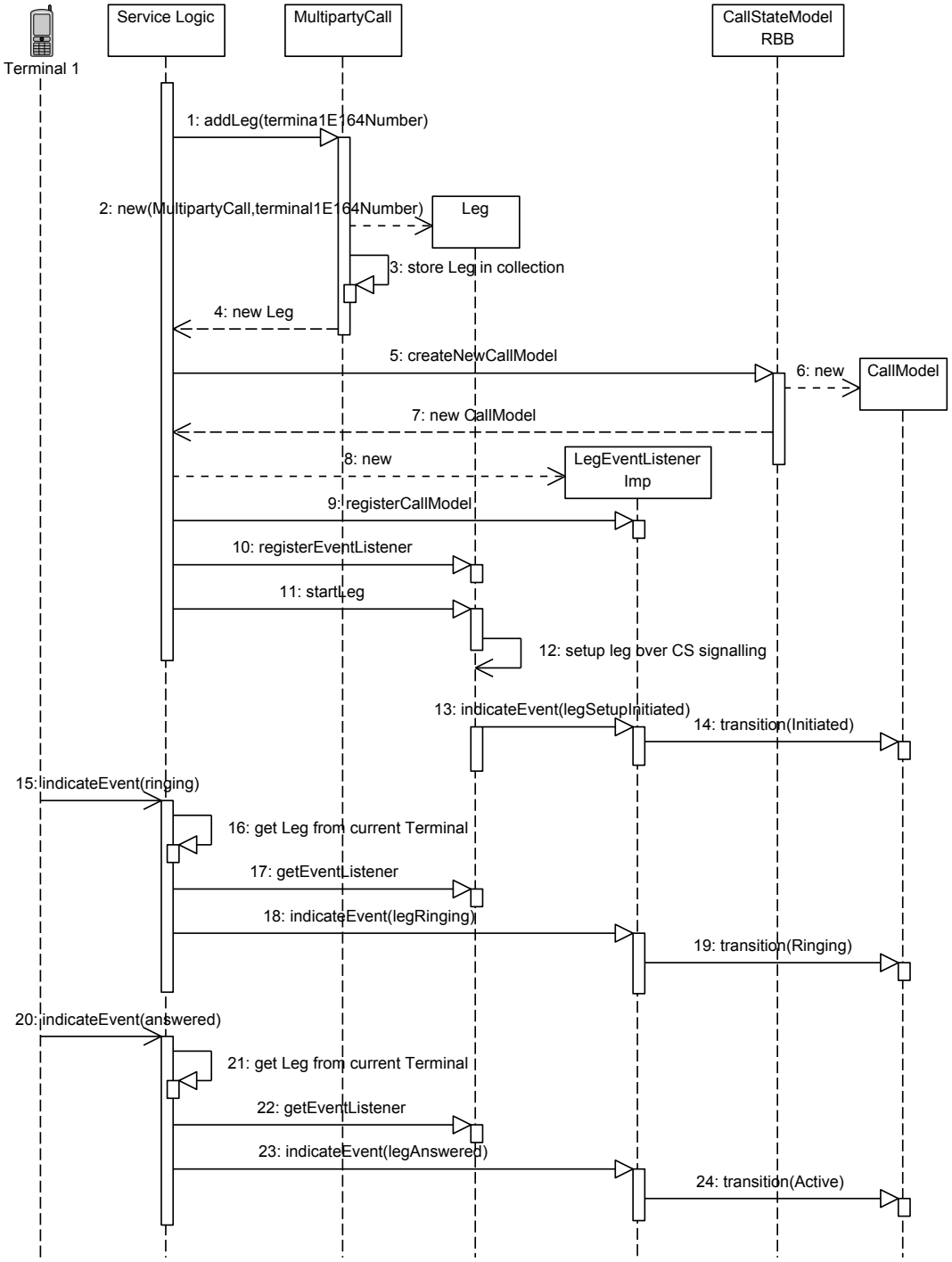


Figure 4.12: SD for adding a leg to a multiparty call

The process of adding a leg to a multiparty call begins when service logic calls `addLeg()` (**1**) on a `MultipartyCall` object. This instructs the `MultipartyCall` to create a new `Leg` object which it associates to the E.164 number passed into to the constructor and also to an underlying reference to a leg in the bearer network (**2**). The `MultipartyCall` then stores this new `Leg` inside its collection of `Leg` objects which are associated to it (**3**). This new `Leg` is then returned to the service logic (**4**) which stores it in such a way that it can be retrieved again within a specific message context.

At this point, service logic would usually register a `CallModel` with the newly created `Leg`. Whilst this does not necessarily have to be done, we include it here since all call related functionality of the framework requires the use of a call model for proper BCCM. Thus, the functionality of the `CallStateModelRBB` is invoked by requesting the creation of a new `CallModel` object (**5-7**). Then, an implementation of the `LegEventListener` interface, which has been specified to update a `CallModel` in response to call related events, is created (**8**) and the `CallModel` is registered with it (**9**). This `LegEventListener`, whose implementation is formally introduced in section 4.4.1 within the conference call service, is registered to the `Leg` just created above (**10**).

The `Leg` is now ready to be connected to the multiparty call bearer stream. To do this, the service logic calls `startLeg()` (**11**) which initiates the call leg setup using CS signalling within the bearer network (**12**). When the bearer network has itself initiated the setup of the call leg, it signals up to the application layer and in response a `legSetupInitiated` event is indicated (**13**) resulting in the `CallModel` transitioning to the `Initiated` state (**14**).

Since the process of a terminal joining a multiparty call is a 3rd-party initiated process, the terminal is contacted via the bearer network and therefore goes through the process of *ringing* and then being *answered*. However, since in this reference sequence we are considering the case for when a terminal actually requests via ALS that it join the multiparty call, the terminal does not actually ring or have to be manually answered by the user. Instead these two events occur without the user's knowledge. The formality is kept so as not to break the state transition rules of the `CallModel`. As such, when the terminal is contacted via the bearer network, it indicates a `ringing` event over ALS (**15**). In response, the `Leg` is retrieved using the message context (**16**) and its registered `LegEventListener` is retrieved (**17**). Service logic uses this event listener to indicate a `legRinging` event (**18**) which causes the `CallModel` to transition to the `Ringing` state (**19**).

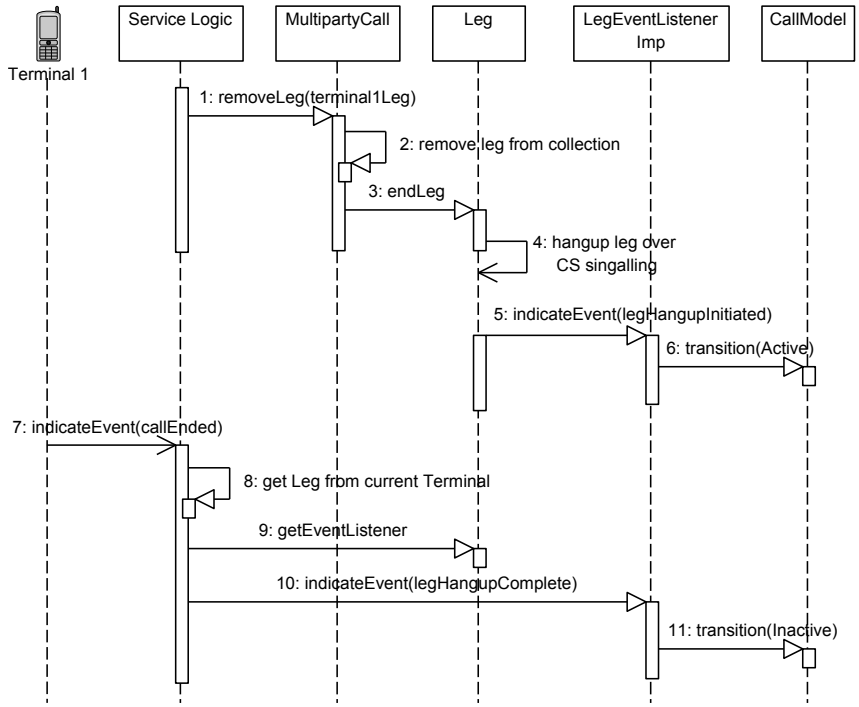


Figure 4.13: SD for removing a leg from a multiparty call

When the call is answered (or more correctly, the terminal accepts the connectivity request from the bearer network), it indicates an `answered` event to the AS (**20**). Once again, the related `Leg` object and then its registered event listener are retrieved (**21, 22**). A `legAnswered` event is indicated to this event listener (**23**) making the `CallModel` transition to the `Active` state (**24**). At this point, a stream should be setup within the bearer network between the terminal and the multiparty call and the user can participate in the multiparty call.

