UNIVERSIDADE da MADEIRA

wContact

Improving Social Communication

Nuno David Santos

Master's thesis in Software Engineering

wContact

Improving Social Communication

Funchal, 2010

Nuno David Santos

University of Madeira

Supervisor: Ian Oakley

Abstract

Contact management today is ubiquitous and multi-channel: phone, IM, email, VOIP to name but a few. However, the contact management technology commonly in use today has not changed significantly in many years. It remains far from an ideal service, falling down on issues such as its utility, ease of interaction, the efforts required to maintain it and its reliability.

Another important factor relies how mobile communication devices have become more commonplace increasing the potential to be interrupted by them. Indeed, it has been argued that technologies such as the mobile phone have "reconfigured time and space". They have also fundamentally altered notions of availability, promoting a vision in which users are always-on and continually connected. While this confers many advantages it also increases the potential for disruption to users engaged in other activities or seeking rest

This document describes wContact, a system that provides a unified service of synchronized contact profiles (incorporating privacy controls) that contain all contact info. This is held in a distributed system (a cloud computing service) and broadcast over a data connection to address book clients. Ultimately this takes the form of a minimal social network where adding or removing contacts is equivalent to adding or removing a friend in a more conventional service such as Facebook.

The wContact is complemented with a system which automatically manages status on a mobile device. Its contribution lies in the adoption of a number of recent technological advances to create a realistic framework for a sophisticated mobile context-sensitive tool to capture activity and infer and share availability. It does so by proposing a context model based on the integration of multiple sensor inputs (e.g. microphone and accelerometer) and internal device state (e.g. battery level) which is broadcasted.

In summary, wContact represents a substantial improvement over today's systems in how contacts are treated and managed. It focuses on simplifying the way they are exchanged, increasing the quantity and quality of contact details and providing awareness clues to better mediate engagement with contacts. Finally, to ensure the durability of contacts the system provides a seamless automatic update process.

Table of Contents

Abstract 3

Index of Figures 7

Index of Tables 8

CHAPTER1 - Introduction 9

1.1 Motivation 10

1.2 Objectives 11

1.3 Document Structure 11

CHAPTER 2 - State of the Art 13

2.1 Communication 13

2.2 Virtual Communities 14

2.3 Cloud Computing 17

2.3.1 SaaS – Software as a Service 18

2.3.2 PaaS – Platform as a Service 19

2.3.3 IaaS - Infrastructure as a Service 19

2.3.4 Architectural Map 20

2.3.4 Summary 22

2.4 Sharing Awareness in Mobile Devices 22

CHAPTER 3 - Awareness 25

3.1 User Research 25

3.1.1 Online Survey 25

3.1.2 Live Interview 26

3.2 Sensors 26

3.3 Activity Logger 27

3.4 Analyzing Data 28

3.4.1 Accelerometer 28

3.4.2 Microphone 30

3.5 Awareness conclusions 30

CHAPTER 4 - System Design 31

4.1 Activity Map 31

4.2 Activity Profiles 32

4.2.1 User Updates 32

4.2.2 Automatic Detection 32

4.2.3 Certificate Manager 33

4.2.4 Privacy Manager 33
4.2.5 Storage 33
4.2.6 Notification Manager 33
4.3 Participation Map 33
4.4 Role Profiles 34
4.4.1 R01 Auto detection status 34
4.4.2 R02 Certificate validation 34
4.4.3 R03 Certificate ok 34
4.4.4 R04 Update datacenter 35
4.4.5 R05 Notify clients 35
4.4.6 R06 Privacy filter 35
4.4.7 R07 Update data 35
4.4.8 R08 Manual update 35
4.5 Use Case Model 36
4.6 Task Cases 37
4.7 Quality Attribute Workshop 40
4.7.1 Non-functional requirements 40
4.7.2 Quality Attributes 40
4.7.3 Scenarios 40
4.7.5 Attribute Driven Design 43
4.8 Architecture 44
CHAPTER 5 - System Implementation 46
5.1 Material and Methods 46
5.1.2 Google App Engine 46
5.1.3 Android OS 47
5.1.4 REST (Representational State Transfer) 49
5.1.5 XMPP (Extensible Messaging and Presence Protocol) 50
5.1.6 Summary 51
5.2. System Overview 52
5.2.1 Authentication / Registration 52
5.2.2 Edit Profile 52
5.2.3 Privacy 53
5.2.4 Remove User 53
5.2.5. Define Settings 53

5.2.6. Change Status 54
5.2.7 Remove Profile 54
5.2.8 Background Process 54
5.2.9 Summary 54
5.3 Server Implementation 55
5.3.1 Datastore 55
5.3.2 Servlets 58
5.3.3 Cloud Response 62
5.3.4 Certificate Manager 62
5.3.5 Privacy Manager 62
5.3.6. Summary 63
5.4 Client Implementation 64
5.4.1 Database 64
5.4.2 Layout 65
5.4.3 Cloud 71
5.4.4 Contact 73
5.4.5 Status 73
5.4.6 Background Service 73
5.4.7 Notifications and Invitations 74
5.4.8 Awareness 74
5.4.9 Widget 75
5.4.10 Summary 76
CHAPTER 6 - Conclusion 77
Bibliography 80

Index of Figures

Figure 1 – A typology of virtual communities 14
Figure 2 - Cloud computing basic approach 18
Figure 3 -Architectural map of cloud computing 21
Figure 4 - Activity recognizer structure 27
Figure 5 - Activity Logger Screenshots 27
Figure 6 - Activity Logger web site screenshot 28
Figure 7 - Dataset web site screenshot 28
Figure 8- Data collected while walking 29
Figure 9 - Data collected while driving 29
Figure 10 - Activity Map 32
Figure 11 - Participation Map 34
Figure 12 - Manual update role task cases 36
Figure 13 - Certificate validation role task cases 36
Figure 14 - Certificate ok role task cases 37
Figure 15 - Create notification role task cases 37
Figure 16 - Notification queue role task cases 37
Figure 17 - System architecture 44
Figure 18 - Android OS 48
Figure 19 - Database diagram 55
Figure 20 - PMF class 56
Figure 21 - CertificateManager class 62
Figure 22 - Privacy representation 63
Figure 23 - Database hierarchy classes 64
Figure 24 - IDatastore interface 65
Figure 25 - IActivityView 66
Figure 26 - Tabs hierarchy 68
Figure 27 - Cloud access hierarchy 72
Figure 28 - CloudResponse class 72
Figure 29- Home screen widget 76

Index of Tables

Table 1 – Stage of Group Development 15
Table 2 - Add contact task case 38
Table 3 - Delete contact task case 38
Table 4 - Update profile task case 38
Table 5 - Accept contact invitation task case 38
Table 6 - Decline contact invitation task case 38
Table 7 - Get all contact list task case 38
Table 8 - Validate certificate task case 39
Table 9 - Add contact task case 39
Table 10 - Delete contact task case 39
Table 11 - Update profile task case 39
Table 12 - Accept contact invitation task case 39
Table 13 - Decline contact invitation task case 39
Table 14 - Get all contact list task case 39
Table 15 - Create notification task case 39
Table 16 - Apply privacy filter task case 40
Table 17 - Send notification task case 40
Table 18 - Quality attribute scenario #1 41
Table 19 - Quality attribute scenario #2 41
Table 20 - Quality attribute scenario #3 41
Table 21 - Quality attribute scenario #4 42
Table 22 - Quality attribute scenario #5 42
Table 23 - Quality attribute scenario #6 42
Table 24 - Quality attribute scenario #7 43
Table 25 - Contact details types 57
Table 26 - Notifications types 58
Table 27 – Web services description 59

CHAPTER1 - Introduction

Contact management today is ubiquitous and multi-channel: phone, IM, email, VOIP to name but a few. However, the contact management technology commonly used has not changed significantly in many years. It remains far from an ideal service, falling down on issues such as its utility, ease of interaction, ease of maintenance and reliability.