Removing a Leg from a Multiparty Call

The removal of a leg from multiparty call would often occur in response to an ALS request from the terminal. This would set off the sequence represented in figure 4.13. Beginning with service logic calling `removeLeg()` on the `MultipartyCall` object (**1**) to which the terminal is associated by passing in the `Leg` object representing the bearer network stream to the terminal. This would precede the `Leg` object being removed from the `MultipartyCall`'s collection of associated `Leg`s (**2**). This would then be followed by a call to `endLeg()` on the `Leg` object (**3**) making the RBB request from the bearer network that it disconnect the terminal from the multiparty call within the network (**4**). Once this disconnection process has begun, the bearer network signals this fact up to the framework, and a `legHangupInitiated` event is

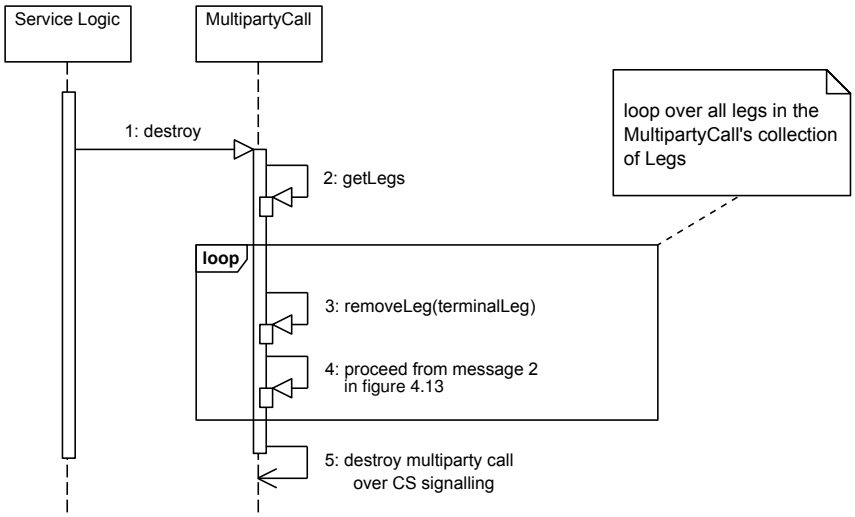


Figure 4.14: Destroying a multiparty call

indicated to the `LegEventListener` implementation (**5**) and the `CallModel` would be instructed to transition to the `Active` state (**6**).

Once the terminal has been fully disconnected from the multiparty call over CS signalling, it sends a `callEnded` event to the service logic in the AS (**7**). To respond to this, the relevant `Leg` object is retrieved using the message context (**8**). The `Leg`'s event listener is retrieved (**9**) and a `legHangupComplete` event is signalled to it (**10**) making it update the `CallModel` to a `Inactive` state (**11**) indicating that the terminal's leg has been completely disconnected from the multiparty call.

Destroying a Call

If the multiparty call has reached its conclusion there may be a desire to end it completely. There can be no guarantee that all connected terminals would have willingly disconnected from the call before the call is destroyed in the bearer network. Thus the sequence for gracefully destroying this call would involve forcing all terminal's to disconnect from this call and only then instructing the bearer network that the call be completely cleared away. The multiparty call RBB has been designed to cause this all to happen with a single call to the `destroy()` method on a `MultipartyCall` object. This is presented in figure 4.14.

When service logic calls `destroy()` (**1**), the `MultipartyCall` object retrieves all the `Leg`s associated to it and stored in its collection (**2**) and iterates through them, invoking the same set of messages to be called as per the leg disconnection SD of

figure 4.13 (**3, 4**). These messages are not repeated here, and the reader is referred back to this figure. The only slight difference with that set of messages is that the call to `removeLeg()` (**3**) originates from within the `MultipartyCall` object as opposed to originating from external service logic. Once all terminals have been disconnected from the call, the bearer network is signalled to destroy the multiparty call within it (**5**), effectively ending the multiparty call completely.

These SDs have covered that usage of the multiparty call RBB adequately enough for reuse within the conference call service. Thus we now leave the detail of this RBB and begin exploring the details of the conference call service which was our original aim.

4.4 Conference Call Service

The conference call service is a service which provides more structure to the concept of a multiparty call. Multiparty calls can be implemented in various ways. For example, user A can call user B, then put user B on hold while contacting user C and then bringing user B into the conversation with user C. In this way, a single user, user A, initiated the call and controlled it. The call did not exist until user A called user B and other users have no way of selectively joining this call without being contacted by user A.

The conference call service formalises multiparty calls into conference calls. That is these calls can be joined and left at will and are uniquely identified by a specific value, so that terminals are able to locate and then join them. We now detail this service as a means of showing how the framework copes with more complex call-based services.

4.4.1 Structure and Functionality

From a high-level the service is organised thus: created conference calls are stored in the service and identified by unique identifying strings. When a terminal joins this service, it joins the *lobby*. In doing so, it is furnished with a list of all unique identifiers for each of the conference calls in existence created by the service. This lobby is not represented by an object, rather it is just a means of stating that the terminal is not attached to any conference call. If a terminal creates a conference,

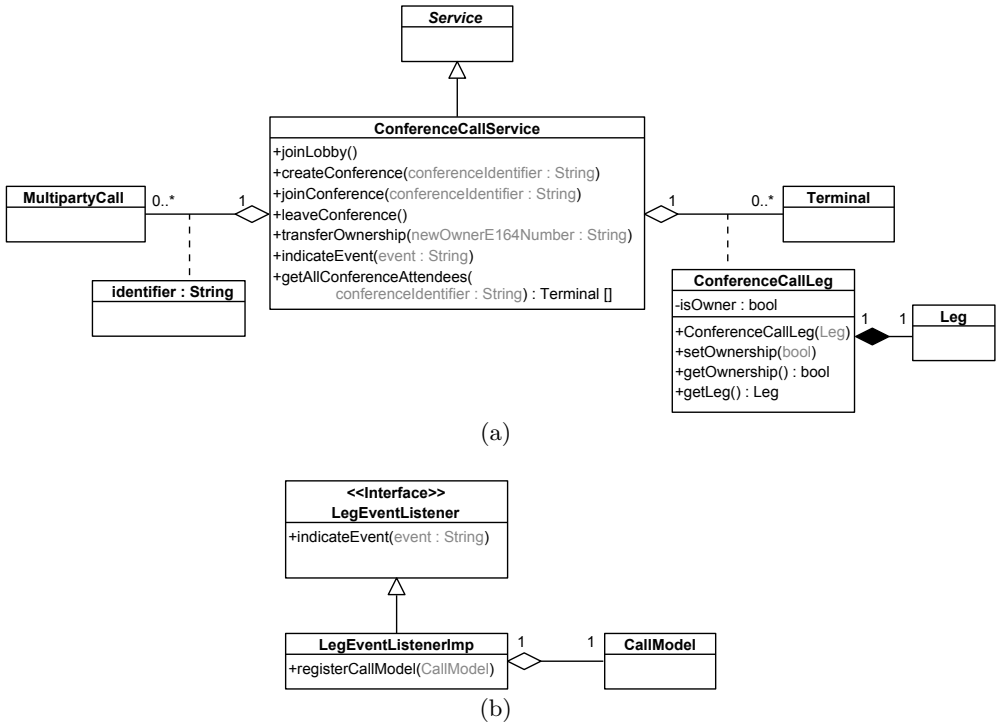


Figure 4.15: AS conference call service classes

it becomes the owner of this conference. Usually this would mean that the user of the terminal has the ability to prevent other terminals from accessing the conference call or remove terminals that are already part of the conference call (amongst other functionality). However, for this simplified conference call service, ownership simply means that if the owner leaves the conference, the conference will be destroyed. This in turn specifies that a conference cannot exist without an owner, and thus when a conference is created, the owner is automatically placed in it.

Main Conference Call Service Classes

Using this high-level description as a guide, we now detail the objects that make up the service. Figure 4.15 shows all the classes introduced to support the service's functionality. Starting with figure 4.15(a), the `ConferenceCallService` is a formal `Service` object that functions as the first-contact singleton for the service. Its purpose is to create new conference calls, store references to these conference calls against unique identifiers and to associate terminals with legs in these conference calls.

The `ConferenceCallService` hosts two maps to cater for this functionality. An

identifier string-to-`MultipartyCall` bidirectional map allows terminals to identify a conference call which they would like to join uniquely, noting that the combination of a multiparty call RBB's `MultipartyCall` object with a unique string makes up a conference call. A unidirectional `Terminal`-to-`ConferenceCallLeg` map enables message context functionality, allowing the current terminal reference to be used to retrieve its associated conference call leg and hence conference call (since a `Leg`, which is wrapped in a `ConferenceCallLeg`, contains a reference back to the multiparty call as per the multiparty call RBB).

This service introduces a `ConferenceCallLeg` class as a wrapper to the multiparty call RBB's `Leg` to provide the `Leg` with the properties needed for the ownership functionality of the conference call service. This class defines two methods to support conference call ownership, namely `setOwneship()` (which sets the terminal corresponding to the `ConferenceCallLeg` as an owner or not based on a `true` or `false` boolean value, respectively) and `getOwnership` (which simply determines if the terminal is the owner or not). Besides the constructor, which takes the `Leg` to be wrapped as an argument, the `getLeg()` method is the last method of the class which allows the wrapped `Leg` to be retrieved so that it can be used to interface with the multiparty call RBB.

The `ConferenceCallService` object has various, mainly remotely-callable, methods which represent the basic functionality of the service. They are each described in the following list:

- `joinLobby()` (detailed in figure 4.17) - A terminal calls this method to create an initial association to the service. In response, it receives a list of unique identifiers for the available conferences that it can join.
- `createConference()` (detailed in figure 4.18) - A call to this method creates a new conference and automatically attaches the terminal that requested the creation to this conference.
- `joinConference()` (detailed in 4.19) - Makes a terminal join the conference specified by the `conferenceIdentifier` string.
- `leaveConference()` (detailed figure 4.21) - In response to a call to this method, the requesting terminal is removed from the conference in which it is currently participating. Note that this method does not require a conference identifier as an argument as message context functionality ensures that the relevant conference call can be retrieved.

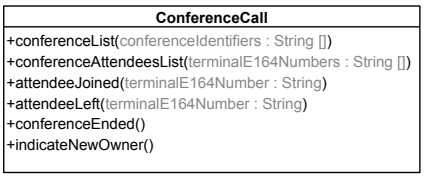


Figure 4.16: Terminal conference call class

- `transferOwnership()` (detailed in figure 4.20) - The service allows for a conference to change ownership such that the original owner can leave the conference without it being destroyed. The method is remotely called with an E.164 number as an argument indicating the new owner.
- `indicateEvent()` - This method is for enabling the ALS of bearer network level events (such as the success of a stream being setup between a terminal and the remainder of a conference).
- `getAllConferenceAttendees()` - This is the only method which is not remotely callable. It is a utility method which iterates through the `Terminal`-to-`ConferenceCall` leg map and builds up a list of `Terminal` objects which are associated to a specified conference call. It is possible to identify this relationship in the following manner: `conferenceIdentifier` → `MultipartyCall` → `Leg` → `ConferenceCallLeg` → `Terminal`.

Bearer Network Event Listener

In the SDs of figures 4.12 and 4.13 an implementation of the `LegEventListener` interface of the multiparty call RBB was used. This implementation was the `LegEventListenerImp` class in figure 4.15(b). Besides the `indicateEvent()` method which has been implemented to update a `CallModel`, if a bearer network event occurs, this `LegEventListenerImp` introduces a `registerCallModel()` method which allows the `CallModel` to be registered with it.

Conference Call Service Class in Terminal

For this service to function correctly, certain remotely-callable methods have to exist in the terminal as well. These methods are grouped into `ConferenceCall` class existing on the terminal (shown in figure 4.16) and their uses are summarised in the following list:

- `conferenceList()` - This method is called by the AS to send the terminal a list of all the conferences being hosted. Usually called in response to the terminal joining the "lobby".
- `conferenceAttendeeList()` - In response to a terminal joining a conference call, the AS calls this method to send the terminal a list of E.164 numbers of the attendees currently participating in the call.
- `attendeeJoined()` - When a new attendee joins a conference in which a terminal is already a participant, a call to this method will alert the terminal along with the E.164 number of the new attendee.
- `attendeeLeft()` - This method is similar to the case of a new attendee joining a conference, except it is used to indicate that an attendee has just left a conference call.
- `conferenceEnded()` - If a conference call is ended before a terminal has voluntarily disconnected from it, it is informed that the call has ended using this method, giving it a chance to perform the necessary functions to disconnect gracefully from the conference call.
- `indicateNewOwner()` - Since ownership of a conference call can be transfered from one terminal to another, if this occurs, the new owner is notified using this method.

4.4.2 Dynamic Operation

With the static structure of the service defined, we now move onto to presenting SDs defining the dynamic behaviour of the service.

Initial Conference Call List Retrieval

Before taking part in a conference call, a terminal must first join the conference call "lobby" and retrieve a list of available conference calls . This process is performed according to the SD of figure 4.17. A terminal remotely calls `joinLobby()` on the `ConferenceCallService` object (**1**), making the object retrieve a list of conference call identifiers from the conference call identifier string-to-`MultipartyCall` map (**2**). This list is then sent to the terminal (**3**). This process does not change the state of the conference call service and is just a mechanism for the terminal to retrieve available conference calls.

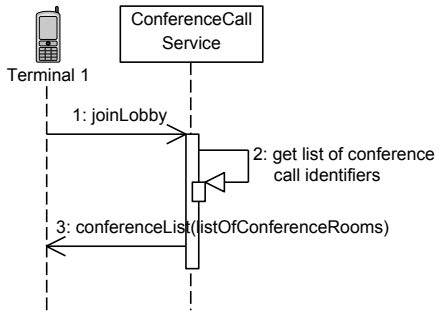


Figure 4.17: SD describing the initial retrieval of available conference calls

Creating a Conference Call

Once a terminal has received a list of all available conferences, its user can either create a new conference or join an existing one. Figure 4.18 covers the message sequences which occur in response to a user opting to create a new conference. As mentioned earlier, once this new conference has been created, the user is automatically placed in it with the role of being the owner, since in this service it has been specified that a conference call cannot exist without an owner as a participant.

The case of joining a conference is covered after the description of the creation of a conference call. Some parts of the conference call creation message sequence will be reused in the joining a conference call sequence. Further, some message sequences describing the operation of the multiparty call RBB are reused here.

When a user decides to create a conference call, a `createConferenceCall()` message is sent over ALS, with a unique name for this conference selected by the user as an argument (**1**). This causes the `ConferenceCallService` object to orchestrate the creation of a `MultipartyCall` object and hence the setup of a multiparty call in the bearer network. This is done by calling `createCall()` on the `MultipartyCallRBB` object (**2**), which results in a new `MultipartyCall` object being constructed (**3**) and the bearer network being signalled to create an associated multi party call (**4**). The new `MultipartyCall` is then returned to the conference call service logic (**5**). This new `MultipartyCall` is mapped to the identifier string sent in the `createConferenceCall()` message (**6**).

With the multiparty call created, the terminal now has to join it as an owner. Since the multiparty call RBB uses E.164 numbers in the arguments to the methods used for setting up calls, the current terminal's E1.64 number has to be retrieved. First, the current message context `Terminal` is retrieved (**7**). Then its E.164 number is