For many years, contact management was based entirely on a single telephone number and/or address noted down in a handwritten address book. Although technology has moved on, this approach still remains the dominant metaphor. When users aim to add a new contact in a modern system, they still manually insert it into a digital list and need to double-check its correctness by hand.

Although functional, this paradigm is under strain. New ways of communicating with others (e.g. email, chat, virtual communities, blogs, etc.) each require each a unique id in order to establish communication; many users now have multiple IDs and accounts. The substantial increase in information being stored about a particular person, or contact, makes it challenging to maintain a comprehensive list of their addresses on all the possible channels of interaction. Furthermore, the complexity of the modern contact information (typically mixing numeric with alphanumeric characters and possible symbols) is increasing, adding to the overall challenge of maintaining an accurate and comprehensive set of contacts.

The transformation of contact management from analog to digital form (from printed to virtual address books) brought many advantages such as the rapid search for a particular contact, unambiguous access to the identifiers (often complex and non-intuitive strings) and the possibility to synchronize information between multiple devices. Synchronization in particular has brought immense benefits, as it allows users to access all their contacts where needed (on multiple devices/applications) and minimizes the risk of losing them (e.g. due to hardware failure). However synchronization cannot guarantee that contact information is accurate and each individual's data changes regularly as they engage with new mediums, or alter their identities on old ones.

Consider the example of a user who takes a new phone number. It is this of contacts responsibility to convey this new number to his contacts. Depending on the number they have, this task can become onerous and even costly. Furthermore, even when this task is achieved, problems can still occur. These include a failure to remove old contact details when new ones are added (causing confusion between the two items) and an error or omission in entering the new contact information (causing a complete breakdown in the communication).

This example represents a real life situation when a user changes his or her phone number and takes great care to convey this to their contacts. However, others contact information, such as a personal website, are often not treated with this level of attention and changes often go un-discussed and unrecorded.

Another issue with contacts is that they can be seamlessly passed to third-parties without the knowledge of their owner. It is common to share the information relating to one contact with another. However, this situation represents an undesirable one in which users have no control of their privacy.

In summary contact information is the basic means required for establishing communication. Although this remains true, technological advances have radically changed the way we communicate, opening up novel channels and mediums and changing the ways we plan and spend our time. Indeed, it has been argued that technologies such as the mobile phone have "reconfigured time and space" [30]. They have also fundamentally altered notions of availability [19], promoting a vision in which users are always-on and continually connected. While this confers many advantages (e.g. greater flexibility and levels of perceived security) it also increases the potential for disruption to users engaged in other activities or seeking rest. Addressing this problem space is a considerable body of work on human interruptibility [17], both in terms of how it can be reliably determined (via sensing systems and context models) and also in how it is understood and managed [e.g. 11]. Commonplace communication tools, such as instant messaging services, reflect the importance of this issue by allowing users to explicitly set a status indicating availability. This thesis suggests that sharing contact information and sharing availability (or awareness information) are concerns that are tightly related.

The solution this thesis defines to the problems of contact management relates to the distributed storage of the data. This conveys a number of advantages, most notably a mechanism by which information can be seamlessly updated and shared between users. This includes both relatively persistent information (such as phone number or user IDs) and also transient information such as status (either manually entered or derived automatically). To attempt a final novel aspect of awareness detection is the explicit choice to avoid capturing or using explicit location information, thereby side-stepping many of the privacy issues this ability raises (see Consolvo *et al*. [6] for a general discussion) and also the fact that this technology is typically only available in specific environments (e.g. outdoors for GPS or in sites with a high density of WiFi beacons).

1.1 Motivation

The underlying motivation for this work is that humans are social beings; communication is a core part of our existence. With the advanced technology available today it is unjustifiable that one cannot communicate with others due to poor contact management systems.

As modern life becomes increasingly mobile and individuals frequently end up away far from family and friends. Although communication technologies bridges these gaps and can keep people together, the difficulty of maintaining all the information required to get in touch with others, and the complexity of the forms of the IDs required to achieve this are a significant barrier. Often a single wrong character stored or entered can prevent the realization of a communication.

A further difficulty with modern communication channels is that their diversity causes problems. It is hard for two users to exchange all relevant communication IDs (e.g. phone, IM, VOIP, email) simply due to the quantity of systems present. This problem is exacerbated when users change or update their IDs and need to convey this to all their contacts.

Finally, this thesis also seeks to create more parity in communication. It is, by nature an act with at least two parties. However, prior to initiating, one party is typically unaware of

the availability of the other, thus increasing the likelihood of making an unwanted intrusion. This thesis is motivated to create better technologies to share information to avoid this undesirable situation.

1.2 Objectives

The main objective of this thesis is to redesign and improve how contact management is performed. It aims to progress from the arduous activity of users independently managing their own list of contact data to a model in which users simply publish their contact information to a distribution store. By managing connections to other users in that store, each user is able to gain access to accurate contact information at minimal cost.

A secondary objective is to share the availability status of users with their contact information in order to ensure that there is an effective improvement in the quality of communication with the system. The goal of this work is to prevent the disruptions caused by communications. Another objective in this area is to increase the control users have over the privacy of their contact information, by allowing them to define levels of privacy to each profile element the system should allow them to control which of their contacts have access to which of their information.

The thesis will also implement a system to automatically manage status on a mobile device, thus reducing user effort to maintain status. Its contribution lies in the adoption of a number of recent technological advances to create a realistic framework for a sophisticated mobile context-sensitive tool to capture activity and infer and share availability. It does so by proposing a context model based on the integration of multiple sensor inputs (e.g. microphone, accelerometer and compass) and internal device state (battery level, device profile).

In summary, the overall objective is to provide a unified service of synchronized contact profiles (incorporating privacy controls) that contain all contact info. This is held in a distributed system (a cloud computing service) and broadcast over a data connection to address book clients. Ultimately this takes the form of a minimal social network where adding or removing contacts is equivalent to adding or removing a friend in a more conventional service such as Facebook.

1.3 Document Structure

This thesis is organized as follows:

Chapter 2 – State of the Art

This section analyzes the various technologies and existing applications that manage contacts and which are aware of the situation of the user. It also explores the concepts of cloud computing and virtual communities to provide a foundation for this work.

Chapter 3 – Activity Recognizer

This chapter presents the research conducted on awareness and the systems produced. These include a small application for logging data and a description of the awareness system and the mechanisms for creating status and sharing activities.

Chapter 4 – System Design

This chapter includes the models describing the entire system and architecture. The processes used in this chapter include: Activity Map, Activity profiles, Participation Map, Role Profiles, Use case model, Task Cases, Quality Attribute Workshop

Chapter 5 - System Implementation

This chapter presents a description of the application development process as well as class structure and sequence diagrams.

Chapter 6 – Conclusion

This section closes the thesis with an analysis of the prototype as well as personal experience. It also proposes ideas for future work.

CHAPTER 2 - State of the Art

This chapter reviews the relevant literature to the main topics covered in this thesis. It is divided into four topics. The first two cover communication and virtual communities. The third covers cloud computing, a technological platform on which such communities can be implemented and the fourth covers awareness technologies, a core component of many virtual communities. It particularly focuses on awareness captured and conveyed via mobile devices.

2.1 Communication

In 340 b.c. Aristotle stated that "man is a social being" and suggested that "social life in a community is a necessary condition for a man's complete flourishing as a human being". Although 2300 years have passed the reality remains and man is still naturally needy, requiring communication with others to achieve happiness and fulfillment.