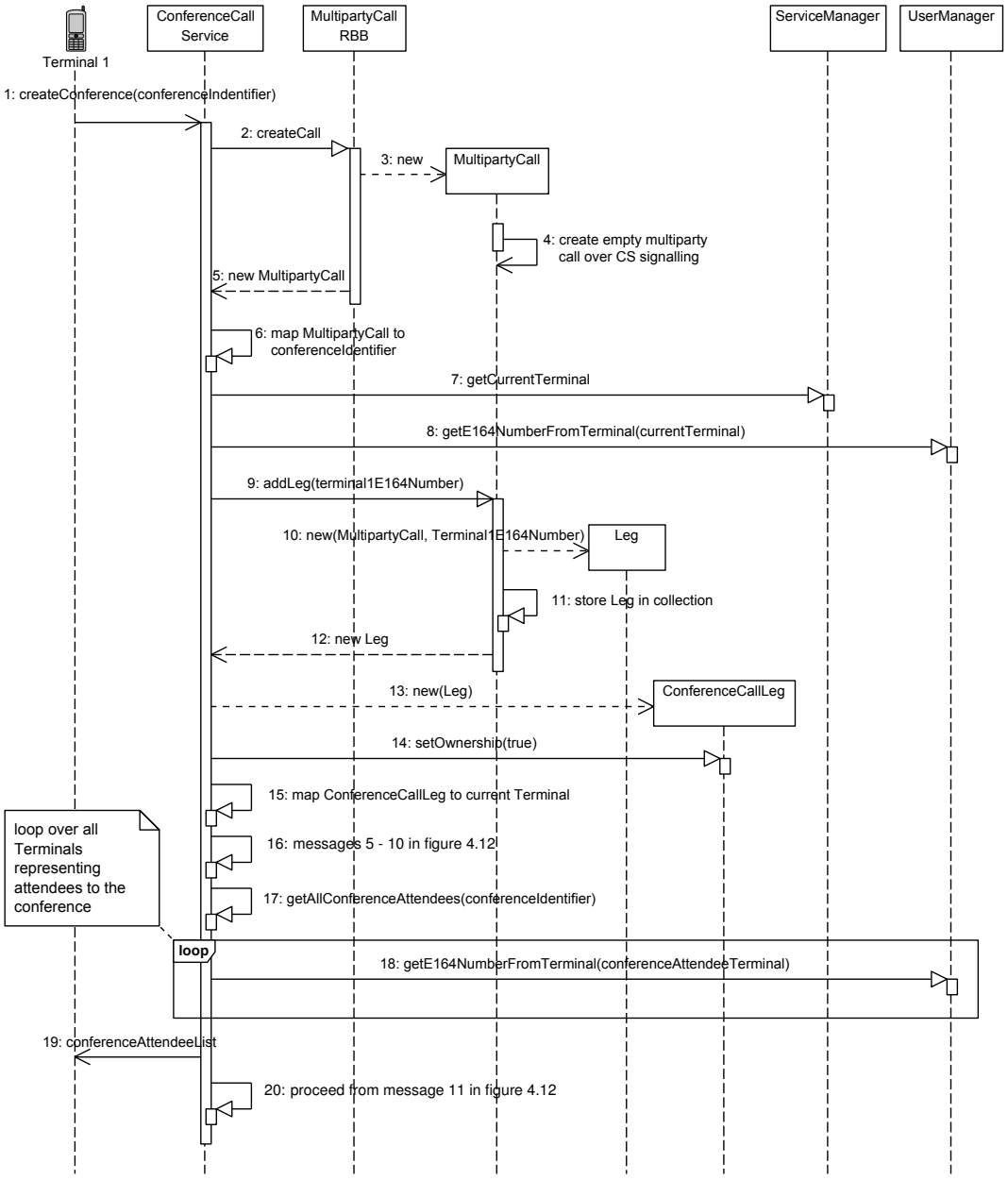


Figure 4.18: SD for creating a conference call

retrieved from the `UserManager` (**8**). By calling `addLeg()` on the `MultipartyCall` with this E.164 number as an argument (**9**) a new `Leg` is created (which also initialised a call leg within the bearer network) (**10**), stored in the `MultipartyCall` (**11**) and returned back to the service logic (**12**). This `Leg` is wrapped in a new `ConferenceCallLeg` (**13**) and the service logic indicates that this leg is attached to the owner of the conference call (**14**). The `Leg` is then mapped within the `ConferenceCallService` to the current message context `Terminal` (since it also represents a bearer stream to the same terminal) (**15**).

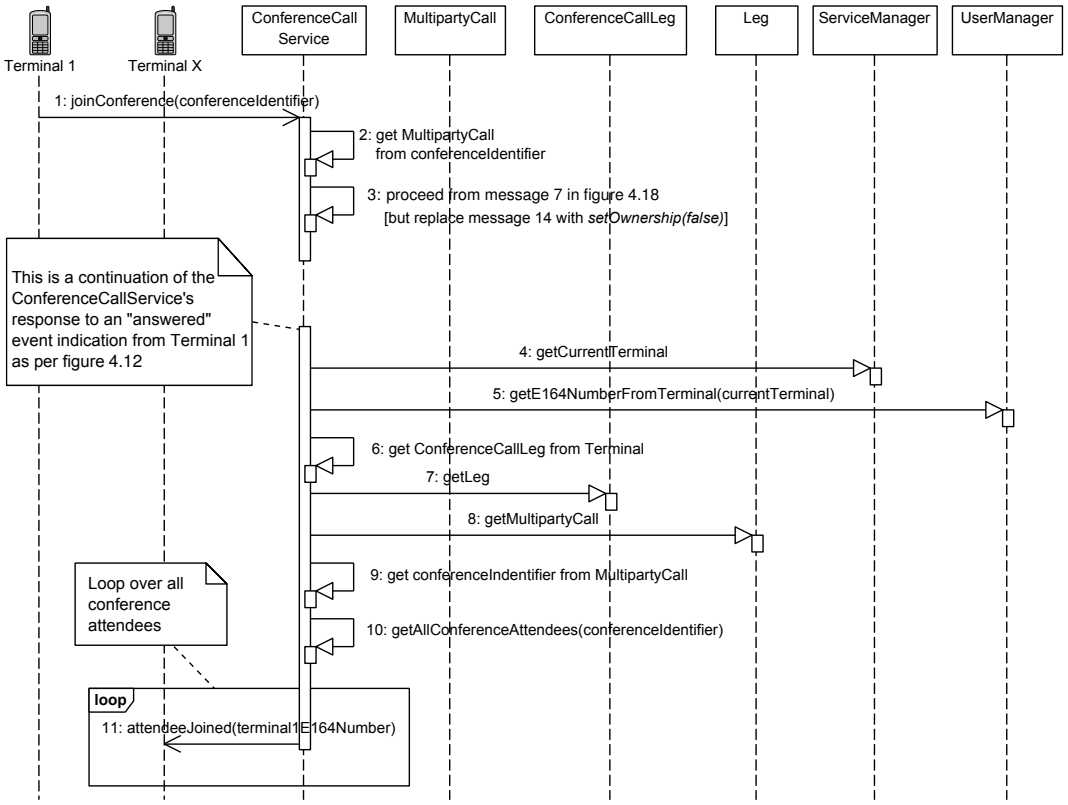


Figure 4.19: SD for joining a conference call

At this point, reference message sequences are reused from the multiparty call RBB. These cover the logic for associating the newly created `Leg` to a `CallModel` using a `LegEventListener` (**16**). Once this is complete, the service logic retrieves the list of parties already in the conference call (**17**) and each of their E.164 numbers (**18**) which are then sent to the current terminal (**19**). Obviously at this point as the conference call has just been created, the list of E.164 numbers will be empty. We are only indicating this sequence here since this logic will be reused by the next SD for a user joining a conference call.

The entire process ends by following the same reference message sequence for adding a new leg to a multiparty call which has the service logic request the start of setting up a bearer stream to the current terminal along with all the relevant ALS messages, event notifications and call model transitions to support this (**20**). With this process complete, the conference call is now fully setup and a bearer stream has been joined to the owner.

Joining a Conference Call

In order to respond to a request from a terminal that it join a specific conference call, the AS follows the logic specified in figure 4.19. The initial part of this sequence reuses the message sequence of adding an owner to a newly created conference call, except in this case, not setting the current terminal to be an owner. Having connected the terminal to a conference call within the bearer network and associated it in the application layer as well, all other terminals in the conference call are alerted. The entire process is now detailed.

When a terminal wishes to join a conference call, it calls `joinConference()` on the `ConferenceCallService` object, using the unique conference identifier as an argument (**1**). The service then retrieves the `MultipartyCall` object associated to this identifier (**2**) and undertakes the process of connecting the terminal to the conference call. This involves creating a `Leg`, wrapping it in a `ConferenceCallLeg` which is mapped to the current `Terminal`, sending the list of the conference attendees to the terminal and then connecting the leg within the bearer network (with all associated ALS method calls, event notifications and call model transitions) (**3**).

When the terminal is fully connected, the service has to alert other conference call participants of the new terminal. To do this, when the terminal indicates the `answered` event at the end of the bearer connection setup, the service logic uses this opportunity to notify the other participants. The list of participants has to be retrieved using the `getAllConferenceAttendees()` method which requires the current conference call's unique identifying string. To obtain this, the current message context `Terminal` (which is a reference to the newly connected terminal) is retrieved (**4**) followed by its E.164 number (**5**). The newly created `ConferenceCallLeg` mapped to this `Terminal` is retrieved (**6**) and then used to obtain the `Leg` object representing the new terminal's bearer connection within the network (**7**). Using the `Leg`'s `getMultipartyCall()` method, the `MultipartyCall` to which this `Leg` belongs is fetched (**8**). The `MultipartyCall` is mapped to the conference call's identifying string which can now be obtained (**9**). This string is passed into the `getAllConferenceAttendees()` method which returns a list of `Terminal`s representing the other conference call participants (**10**). The service logic iterates through this list, sending the new terminal's E.164 number to each terminal via a call to these terminal's `attendeeJoined()` method (**11**). This ends the process of a new terminal joining a conference call, and the terminal's user can now participate in the call.

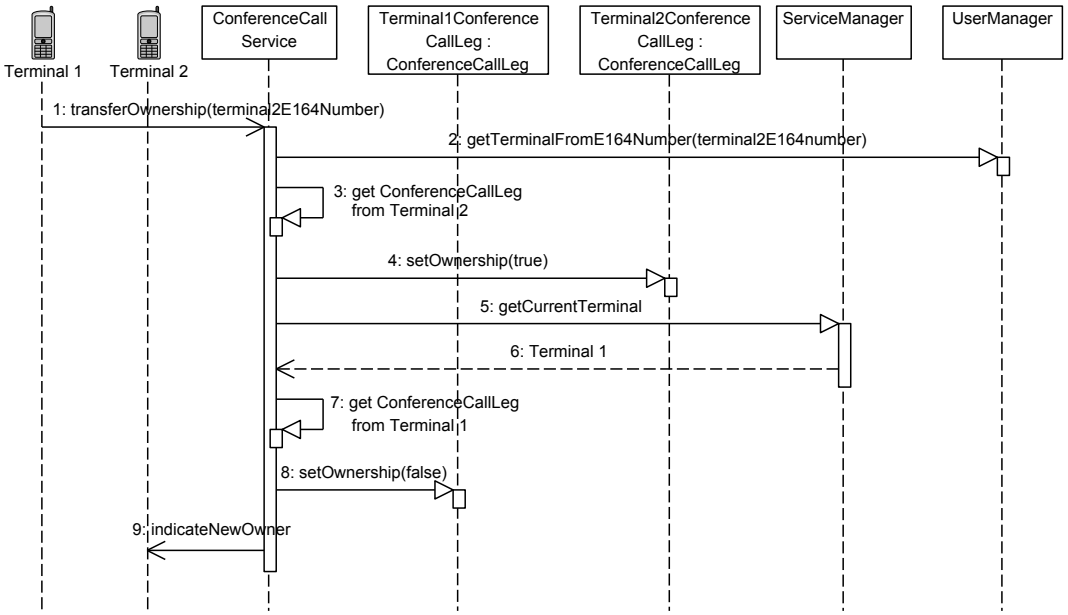


Figure 4.20: SD for transferring ownership of a conference call

Transfer Ownership

The service offers the owner of a conference call the ability to transfer ownership to another terminal. Figure 4.20 covers the message sequence involved in transferring ownership from a hypothetical Terminal 1 to another Terminal 2. The process begins when Terminal 1 calls `transferOwnership()` on the `ConferenceCallService`, passing in Terminal 2's E.164 number as an argument (**1**). The service logic calls on the `UserManager` to retrieve Terminal 2's representative `Terminal` (**2**) which is in turn used to retrieve the `ConferenceCallLeg` to which it is mapped within the `ConferenceCallService` (**3**). A call to `setOwnership()` on this `ConferenceCallLeg`, passing in a boolean value of `true`, results in Terminal 2 becoming an owner of the conference call (**4**). Then to remove Terminal 1's ownership, its representative `Terminal` is obtained using message context functionality (**5, 6**), allowing its associated `ConferenceCallLeg` to also be obtained (**7**). Terminal 1's ownership is then revoked by calling `setOwnership()` with `false` as an argument (**8**). Terminal 2 is then notified that it is the new owner (**9**).

With Terminal 2 as the new owner, if it leaves the conference call, the conference call will be destroyed as will be shown in the next message sequence for leaving a conference.

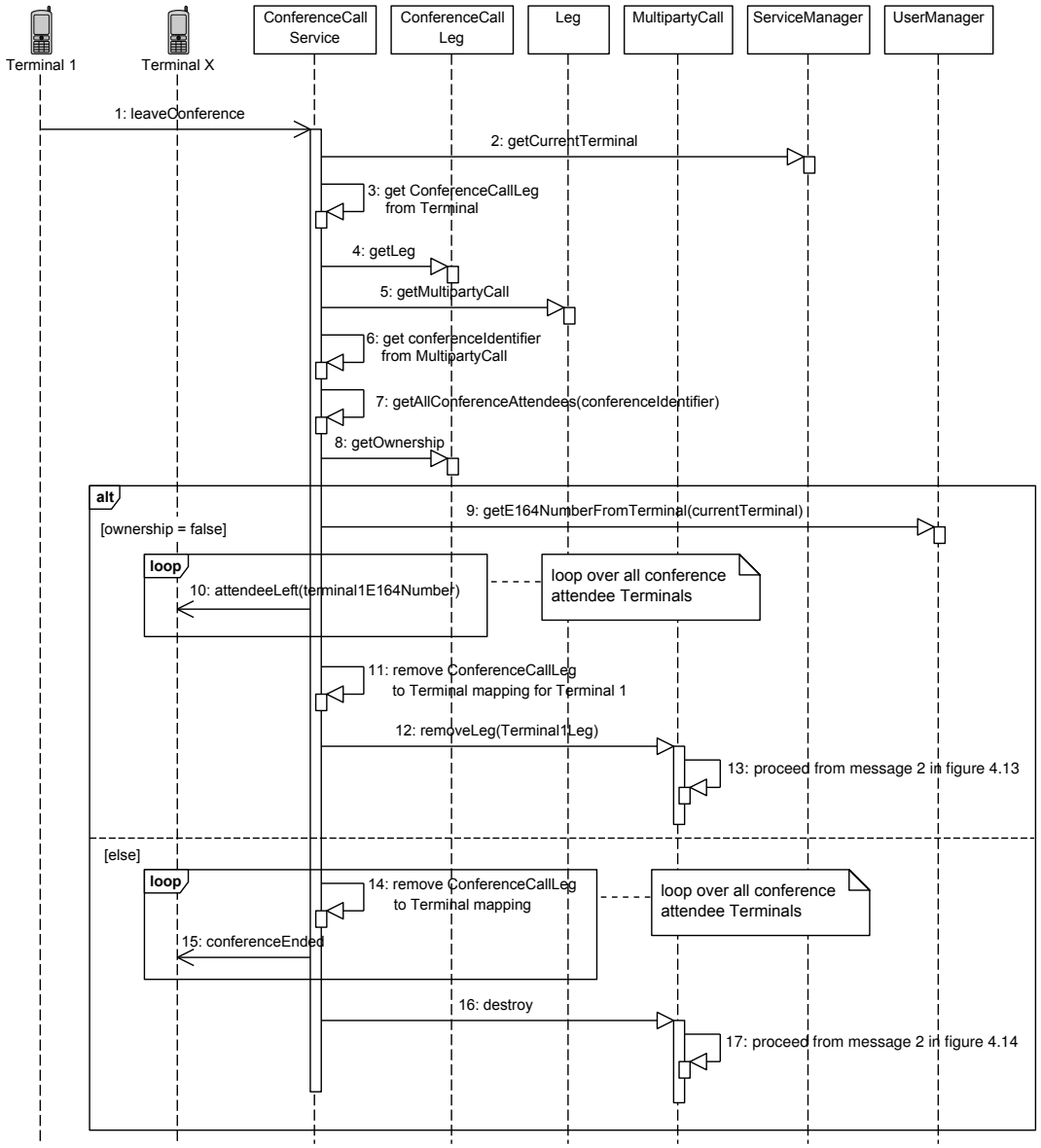


Figure 4.21: SD for a terminal leaving a conference. This terminal may or may not be the conference owner.

Leaving a Conference

When a terminal leaves a conference call, one of two things can occur. If it is not an owner, it simply leaves the conference and all other participants are notified. If it is an owner, all other terminals are forced to disconnect and then the conference call is destroyed. Both cases are covered in figure 4.21 and are separated using an *alternative* combined fragment in UML. The entire process reuses reference message sequences for the multiparty call RBB.

When a user wishes to leave a conference call, his or her terminal requests this by calling `leaveConference()` (**1**). Four pieces of important information are now required: the `Leg` object representing the terminal in the conference call, the `MultipartyCall` object representing the conference call, the Terminal's ownership status over the conference call and a list of participants in the call. These are all retrieved thus: first the `Terminal` object representing the requesting terminal is accessed using the current message context (**2**). The `ConferenceCallLeg` mapped to this `Terminal` is then retrieved (**3**), allowing for the `Leg` representing the terminal in the conference call (**4**) and hence the `MutlipartyCall` of this conference call to be accessed (**5**). The `MultipartyCall` is associated to an identifying string within the `ConferenceCallService`, so this string can now be retrieved (**6**) and then used in the `getAllConferenceAttendees()` method to get a list of participants in the call (**7**) (to be used later). Lastly, whether the terminal is owner of the conference call is determined (**8**). All required information has now been retrieved.