Reflecting the importance of this drive are many innovations in communication systems throughout history. For example in the XVI century, postal services were widely introduced. In these early systems messengers traveled on foot or horseback, often through dangerous regions or along hazardous routes.

In the XIX century address book systems were created to inform the general population about particular people or commercial or administrative establishments. Later in that century Antonio Meucci invented the "telettrofono", now known as the telephone, and immediately perceived the need to produce a book containing the numbers of subscribers.

Today, more than ever, our daily activities involve managing interpersonal communications but this activity is poorly understood. Whittaker et al [36] define the contact management problem as maintaining knowledge of one's contacts. Once people are exposed to an unmanageable number of potential contacts and it becomes challenging to store and access detailed information about them.

Whittaker et al. divides these problems into three categories: Contact Selection, Data Entry and Diversity of Tools. Contact selection takes place upon receiving a communication from a novel source, and relates to determining whether or not to store this data. Data entry makes reference to the cost of acquiring (and accurately maintaining over time), complex numerical and alphanumerical contact details - data containing an incorrect character is of little use. Diversity of tools refers to the fact that people track their contacts using a variety of tools including paper address books, PDAs and email and that these tools are ubiquitous and often incompatible. Consequently, large efforts need be made to keep them synchronized and ensure there is no loss of valuable data.

In order to solve these problems Whittaker et al. conducted an experiment to investigate how people identify important contacts based on relationship between communication history, communication style and user's perception of contact importance. While frequency was confirmed (i.e. people interacted more frequently with important than unimportant contacts) there were also exceptions to this rule in the form of important, but occasional contacts. Longevity was also a strong predictor of contact importance – users may

have had significant interaction with a contact in the past but, due to social or work changes, they may no longer be important.

This study concluded that users are exposed to a large number of potential contacts, but the onerous nature of data entry means that they end up being conservative about who they add to their contact management systems. However, a system in use over the long term typically contains many out of date contacts – one significant finding from this research is that contact information changes frequently and that address books present this irrelevant information together with salient, up to date information. The authors conclude with the suggestion that current address book systems would benefit from better tools to 'clean up' contact archives based in an algorithm to identify stale contacts and relegate those to a secondary interface view.

2.2 Virtual Communities

Nowadays, with the boom of Internet, more people participate in virtual communities than engage in online purchases [16]. The popularity of virtual communities reflects the fact that individuals are using new technologies to fulfill both social and economic goals [37]. There are a large number of definitions for virtual communities, but Porter [26] provides a concise summary. He defines a Virtual Community as an aggregation of individuals or business partners who interact around a shared interest, where interaction is at least partially supported or mediated by technologies and guided by consensual protocol norms.

Porter also proposed a typology for virtual communities, see figure 1, which includes two first-level categories: Member-initiated and Organization-sponsored where the first are community established, and remain managed by members while second are communities that are sponsored by either commercial or non-commercial organizations. A second level of this topology is based on the general orientation of the relationship where importance is revealed once several disciplines recognize the importance of relationships in the context of virtual communities.

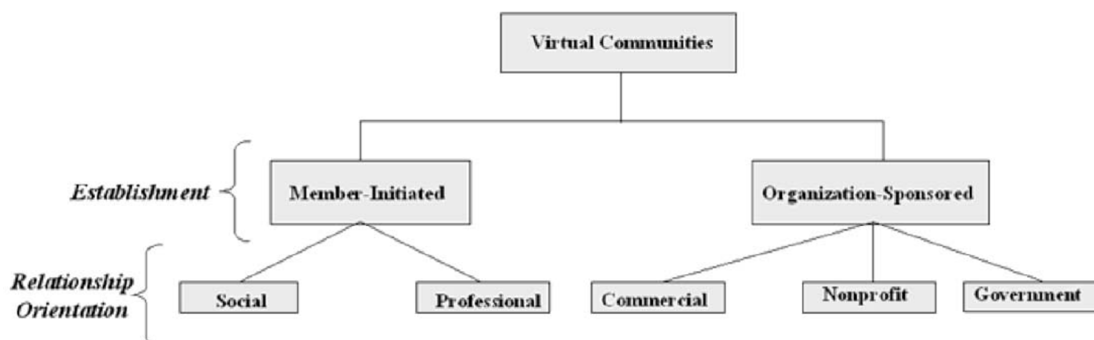


Figure 1 – A typology of virtual communities

Porter also proposes five attributes that can be used to characterize virtual communities: Purpose (describes the specific focus of discourse or focal content of communication among community members), Place (defines the location of interaction:

completely virtual or partially virtual), Platform (the technical design of interaction that might enable synchronous, asynchronous or both forms of communication), Population (group structure and type of social tie supported) and Profit Model (whether a community creates tangible economic value where value is defined as revenue-generation). These attributes give researchers a consistent and practical way to describe virtual communities and, from a managerial perspective, virtual communities with unique combinations of attributes are likely to have different critical success factors and associated outcomes for both members and organizational sponsors.

Gupta [14] presents an approach to virtual communities based on the notion that these are places on the web where people can find and then electronically "talk" to others with similar interests, playing an important role in many aspects of a member's life, from forming and maintaining friendships and romantic relationships to learning, making purchasing decisions and consuming products or services. Virtual communities are more concerned with human relationships when compared to traditional communities (neighborhood, town or region based), because the members are not physically bound together – their social ties remain at the core of the community.

Gupta cites Tuckman [34] any community, whether traditional or virtual, evolves through five stages namely forming, norming, storming, performing and adjourning [34]. Table 1 represents those stages:

	Group Structure The pattern of interpersonal relationships; the way members act and relate to one another.	**Task Activity** The content of interaction as related to the task at hand.
Forming: orientation, testing and dependence	Testing and dependence	Orientation to the task
Storming: resistance to group influence and task requirements	Intragroup conflict	Emotional response to task demands
Norming: openness to other group members	Ingroup feeling and cohesiveness develop; new standards evolve and new roles are adopted	Open exchange of relevant interpretations; intimate, personal opinions are expressed
Performing: **constructive action**	Roles become flexible and functional; structural issues have been resolved; structure can support task performance	Interpersonal structure becomes the tool of task activities; group energy is channeled into the task; solutions can emerge
Adjourning: **disengagement**	Anxiety about separation and termination; sadness; feelings toward leader and group members	**Self-evaluation**

Table 1 – Stage of Group Development

Then Gupta suggests that as a virtual community is a multi-disciplinary concept its benefits can also be viewed from various perspectives. These include:

- the Technology perspective of providing ubiquitous cheap and fast communication;
- the Business perspective of helping to establish a leading brand, increasing barriers to entry, developing critical mass and raising interest among customers;

- the E-commerce perspective in which trust (and the trust building capabilities of virtual communities) are an important factor;
- the Marketing perspective of providing a broad range of information from fellow customers;
- the Sociological perspective where knowledge exchange is an important benefit and individuals can either give information or acquire information;
- the Economic perspective reflecting that virtual communities can create revenue by charging usage fees, content fees and transactions and advertising fees;
- the Learning perspective with focuses on stimulating continued learning and nurturing a sense of fellowship and identity, and the ability to enhance the learning process by improving access to special stimulations and demonstrations.

As a tool to support a marketing relationship, the Internet is an efficient channel as it has unique capabilities to deliver high levels of personalization and customization to satisfy the needs of different market segments. It is widely acknowledged that virtual communities can help enhance long-term relationships between the costumer and companies. Morgan and Hunt [22] define relationship commitment, a concept central to relationship marketing, as exchange partners exhibiting maximum efforts to maintain a tie. In relationships in which the parties identify commitment among the exchange partners as key to achieving valuable personal outcomes, and they endeavor to develop and maintain this precious attribute. In the context of virtual communities, such investment is in the form of data sharing, knowledge sharing or sharing personal information.