It is now at this point that a process breaks into one of two message sequences depending on whether the terminal is the owner of the conference call or not. If the terminal is not the owner, then the sequence of simply removing the terminal from the conference call and alerting all other terminals is followed. This is performed in the following steps. The current terminal's E.164 number is retrieved (**9**) and the list of call participants' `Terminal`s retrieved earlier is iterated over, calling `attendeeLeft()` on each of them with the current terminal's E.164 number as an argument (**10**). Then the `ConferenceCallLeg` object representing the current terminal's bearer network connection is removed from its mapping to the terminal's `Terminal` object (**11**) – effectively removing its application layer association to the conference call. After this, the multiparty party call RBB is called on to remove the bearer network connection. By calling the `removeLeg()` `MultipartyCall` method with the current terminal's `Leg` as an argument this process is started (**12**). This involves removing the `Leg` from the `MultipartyCall`, calling `endLeg()` on the `Leg` to end the connection in the bearer network and handling all events that occur in response by updating the `CallModel` appropriately (**13**).

For the case where the terminal disconnecting from the call is the conference call owner, all other terminals have to also be disconnected from the call and along with the call having to be destroyed. This is achieved in the alternative flow of messages in figure 4.21. Before the conference call is ended in the bearer network, all `Terminal` objects representing call attendees, have to be disassociated from the `ConferenceCallLeg`s of the call (**14**). Also, `conferenceEnded()` has to be called on each of the terminal's participating in the call (**15**), thereby indicating over

ALS that the call is about to be ended. After this, an invocation of the `destroy()` method on the object representing the conference call in the bearer network initiates the destruction of the call (**16**). The actual process of destroying the call using the multiparty call RBB has been covered previously, but in summary involves: getting all the `Leg` objects associated to the call, calling `removeLeg()` repeatedly on the relevant `MultipartyCall` with each `Leg` as an arguments, handling all bearer network events as they occur and then finally destroying the call once all call legs have been disconnected (**17**).

4.5 Conclusion

Using the framework built up in chapter 3 and the ALS service development paradigm, we have quite simply built up complex services. The service logic for all these services, including the BCCM-based conference call service, is completely constrained to the application layer. The state of each service can be determined without accessing the bearer network and the operation of the service can be fully comprehended in detail by looking only at the objects instantiated by the framework and the sequences of messages that they send between each other. This point is emphasised when one considers the fact that all the message sequences making use of bearer network functionality show interaction with bearer network with only single messages amongst many application layer messages.

The services used to demonstrate the framework are built up using important parts of the framework including ALS (used for bearer network level event indication as well), message context functionality and the ability of a service to invoke the service logic of other services. The framework's functionality is simple enough to allow services to be developed easily and general enough to give flexibility in service design whilst ensuring that service logic can be developed to be fully hosted in the application layer.

This presentation of various services has attempted to promote the simplified service framework and ALS as worthy contenders for service development in a modern telecommunications. Whilst no attempt has been made to formalise all aspects of the framework or of the protocols used over ALS, it is already possible to gauge the power of a service environment based on the concepts that they put forward.

The next chapter look to conclude this research report by considering the grander

purpose of ALS and application layer focused service development within the highly competitive value-added telecommunications service business. Amongst looking at the potential of the framework, further work on it will be considered so as to make it more appealing to service developers not just as a tool but as a completely new way to look at service development.

Chapter 5

Conclusion

The research presented in this report undertook to simplify service development by first identifying the weaknesses in the current modern service development environment and then considering the overuse of the bearer network to support service logic as the main culprit causing service development to be overcomplex. Using the "complexity-at-the-edge" design of the Internet as inspiration, chapter 1 introduced the intuitive solution to this being the removal of the heavy reliance on the bearer network and its CS signalling and only having terminals and ASs host and process service logic.

5.1 Service logic exclusively in the application layer

In the contemporary solution to complex service development, service logic is distributed using the bearer network as transport and the service logic in the application layer has to interface with the bearer network. Inherently this forces service developers to make use of CS protocols to interface with the bearer network, resulting in the incorrect use of a protocol as an API and abstractions of the bearer network leaking into service logic. Forcing service logic into the application layer does away with the need for using bearer network-centric protocols when developing services and also ensures that software developers from an IT background need not be very familiar with telecoms signalling technologies — which widens the base of programmers correctly qualified to develop telecoms services.

This migration into the application layer is covered in chapter 2 and is a process involving two main components.

1. Replacing CS signalling for service logic distribution with a mechanism more suitable to the application layer

2. Structuring the application layer formally so that the abstractions from the bearer network give a complete, adequately detailed overview of the particular part of the bearer network being monitored and controlled. This structuring should also aim to provide a lot of reusable functionality and simplification in the deployment of services.

The aim is not to replace the bearer network, but rather to make it secondary to the functioning of service logic. This is an inversion of control in which service logic calls on the bearer network when the setup of a bearer stream is required as opposed to service logic being triggered only by specific events during the processing of a bearer connection. The idea is that when the application layer is the locus of service logic, sessions begin and end in the application layer. The first contact that a terminal will make is with an application server as opposed to some node in the bearer network.

ALS plays the critical part here. By allowing application logic in terminals to communicate directly with logic in ASs, the bearer network is not involved in the service initiation and invocation at all. In fact, it would be possible for an entire service session to proceed without the bearer network playing a part. ALS is not a communication protocol but rather a design feature of a service development environment that differentiates between "downwards" and "sideways" signalling. When ALS is introduced the communication between terminal and AS is seen as "sideways" such that the messages relevant to application logic are not processed in the lower layers of a network infrastructure as would be the case when "downwards" signalling provides communication between inter-layer network entities.

In keeping with the constraints of the application layer, service developers need to be correctly abstracted from the bearer network and also have an intuitive interface to ALS to make it both robust and appealing for service development. This can be accomplished by enforcing some amount of structure on the application layer. As it stands, service development environments are not standardised. With every service development environment standard or product offering comes a different approach to abstractions and managing service logic. This research report showed how service development environments can be better structured when service logic stays in the service layer. Firstly, the environment can be structured as a framework in which service logic execution is controlled by this framework. Secondly bearer network abstraction and other reusable logic can be grouped into simple atomic building

blocks whose functionality can be orchestrated into complex composite services.

5.2 Framework

This research report developed a framework which supports application layer focused service development. Whilst not detailed to the extent required for implementation by a telco, it does cover the technical requirements of service environments: AAA, bearer network interfacing, service logic distribution and reusable functionality. It also goes further by supporting the unique feature of ALS as a critical part of its infrastructure.

The framework is built in an IT-friendly manner by using OO to represent all components of the framework and advocating services being built up using OO. Further, the entire framework is kept well decoupled using various manager objects which control access into various parts of the framework. This decoupling is also supported by message context functionality and dynamic method invocation, whilst a form of remote method invocation is used for enabling terminals and ASs to access each other's functionality.

Chapter 3 detailed the features, structure and dynamic operation of the framework and explained how services are built up and deployed within the framework.

5.3 Main features

When developing the services within the framework, as is done in chapter 4, certain aspects of the application layer constrained service development become apparent as being very useful.

Object-orientation Keeping application logic in the application layer does not necessarily call for the use of the OO. However in this report, its use was emphasised due to its strong abstraction and decoupling abilities. It is also useful along with UML for representing high-level overviews of service logic.

Reusability Being able to reuse already implemented functionality in a controlled manner plays a critical role in this service environment. Even though the framework

is built up using the principles of application layer service development it still requires the use of the bearer network. Whilst the fact was emphasised that a programmer should have no direct contact with the bearer network and its protocols, there still needs to be a standardised means for accessing the *functionality* of the bearer network especially for BCCM-supporting services. To this end, the framework was built to support RBBs, which, as was seen from the various SDs describing service logic, is critical to simple implementation of complex services.

Separation of "sideways" signalling It was recognised when building up the framework that ALS could not be built into an RBB. Since services are initiated in terminals and service logic is invoked in the application layer, ALS is a critical element of the framework and thus had to be treated as a separate function of the framework. The programmer is presented with a very abstracted view of ALS in which sending a message to a remote terminal only requires a call to a method on an object representation of that terminal and receiving a message only requires that service objects be aware that method calls may originate from remote terminals. In spite of its simple implementation, ALS is pivotal in ensuring intuitive service logic distribution.