Gupta also discusses issues related to virtual community dynamics like design of virtual communities and factors arising during their evolution. These include: the role of trust in enhancing virtual community activities; understanding virtual community characteristics in order to stimulate activities in virtual community; the formation of relational ties in virtual communities; and understanding the role of personalization in e-commerce. Other key issues are related to the application of virtual communities, such as understanding their potential industries like insurance, health care, education and tourism.

Gupta concludes that virtual communities have evolved from simple exchange systems to the currently extant rich and diverse web-based communities. They have advantages over face-to-face communities in that they are larger and more dispersed in space. They are mainly formed for four purposes: transaction, fantasy, interest and relationship. They can be beneficial to different disciplines such as business, e-commerce, marketing and education. Establishing such systems can aid relationship marketing, building relationship commitment, and enhancing store image and customer loyalty.

With the increasing popularity of these virtual communities and social networks, large-scale distributed systems become crucial enabling technologies. Cloud computing is a term coined for a recent trend toward service-oriented cluster computing that simplifies the use of large-scale distributed systems through transparent and adaptive resource management [5]. It provides simplification and automation for the configuration and deployment of an entire software stack relying on operating system virtualization implemented using high-level

language technologies, APIs and web services. It is to this topic that this literature review now turns.

2.3 Cloud Computing

In the beginning of the 90s, distributed computing technologies targeted at enterprise systems, such as the Open Group's DCE and the Object Management Groups CORBA, offered programmable interfaces to overcome the complexities of remote procedure calls [18]. Specifications as OMA (Object Management Architecture), in case of CORBA, served as reference architectures for services provided at different levels of the frameworks, providing guidance on how standardization of component interfaces penetrates up applications in order to create plug-and-play component software environments based on object technology [24].

A decade later the Grid Community proposed the Open Grid Services Architecture (OGSA), which was based around services in a distributed interaction and computing architecture assuring interoperability on heterogeneous systems to enable different types of resources to communicate and share information [33], as an effort to standardized Grid services interfaces and help adoption of Grid technologies across organizations at global scale.

A similar purpose for the Web is SOA (Service Oriented Architecture) inherent in the WS-I specifications that provide a loosely integrated suite of services that can be used within multiple business domains.

After a decade of maturing, SOA evolved beyond its traditional roots in RPC to encompass the forms of interaction found in representation state transfer. While enterprises focus on SOA, the data center and operations managers have server virtualization to increase and reduce cost. The provisioning of virtual servers positioned the evolution of SOA where services move into the cloud.

Discussing cloud computing is to discuss a service-oriented cluster computing based on Service-Level Agreements [4]. Cloud computing offers mainframe or better infrastructure through a small set of services delivered globally over the internet simplifying the use of large scale distributed systems through transparent and adaptive resource management, and providing simplification and automation for the configuration and deployment of an entire software stack.

Mobile applications traditionally are constrained by limited resources such as low CPU speed, small memory and a battery-powered computing environment. Cloud computing has the potential to radically alter how device software is developed and deployed while at the same time removing constraints on device functionality.

Cloud computing leverages high performance, relatively inexpensive commodity computers clustered closely together in a datacenter and connected to other similar data centers located globally as close to the users as possible [32]. It provides elastic computing infrastructure and resources which enable resource-on-demand and pay-as-you-go utility computing models [38].

Cloud computing relies on operating system virtualization for functionality and performance isolation supporting per-use or per-application customization of services typically implemented using high-level language technologies, APIs, and web services which present application developers with a small set of services providing a limited set of operations , reducing the learning curve for developers. Services are invoked using simple REST XML request or RSS, making them easy to use from any programming language.

Cloud computing is based on three levels as shown in Figure 2. These three models, Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) support flexible and efficient ways to augment computing, storage, and communication abilities for applications on resource-constrained devices and enable novel IT business models such as resource-on-demand and pay-as-you-go.

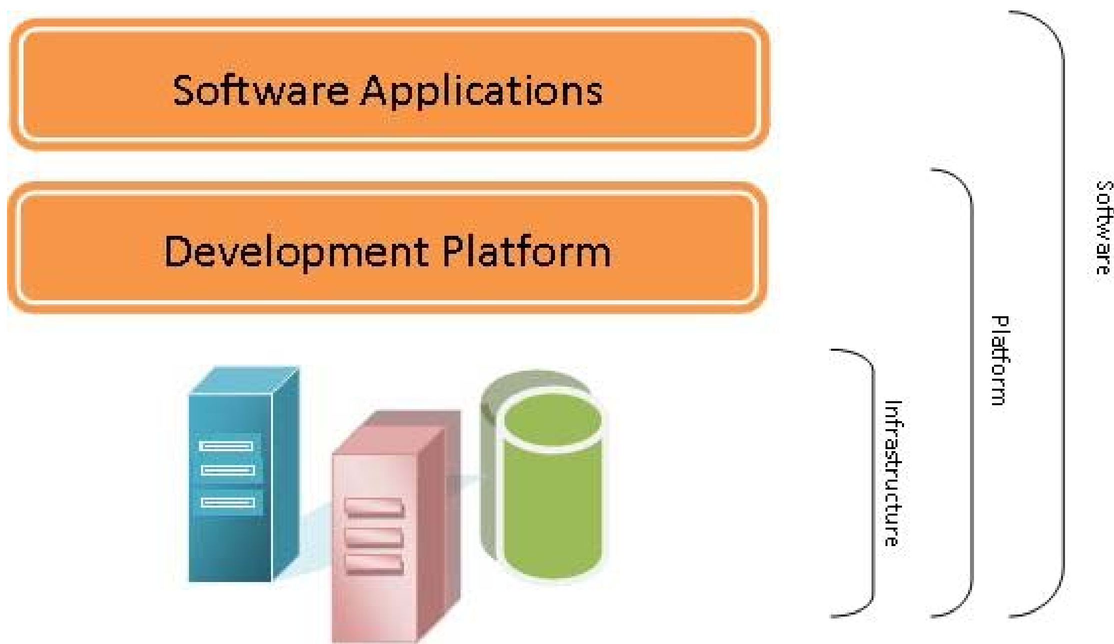


Figure 2 - Cloud computing basic approach

2.3.1 SaaS – Software as a Service

Software as a Service delivers application service over the internet buying deployment infrastructure as a service from utility providers. SaaS is also known as “software-on-demand” and it is currently the most popular type of cloud computing because of its high flexibility, great services, enhanced scalability and low maintenance. Yahoo mail, Google docs and CRM applications are all instances of SaaS.

SaaS describes systems in which high-level functionality is hosted by the cloud and exported to thin clients via the network. The software is hosted on centralized network servers to make functionality available over the web.

The main feature of SaaS systems is that the API offered to the cloud client is for a complete software service and not programming abstractions or resources. It is based on the concept of renting software from a service provider rather than buying hardware personally.

In summary, SaaS is a new delivery model which provides flexibility to both provider and the costumers and has the great benefit of being able to run the most recent version of applications without the pain of installation and upgrading. Instead, it is replaced by a simple request to run a specific version of the software eliminating the need to install hardware or software on client premises.

A major challenge for SaaS providers is ensuring the security and privacy of client data which is held outside the organization. SSL VPNs and client based encryption as well as mission critical fault tolerant, environmentally and physically protected datacenters are minimum capabilities that enable the use of external service based software.

2.3.2 PaaS – Platform as a Service

PaaS offers an integrated environment to design, develop, test, deploy and support custom applications following the pay-as-you-go of SaaS avoiding large up-front investments. This development platform allows users to write their code and the PaaS uploads that code presenting it on web.