Simplification and Abstraction The ability of the framework to represent service logic very simply using object-orientation rose naturally out of the implementation of ALS, the structuring of the framework and the abstraction of the bearer network. The simplification goes hand in hand with the abstracted views of the various components of the framework. The simplification is embodied in the manager objects which make up the framework which are capable of controlling access to the service logic without breaking the message passing relationships of service objects. The simplification is also captured by the fact that service logic can treat terminals as being co-located by having the abstraction of the transport of service logic messages to distributed nodes provide this. Further, using correctly abstracted OO BCCM RBBs (such as for two party and multiparty calls), the programmer is presented with intuitive and simplified access into the bearer network.

BCCM in the application layer When service logic gets moved completely into the application layer one must consider what happens to the control of the bearer network and its representation in the application layer. Of the two options for this, the first is that the application layer not be heavily integrated with the control

and monitoring of calls, simply making requests to the network to control calls and obtain their state. The second option, which provides more intuitive integration into the bearer network without sacrificing abstraction is to have a model of the call maintained in the application layer so that it can be both controlled and monitored. A BCCM RBB can be built to ensure that a model always correctly represents the state of a call. A programmer can then use an OO interface to the call, making method invocations on an object representation of the call and treating the call as if it exists in the application layer. The second approach is better supported, since in the inversion of control in which the application layer becomes the focal point of service logic, it is intuitive to also have the primary point of BCCM in the application layer.

5.4 Future work

It was mentioned throughout this report that no attempt will be made within the report to formalise the functionality and protocols used by the framework. To make this into a service development environment which could be integrated by a telco these would have to be considered very carefully and in detail. This standardisation would cover the ALS protocol, the structure of a homogenised set of RBBs especially for interfacing with the bearer network and formalising the manner which AAA is performed.

Service interaction needs to also be considered. Services built up in this framework can be very complex and constructed using many objects with many method calls, opening up the possibility of incorrect operation due to objects communicating not according to design. This is exacerbated by the fact that service logic can be invoked at any point by any terminal and more so by the fact that services are also able to be invoked by the logic of other services. Thus, further design would have to go into implementing some control over method calls based on whether they would potentially cause a service to go into an invalid state.

The further item of important future work is in regards to BCCM. It may seem at first unintuitive to bring bearer functionality up into the application layer especially since this research report aimed to decouple service logic from the bearer network properly. A formal implementation of this framework would have to ensure that the bearer network remains under the control of the telco keeping the BCCM simply in the form of model whose representation it keeps updated. A programmer should

have no control beyond this model.

Lastly, business considerations must be made. It can be noted that the framework can be implemented in two ways. The first places more control in the hands of a 3rd-party service developer. In this case, the AS is not controlled by the telco (the owner of the bearer network) and connects to the bearer network over an open interface. The 3rd-party service developer can decide what service logic should be run. This also requires that terminals be allowed to connect directly to these 3rd-parties without traversing the telco's bearer network. The other implementation would have the AS under the control of the telco which can decide the services to be hosted. Obviously this creates a more closed service development environment, however, the end-user is provided with greater quality guarantees traditionally associated with telcos and does not have to deal with 3rd-parties which may not be as reliable. These are not the only business considerations, but the manner in which application layer focused service development is implemented from a business perspective will greatly shape the actual development of this approach to service development.

5.5 Summary

Simplification was reached by moving all service logic into the application layer and by introducing ALS, ensured that this service logic does not leak into the bearer network. By also considering the structure of an application layer-centric service development environment and various means for removing the need to utilise CS to integrate bearer network functionality into a service, this research presented an efficient, robust and novel approach to service development.

Appendix A

Proof of Concept

As a means for gauging the viability of the framework, an implementation of it was undertaken. The various tools used to support the implementation are discussed here along with a brief look at the result.

A.1 Implementation Tools

In order to develop a reference implementation, the correct tools had to be selected in order for the framework to be programmed efficiently and simply as possible, allowing all the theoretical concepts developed in this report to be supported correctly so as to prove that such a framework is realisable. The tools used are now covered.

Whilst the bulk of the research report does not go heavily into the design details of how terminals handle service logic, we cover some of the technologies used to support it here, since the proof of concept would serve no purpose without correctly functioning terminal logic.

The aim of the proof of concept was to make the implementation as real as possible. Thus service logic was actually distributed amongst an AS implemented on a server computer publicly accessible on the Internet and on cellular telephones with Internet connectivity acting as terminals. The tools used in the proof of concept implementation therefore provide some viable solutions to the various requirements of the framework.

A.1.1 Application Server

The AS implementation was tested on a computer with a server architecture. Whilst no consideration was made for scalability, the server provided a robust platform on which the framework could operate.

Main Implementation Language

Java was selected for implementation of the AS due to its strength in implementing OO designs and capability of following these designs very closely. Whilst various AS frameworks already exist for this language, the decision was taken to implement the framework from the ground up as a means of showing the framework's completeness as an AS.

The Java Standard Edition also has most of the functionality required by the framework already built in. These are covered in the following subsections.

Application Layer Signalling

The research report makes no attempt to standardise the technology over which ALS occurs. There are already many mature technologies based in Java which exist for sending and receiving messages to and from remote terminals. So as to keep the implementation as simple as possible a low-level approach was taken. Connectivity to remote terminals was implemented using sockets. The `ApplicationLayerSignalling` component of the framework completely abstracted these sockets so as not to tie the framework to a particular implementation.

Since the AS is exactly that, a server, it has to handle multiple simultaneous connections from clients (terminals). In a raw form, basic sockets are designed for single connections to single clients. The ability to handle multiple connections has to be built on top of these sockets. The `java.nio` package provides this management of sockets. Since multiple simultaneous clients infers that multithreading has to be used to handle messages sent from these clients, `java.nio` provides mechanisms for new threads to be launched each time a message is received from a terminal. Within these threads, dynamic method invocation occurs to call methods on the desired objects.

Dynamic Method Invocation

Another aim of the framework is to ensure that it can continue running when new services are installed. This means that service logic cannot be precompiled into the framework. However, there still needs to be a mechanism for calling on objects' methods containing service logic when these objects did not exist when the framework was compiled.

Java is a reflective language, meaning that it is possible to both instantiate objects using object names as strings and calling methods on these objects when only a string containing the method name exists. The `Class.forName(String : classname)` method returns a `Class` object on which `getConstructor().newInstance()` can be called to create a new instance of a particular required class. Similarly, calling `invoke()` on a `Method` object (retrieved using the `getMethod()` method) calls a method on the desired object.

If this dynamic method invocation is performed within a new thread created by the ALS component, the AS can respond to remote method invocations from multiple simultaneous clients.

Bearer Network Interfacing

The `TwoPartyCallRBB` and `MultipartyCallRBB` classes encapsulate and abstract bearer network functionality, providing a simplified interface to service logic. Ideally, an actual existing bearer network would be used for proof of concept, showing the framework's applicability to existing infrastructures. The bearer network interface has the potential to be more complex than the framework itself due to the myriad of technologies available used for supporting this. However, there exist some Internet-based services which provide simplified interfaces for this purpose.

The service offered by the Internet service *Twilio* provides a RESTful HTTP web service interface into the public switched telephone network (PSTN), allowing two party and multiparty calls to be setup using simple invocations over HTTP. The service maps well to the two party call RBB. Whilst the multiparty call RBB (and hence the conference call service) was not implemented in this proof of concept, Twilio also provides functionality for this.

Call State Model

The call state model RBB uses a call state model with defined allowable state transitions. In the case of an invalid transition occurring, service logic should be able to handle this gracefully. The detection of these invalid transitions is best performed using Java's exceptions, whereby each call to `transition()` checks if the transition is valid based on the current state of the model and throws an exception if it is not.

Maps

Maps are used through the framework and provide a means for decoupling service logic from the underlying supporting infrastructure of the framework. In many cases where a unidirectional mapping is required, the `java.util.Map` generic class functioned well to support this. However, the Java standard edition is not shipped with bidirectional map functionality. The `BidiMap` class of the Apache Commons' extension to the Java Collection's framework fills this gap. `BidiMap` is a simplified bidirectional mapping implementation which is very similar to the standard basic `Map` class.

A.1.2 Terminal

Service logic is distributed to terminals as well. For the proof of concept a basic mobile framework for supporting this had to be developed. In fact, most of the basic functionality of the AS has to also be supported on the terminal, including ALS and dynamic method invocation.

Main Implementation Language and Platform

The Python programming language is supported on the Symbian operating system (OS). For this implementation, the Python language provided adequate support, using its very strong but simple language features to allow Nokia cellular phones to act as the terminal in the proof of concept.