PaaS refers to the availability of scalable abstractions through an interface from which restricted, network-accessible, applications written in high-level languages can be constructed. Popular examples of PaaS systems are Google App Engine and Microsoft Azure, where users typically test and debug their applications locally using a non-scalable development kit and then upload their programs to a proprietary, high scalable PaaS cloud infrastructure.

Basically there are four types of PaaS solutions: social application platforms, raw computing platforms, web applications and business application platforms.

2.3.3 IaaS - Infrastructure as a Service

IaaS describes a facility for provisioning virtualized operating systems instances, storage abstraction and network capacity under contract from a service provider. These virtualized instances are controlled and configured by users through root ssh connections.

Virtualization enables IaaS providers to offer almost unlimited instances of servers to customers and make cost-effective use of the hosting hardware, and moving infrastructure to the cloud means that we have the ability to scale as if we owned our own hardware and data center.

The great benefits of IaaS are usage based pricing, reduced costs, dynamic scaling and access to superior IT resources.

The most popular example of an IaaS-style computational cloud is the Amazon Web Services (AWS) which includes the Elastic Compute Cloud (EC2), Simple Storage System (S3), Elastic Block Store (EBS) and others APIs and charges per instance occupancy hour and for storage options at very competitive rates. Similar to PaaS systems, these rates are typically significantly less than the cost of owning and maintaining even a small subset of the resources that these commercial entities make available to users for application execution.

Users can buy the infrastructure according to the requirements at any particular point of time instead of buying the infrastructure that might not be used for months. This is the meaning of the pay-as-you-go model - users pay only for what they consume.

2.3.4 Architectural Map

Lenk [18] provides an architectural map of the cloud landscape shown in figure 3. Lenk proposed a first categorization of Cloud technologies as a stack of service types inspired by the "everything as a service" (XaaS) taxonomy. These included: "Infrastructure as a Service" (IaaS), "Platform as a Service" (PaaS) and "Software as a Service" (SaaS).

IaaS

At the lowest level, closest to hardware, Lenk distinguished two types of services: Physical Resource Set (PRS) and Virtual Resource Set (VRS) services. Both provide a management front-end API for a set of pool of resources in order to allow higher level services to automate setup and tear-down, demand-based scalability, fail-over and operating system hosting where primarily functionality includes starting and stopping individual resources, OS imaging, network topology setup and capacity configuration.

In the PRS layer (as Emulab and iLo) implementation is hardware dependent and therefore tied to a hardware vendor, whereas the VRS (as EC2, Eucalyptus, Tycoon, Nimbus and OpenNebula) layer can be built on vendor independent hypervisor technology (as with any virtualization system).

They justify that splitting these Result Set services into two types allows automated management of physical as well as virtual resources and different types of resources such as storage, network and computing nodes might need to be virtualized in different ways.

One level higher up in stack, but still in IaaS layer they distinguished three types of Basic Infrastructure Services: computational, storage and network.

PaaS

Lenk categorized PaaS into Programming Environments and Execution Environments.

They use examples such as Sun's project Caroline, Django Framework, Google's App Engine Joyent's Reasonably Smart and Microsoft Azure as systems that are an Execution Environment PaaS and typically encompass a Programming Environment PaaS.

SaaS

This layer contains all applications that run in the cloud and provide a direct service to the costumer. Application developers can use either the PaaS layer to develop and run the applications or directly the IaaS layer.

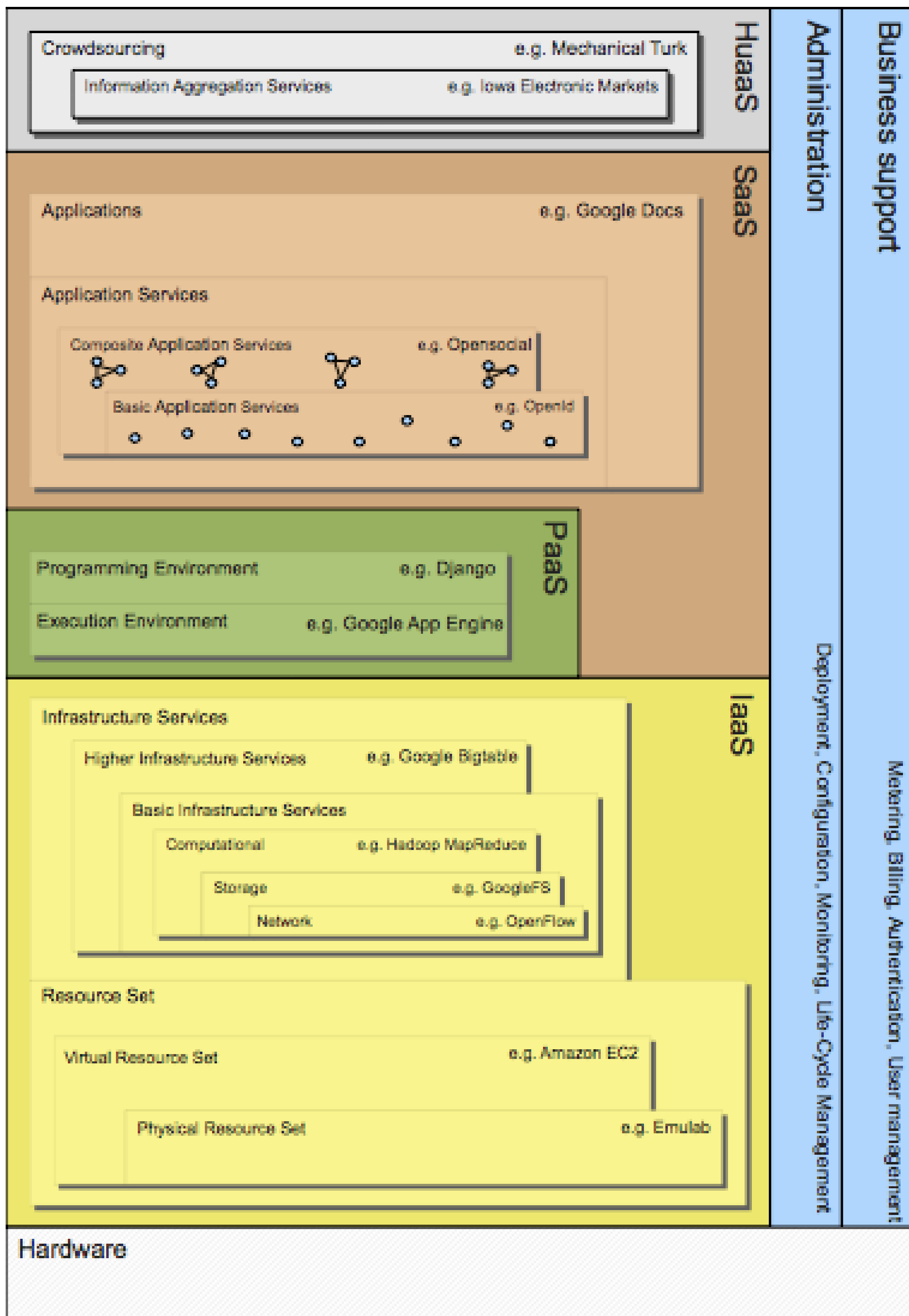


Figure 3 -Architectural map of cloud computing

Lenk divided this layer into Basic Application Services (such as OpenId and Google Maps) and Composite Application Services which also featured mashed-up of other systems.

Opensocial as the prominent example of the later case – it allows entire social networks (such as MySpace) to be used as Basic Services.

HuaaS

HuaaS is the top-most layer in Lenk's stack and is reference to the fact that some services rely on massive-scale aggregation and extraction of information from crowds of people where each individual in the crowd can may use whatever technology or tools he or she sees fit to solve the task.

Croudsourcing is shown in HuaaS layer and makes reference to the fact when human intelligence is used to contribute arbitrary services such as on-demand sub-task solutions

Information Aggregation Services is another category on stack and refers to when human intelligence aggregation services are more controlled and more targeted at predicting events or promoting ideas, since they all aim at producing a single aggregate number representing the popular opinion of the crowd in various ways.

2.3.4 Summary

In summary, cloud computing is a promising new architecture for performing computation, storing data and processing information. By allowing the distribution of, and easing the access to, a wide range of computational resources, innovations such as service based use and the decoupling of computing from specific resource limited devices (such as mobile phones) can be achieved. Recent works to classify and distinguish the different existing varieties of cloud system have helped delimit the possibilities of the field and understand the full scope of the advantages the general approach can confer.

2.4 Sharing Awareness in Mobile Devices

A second focused topic addressed in this thesis is the idea of sharing awareness between users who own a mobile device. Research in the area of novel communication technologies is high profile, potentially high impact and broad in scope. Key social issues such as availability, interruptibility, information (and interaction) overload and privacy have attracted considerable attention in diverse settings. For example, the Connector system [9] adjusts availability settings on a smart phone using a context model derived from augmented smart rooms populated with audio and video recording systems capable of performing speech and face recognition. By modeling and recording users' activities, tasks and social context, Connector seeks to automatically configure the response of their mobile phone to incoming calls and text messages from different sources. A motivating example is that, whilst in a meeting, calls will only be accepted from individuals with VIP status.

However, Connector's reliance on installed infrastructure makes it unfeasible in many situations. Alternative approaches have included inferring activity solely from usage history and context models (e.g. Calls.calm [25]) or using the capabilities built into modern mobile phones to create standalone sensing systems. An example of this is Connecto [2], a GPS-based location aware "friend-finder" which also shared information relating to place tagging, status updates and ringing profiles. A qualitative study revealed a range of sharing behaviors, including the adaption of place tagging to disseminate activities (such as shopping) as simple

narratives. The authors highlight the potential privacy issues inherent in the sharing of actual locations. Other work, such as Live AddressBook [21], a distributed address book in which users could modify their contact information, add custom status text and specify location and availability, has explored the use of status sharing through an address book metaphor. This system relied exclusively on user entered data (relating to location, status and availability), a fact which ensured the accuracy of this material at update time but led to irregular and infrequent updates. A user study revealed users found the system useful and tended to update status semi-regularly (mean of 1.4 updates per day, with 30% in response to a direct system prompt), but that they rated the accuracy of contact information, which varied by location, in the system to be very important. Consequently, the authors were critical of automatically detected location information, pointing out that inaccuracies are an inevitable part of automatic context-sensing systems.

Regardless of source, researchers agree that sharing context can support communication. For example, Mihalic [20] states that mobile systems that use information based on social context can provide a less obstructive and more natural interaction. They identify three dimensions of social context: relation type, mood and communication context and channel. The type relations are: partner, family members, friends, colleague and client. The context and channel relate to location, surrounding and time.

However, they are also negatives to automatic sharing of information. For instance, when sharing location users experience a tradeoff: while disclosing one's location to another user can be valuable it also has risks [6]. Consolvo used the Westin/Harris Privacy Segmentation Model in a location-sharing privacy evaluation and concluded that participant's chose to respond with relatively vague location information. They reported this would be most useful to the entity requesting the information. However, while vague details such as neighborhood name, city, and state can provide useful information, participants also felt that sharing this information might be intrusive; participants mentioned concerns relating to stalking, being monitored and Big Brother. Wang [35] use a comparative study in end-user place annotation concluding that "people usually think and speak in terms of places, which adds personal, environmental and social meaning to a location" and that "manual annotation is still required to complement automated methods". Heyer [15] complement this study by reporting that users mostly relied on pre-populated locations then setting their location to a specific physical one, and that presence notifications are textual and free form, rather than referring to actual locations.

Another important aspect to consider is the user knowledge of when and to whom information is shared. Reily [27] reports that users are not aware of the privacy implications of their actions in digital systems. To counteract this, Smith [31] introduces the concept of a crowd-sourced notion of reputation because on a community of individuals involved in establishing collective ratings for members (e.g. ratings for merchants and buyers on eBay).

One conclusion from this research is that although insufficient access to availability and activity information can lead to unwanted interruptions, providing too much data can be intrusive [13]. In an influential paper, Erickson and Kellog [10] suggest that communication

tools should seek to achieve social translucency, an equilibrium between visibility, awareness of others and accountability.

The awareness work in this thesis is based on the metaphor of the shared contacts developed in Live Addressbook, but combines this with the reliance on automatic sensing and context inference prominent in systems such as Connector and Connecto. In this way, it seeks to combine the best aspects of both these approaches and update them to include technologies currently available in the consumer marketplace such as phones equipped with advanced sensors (such as accelerometers and compasses), powerful processors and data connections. It particularly focuses on automatic detection and sharing of activity in order to avoid the functional restrictions and privacy implications of location sharing and seeks to create a minimal interface which allows its users to interpret and make interferences about the activities of others (and ultimately their availability) in an unobtrusive manner. By making no functional changes to the behavior of the application based on context data, it minimizes the potential problems of erroneous classification – all judgment is left to the human caller.

CHAPTER 3 - Awareness

The work on awareness in this thesis was conducted in a side-project in the early stages of the work. It explored the fact that as mobile communication devices become more common the potential to be interrupted by them has grown exponentially. This disruption is a serious, and not to mention every day, issue for many users. In order to avoid this disruption this thesis implemented a system which is capable of capturing, inferring and sharing (in the context of the wContact system described in later chapter) a range of simple but expressive activities through sensors built into a smart phone

3.1 User Research

Although there is considerable literature on human communication practices and needs, this is a dynamic and rapidly changing domain. Consequently, we conducted a lightweight two-stage exploratory user study to better document and understand the severity of interruptibility issues with mobile phones and the feasibility and acceptability of an automated status sharing as a solution. The initial phase of the study was an online survey; it was followed by a series of semi-structured interviews exploring more deeply some of the issues .

3.1.1 Online Survey

The survey, appendix H, was divided in three parts, respectively capturing demographics, mobile phone usage and opinions and usage of status sharing applications (up to and including micro-blogging services such as Twitter). It was available 13th-19th November 2009 and 112 responses were captured. However, questions were not compulsory, so results are provided by the number of responses made in each case. 64% of respondents were male and the average age was 29.4 (range 16-61, SD 0.76). 35% were students, 3% unemployed and the remaining working in a wide range of skilled and non-skilled professions. 100% indicated they owned a mobile phone and 72% that they either made or received calls on a daily basis, while the remaining engaged in 1-2 calls a week. Friends and family were the most likely communicators and reported inconvenient moments to receive a call were diverse. These included whilst engaged in some form of work activity (e.g. meeting, presentation: 31%), during other general activities (e.g. travelling, eating: 24%), in education (class, exam: 22%), socializing (with friends, dating: 16%) and resting (8%). All respondents indicated that interrupting people's activities with a call (and receiving an awkward and unhappy response) was a situation they would preferentially avoid.

Sharing status was used much more infrequently than calls with 41% indicating they rarely or never share status on a PC and 74% reported the same for mobile devices. The remaining reported infrequent to regular periodicity of status updates via tools such as instant messaging clients and social networks services. When introduced to the idea of a service that automatically detects activity and shares it as status on a mobile address book, 82% indicated it would be between useful and very useful. However, only 59% indicated they would use such as a service either to inform others about their availability, to avoid unwanted calls, or simply to share. The remaining 41% indicated they wanted to remain in control of such information, that they were uncomfortable sharing, or felt the service would not be useful. When queried about which activities would be useful to share, a wide range of suggestions were made, with few clear trends other than the activities of driving and sleeping.