Application Layer Signalling

The `asyncore` built-in Python module provides the asynchronous socket functionality required to allow the terminal to interface with the AS. In a similar way to the AS's implementation of ALS, sending a message only requires a simple call to a `send_async_message()` method. Similarly, receiving messages can be handled asynchronously and handled by newly spawned threads.

Dynamic Method Invocation

Python is a strongly reflective language. Calling a method on an object when only a string naming the object's class and desired method exists, depends on a simple invocation of the `getattr()` built-in function.

Telephony

The main reason that Symbian Python was selected for the terminal implementation is because it provides a very simplified interface into the telephony of the hosting cellular phone. Since two party and multiparty calls are implemented using 3rd-party invocation, the originating terminal has to be able to answer an incoming call automatically, since it is the terminal actually requesting the call (it is of little sense to have to make the user manually answer a call he or she initially requested).

The Python implementation on the Symbian platform has the `telephone` module which allows for the setting of a call state handler function which is called in response to the terminal ringing. The implemented function can be set to answer an incoming call automatically when the terminal begins to ring by calling `telephone.answer()`. Whilst not a perfect implementation, since the cellular phone still rings at least once, it still provides the expected behaviour required when a call is made.

A.2 Result

The resulting terminal and AS implementations allowed for basic testing of the presence service integrated with the two party call service. Using a graphical user interface (GUI) on the cellular phone, the user is able to set his or her presence and

make phonecall's by entering E.164 numbers. All remote functionality is invoked by having method calls marshalled into XML and sent over sockets to the AS and vice versa.

The proof of concept functioned according to design and showed firstly how service logic can be implemented very simply and secondly that such a framework is a viable solution to service development.

References

[1] ITU-T, *Recommendation Y.2201: Next Generation Networks - Service aspects: Service capabilities and service architecture*, International Telecommunications Union, Geneva, April 2007.

[2] 3GPP, *TS 23.228: Technical Specification Group Services and System Aspects; IP Multimedia Subsystem (IMS); Stage 2*, 3rd Generation Partnership Project, March 2009.

[3] C. Gourraud and X. Weibel, "The IMS Application Layer(s)," in *Proceedings of the 10th International Conference on Intelligence in Service Delivery Networks - ICIN 2006*, Bordeaux, France, May/June 2006.

[4] ITU-T, *Recommendation Q.1204: Intelligent Network Distributed Functional Plane Architecture*, International Telecommunications Union, Geneva, March 1993.

[5] R. Jain, J.-L. Bakker, and F. Anjum, *Programming Converged Networks: Call Control in Java, XML, and Parlay/OSA*. USA: John Wiley & Sons Inc, 2005.

[6] G. Camarillo and M. A. García-Martín, *The 3G Multimedia Subsystem (IMS) – Merging The Internet and The Cellular Worlds*. England: John Wiley & Sons Ltd, 2006.

[7] C. Gourraud. (2007, April) "A Classification of IMS Critics, Part 1". The IMS Lantern. Last accessed 15 July 2009. [Online]. Available: http://theimslantern.blogspot.com/2007/04/classification-of-ims-critics-part-1.html

[8] H. Hanrahan, *Network Convergence - Services, Applications, Transport, and Operations Support*. John Wiley & Sons, Ltd, 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1995.

[10] R. Christian and H. Hanrahan, "Towards a Standards-based Service Delivery Platform using Service Oriented Reference Points," in *Convergent Service Delivery - Proceedings of the 10th International Conference on Intelligence in Service Delivery Networks - ICIN 2006*, Bordeaux, France, May 2006, pp. 96–101.

[11] R. Christian and H. Hanrahan, "Structuring the Next Generation Network Using a Standards-based Service Delivery Platform," in *Innovations in NGN: Future Network and Services, 2008. K-INGN 2008. First ITU-T Kaleidoscope Academic Conference*, May 2008, pp. 33–40.

[12] D. Booth *et al.*, "Web Services Architecture," World Wide Web Consortium, W3C Working Group Note, February 2004, last accessed 17 August 2010, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[13] P. Moodley and H. Hanrahan, "Evaluation of the IMS-based MSF architecture against network architectural requirements," in *Proceedings of the Southern African Telecommunications and Networks Conference - SATNAC 2008*, South Africa, September 2008.

[14] 3GPP, *TS 23.141: Technical Specification Group Services and System Aspects; Presence Service; Architecture and functional description (Release 8)*, 3rd Generation Partnership Project, June 2008.

[15] Sun Microsystems, *JAIN SLEE (JSLEE) 1.1 Specification, Final Release, JSR 240*, July 2008.

[16] Microsoft Corporation, *Microsoft Service Delivery Platform Technical Overview*, 2003.

[17] IBM. IBM Rational Unified Service Creation Environment. Last accessed 15 July 2009. [Online]. Available: http://www-01.ibm.com/software/rational/solutions/telecom/

[18] OMA, *OMA Service Environment - Approved Version 1.0.5*, Open Mobile Alliance, October 2009. [Online]. Available: http://www.openmobilealliance.com

[19] B. Fricke, "The Development of a Structured Approach to Service Provisioning in a Parlay Environment," University of the Witwatersrand, Johannesburg, South Africa, Research Report, 2007.

[20] ETSI, *TS 101 314: Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) Release 4; Abstract Architecture and Reference Points Definition; Network Architecture and Reference Points*, European Telecommunications Standards Institute, Sophia Antipolis, France, September 2003.

[21] B. Fricke and H. Hanrahan, "The Development of a Structured Approach to Service Provisioning in a Parlay Environment," in *Proceedings of Southern African Telecommunication Networks and Applications Conference (SATNAC) 2006*, Cape Town, South Africa, September 2006.

[22] 3GPP, *TS 29.198-4-1: Technical Specification Group Core Network; Open Service Access (OSA); Application Programming Interface (API); Part 4:*

Call control; Sub-part 1: Call control common definitions (Release 9), 3rd Generation Partnership Project, December 2009.

[23] Sun Microsystems, *JAIN JCC Specification, JSR 21*, 2002, last accessed 17 August 2010. [Online]. Available: http://jcp.org/en/jsr/detail?id=21

[24] D. Vannucci and H. Hanrahan, "OSA/Parlay-X Extended Call Control Telecom Web Services," in *Proceedings of the 11th International Conference on Intelligence in Service Delivery Networks - ICIN 2007*, Bordeaux, France, October 2007.

[25] D. Horwitz and H. Hanrahan, "The Case for a Simplified NGN Application Layer Infrastructure," in *Proceedings of Southern African Telecommunication Networks and Applications Conference (SATNAC) 2009*, Mbabane, Swaziland, August–September 2009.

[26] Parlay Group, "Parlay/OSA and the Intelligent Network," The Parlay Group, White Paper, 2005.

[27] G. Mamais, M. Perdikeas, and I. Venieris, "Object Oriented Design Methodologies," in *Object Oriented Software Technologies in Telecommunications: From theory to practice*, I. Venieris, F. Zizza, and T. Magedanz, Eds. England: John Wiley & Sons, Ltd, 2000, pp. 49–72.

[28] TINA-C. Principles of TINA. The Telecommunication Information Networking Architecure Consortium. Last accessed 17 August 2010. [Online]. Available: http://www.tinac.com/about/principles_of_tinac.htm

[29] H. van Vliet, *Software Engineering – Principles and Practice*, 2nd ed. John Wiley & Sons, Ltd, 2000.

[30] OMA, *Presence SIMPLE Requirements - Candidate Version 2.0*, Open Mobile Alliance, December 2008. [Online]. Available: http://www.openmobilealliance.com

[31] M. Day *et al.*, *A Model for Presence and Instant Messaging*, Internet Engineering Task Force, February 2000, RFC 2778.

[32] ITU-T, *Recommendation Q.1214: Distributed Functional Plane for Intelligent Network CS-1*, International Telecommunications Union, Geneva, October 1995.

[33] 3GPP, *TS 29.198-4-5: Technical Specification Group Core Network and Terminals; Open Service Access (OSA); Application Programming Interface (API); Part 4: Call control; Sub-part 5: Conference call control Service Capability Feature (SCF) (Release 9)*, 3rd Generation Partnership Project, December 2009.

[34] 3GPP, *TS 29.198-4-3: Technical Specification Group Core Network; Open Service Access (OSA); Application Programming Interface (API); Part 4: Call*

control; Sub-part 3: Multi-party call control Service Capability Feature (SCF) (Release 9), 3rd Generation Partnership Project, December 2009.