3.1.2 Live Interview

In order to better explore this issue, 17 short semi-structured interviews were conducted. Ten participants were male and seven female with ages between 25 and 60. They were asked to describe a recent time when a phone call had interrupted their activity, whether they had cancelled a call due to being engaged in another task and what situations they felt it would be appropriate to make available to others so they could judge whether or not to make a call. The results were more extreme than those of the questionnaire. Nine interviewees indicated that they had been interrupted by a call that day, often suggesting that this was a daily occurrence. Particular activities identified were resting, engaging in a leisure activity and socializing; all these were also identified as activities to share, along with additional tasks of travelling, playing sports and bathing. All of them also recognized that they were the source of unwanted calls and that this was a situation they would like to avoid.

In summary, this research highlighted the ubiquity of communication in the modern world and that interruptibility remains a problem for a wide range of everyday users. Users are also supportive of the idea that automatic activity detection and sharing could address aspects of this problem and are cautiously positive about using such functionality. It also suggested candidate activities which should be detected.

3.2 Sensors

The goal of the automatic status detection is to provide users with up to date information relating to other's activities so they can make appropriate judgments about whether or not to initiate a call. The system makes no restriction on the capabilities of the users to make calls and the goal of the information is to help users make inferences rather than impose rules, or alter functionalities. The status detection is enabled through the use of sensors on the mobile device. Two are currently implemented: the accelerometer, and the microphone. Respectively, these can provide notions of the physical activity a user is engaged in (stationary/mobile) and the kind of environment they are situated in (quiet/loud). The application can also detect the battery level.

Deriving initial categories from the user study, the system used these inputs to make a number of classifications and represent these as status text. These included forms of physical activity (*active*, *walking*, *running*, *driving*) inferred from the accelerometer data and environment conditions (*tranquil*, *conversation*, *loud*) from the microphone data. Status detection was disabled when the battery was low (and the *low-battery* status set), but otherwise, activity classifications from accelerometer data took precedence over those from microphone data. Activity detection took place with the periodicity specified in the settings page, with a default of once every five minutes. Figure 4 depicts the entire system structure, including the priority attached to each sensor and recognition process and the framework in which this information is disseminated between clients and the server.

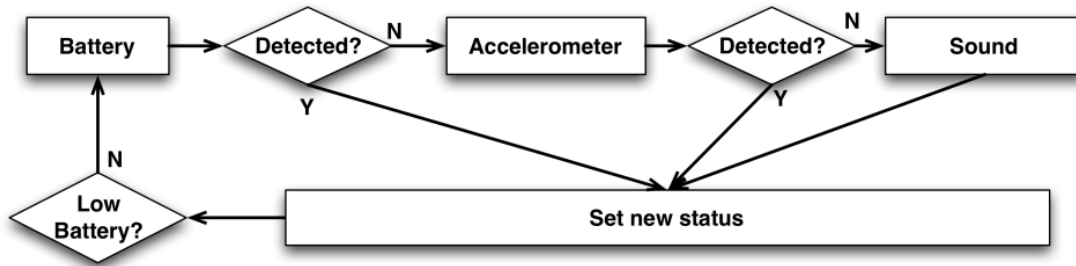


Figure 4 - Activity recognizer structure

3.3 Activity Logger

In order to develop recognition algorithms and verify performance, an application called Activity Logger was designed and implemented. It enabled data logging of the accelerometer sensor of an Android smartphone. The goal of this application was to distribute it to users so they could capture various activities throughout the day. This material could then be studied and analyzed.

This application had a simple user interface that allowed the identification of the user and their activity to be recorded. Once the recording starts a background service collects data and stores it in a file. The time the accelerometer sensor started is displayed in the screen allowing the user to know how long the application has been saving data.

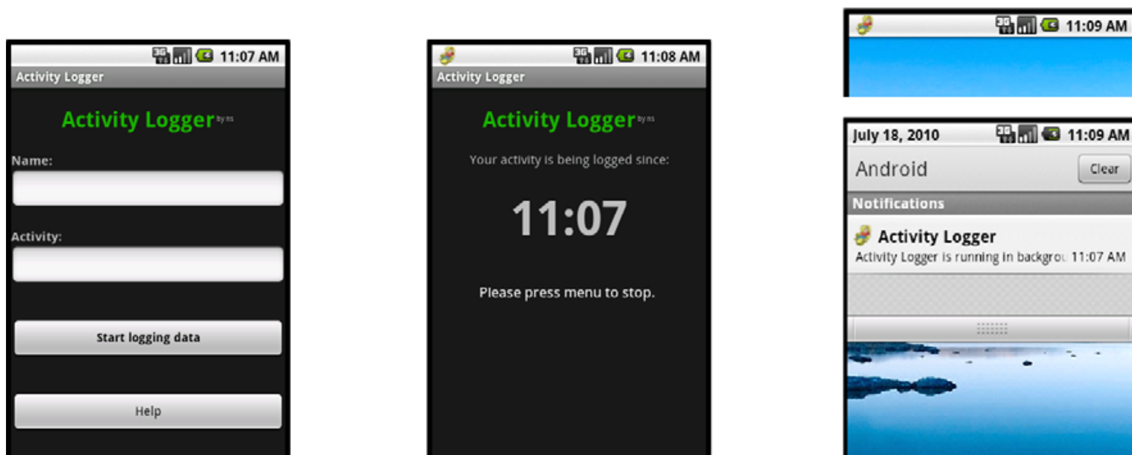


Figure 5 - Activity Logger Screenshots

As the service runs in the background, the user can use the phone without any limitation or interference, and whenever he or she wants to return to the application they may do so through the notifications bar. Figure 5 presents some screen shots of the application.

To make this application accessible it was distributed on a web page: http://muses.selfip.com/ActivityLogger. This provided the following purposes:

- Provide the application download

- Provide a quick way for users to publish the information received by their devices.
- Display information about the activities received

Figure 6 - Activity Logger web site screenshot

This set of information was available for download on another website that supported selection of a small data set for analysis. In this site, http://muses.selfip.com/dataset, important information about the collected data, like duration and frequency, could be set as shown in figure 6. When the create button was selected, an archive of the specified dataset was automatically generated and downloaded.

Choose from the list below to create data set with following settings:

Time: (sec) **N. datasets** Create

File name	Size	N. samples	Time (sec)	Frequency (1/sec)
drive.csv	1808873	53911	1094	49
drive1.csv	1851980	51099	1046	49
helena-bus1.csv	3247544	88808	2943	30
helena-walk.csv	2272943	62354	1251	50
helena-walk1.csv	833826	23454	474	49

Figure 7 - Dataset web site screenshot

However, despite significant effort to develop a substantial body of data for analysis, this objective was unsuccessful. Some users installed this application, but data was never uploaded. This aspect of the project was not successful and (ultimately) data recognizers were built using data derived from the thesis author and his immediate family and friends.

3.4 Analyzing Data

3.4.1 Accelerometer

The accelerometer classifier used 20 second blocks of data sampled at 50 Hz. Tests indicated that capturing this data at the default periodicity of once every five minutes had a negligible effect (less than 5%) on overall battery life. Before recognition, each block of data was normalized: data from each stream was squared, these figures summed and the square root taken of this total. *Walking* and *running* were detected through a simple peak counting

algorithm keyed to detect peaks within specific magnitude ranges and occurring at specific frequencies (0.5-1.2 paces/second for walking and 1.0-2.5 paces/second for running). This approach is similar to that used by Murray-Smith *et al*. [23]. Classifications of *active* took place when there were large data peaks, but a failure to fit the walking or running profiles. *Driving* was detected through looking for repetitive, high frequency low-magnitude oscillations through peak counting and examining the mean and standard deviation.

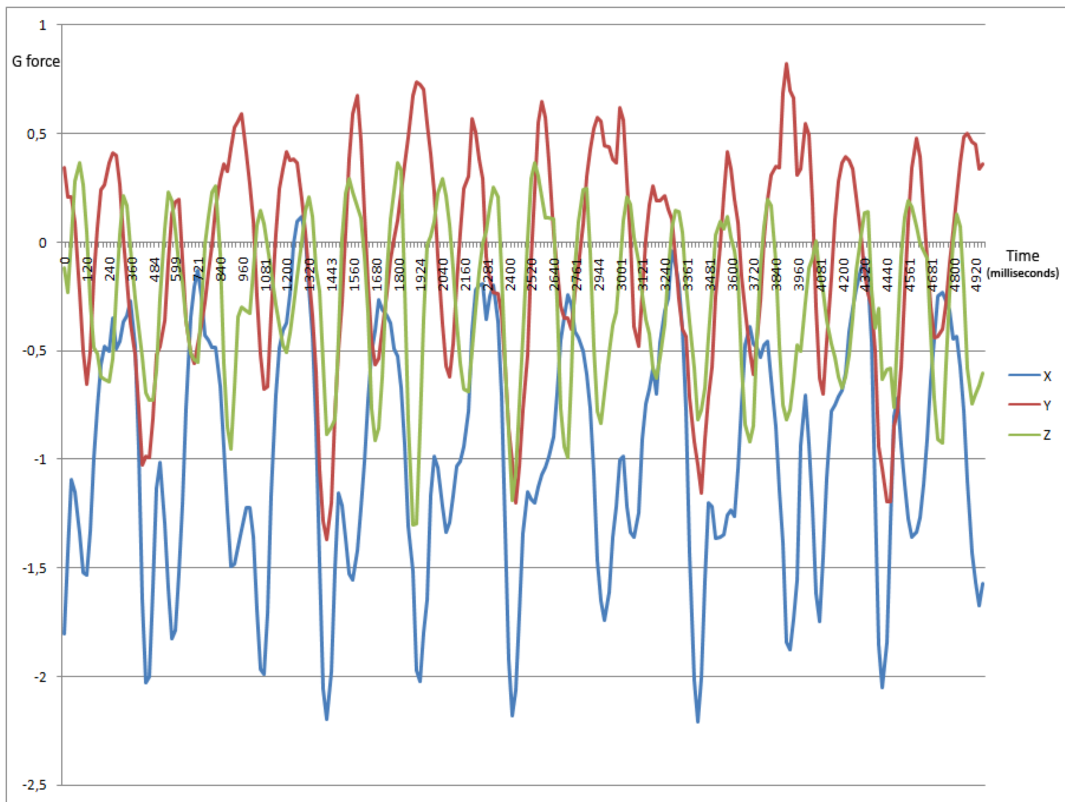


Figure 8- Data collected while walking

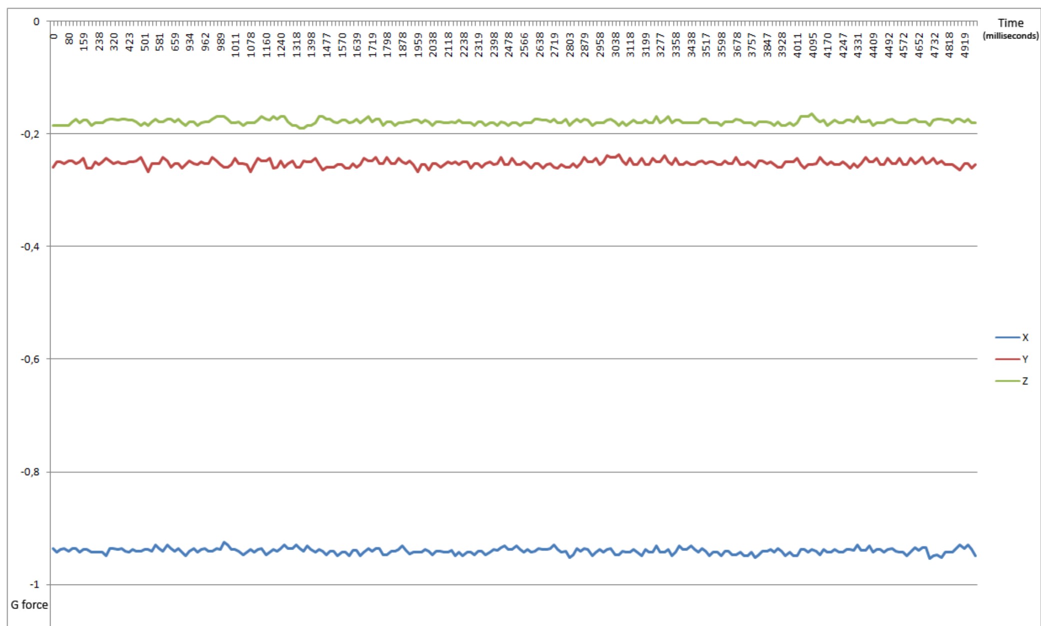


Figure 9 - Data collected while driving

3.4.2 Microphone

Audio levels were captured through the Media Recorder Android API, which is capable of return mean sound amplitude over particular periods of time. As with the accelerometer data, a window of 20 seconds was used to perform recognition. Subjective experimentation in a range of environments led to the determination of bands of mean amplitude corresponding to the categories of *quietness*, *conversation* and *loudness*. Although preliminary tests have shown this method to be effective, caveats to this approach include the fact the bands will be highly dependent on environmental conditions. For example, conversation is likely to appear quieter to the phone if it is situated in a pocket or bag. Further formal testing of the performance and usefulness of this recognizer are currently required.

3.5 Awareness conclusions

The system focuses on activities, rather than location, which has received more attention in the literature [e.g. 6]. The benefits of focusing on activities include reduced privacy implications (compared to location sharing) and directly appealing to users' ability to infer conclusions based on their own contextual knowledge. For example, at 2am, seeing that one friend is in a quiet environment, whilst another is in loud environment is probably sufficient to make useful inferences about their likely activities: one is probably sleeping, while the other is awake and likely to be socializing.

Further tests are required of the system recognition performance, preferably with data generated using the developed Activity Logger. Furthermore, the system should include additional sensors such as digital compasses that may be a robust mechanism to predict valuable data such as mode of transportation. However, although only partly successful, this thread of work suggests that awareness can maintain the benefits of modern communication technologies while addressing one of their key flaws: their ability to interrupt and disrupt other aspects of everyday life. This functionality was ultimately integrated into the wContact application described in the following chapters.

CHAPTER 4 - System Design

This chapter presents the methods of Usage Centered Design used to create the wContact application as well as design its architecture. The application improves upon current mobile phone address books and their basic operations. The main problem with address books lies in the fact that contacts are asynchronous, once they are inserted we cannot be certain that the data is reliable. Another major improvement is to supply awareness information trough a status system, so that when users communicate with they can do so with increased confidence that the time and channel is appropriate.

The contact management will be similar to a virtual community based on invitations and contacts shared with differing privacy levels. The synchronization will be managed through the cloud and mobile phone using a notify manager that responds in real time to changes made instead of waiting for client requests.

Communication between phone and the cloud will be using certificate management so as to provide security and avoiding abuse of login authentication. In order to complete the design modeling of the application the following methods were used:

- Activity Map
- Activity Profiles
- Participation Map
- Role Profile
- Use Case Model
- Task Cases
- Quality Attribute Workshop
- Attribute Driven Design

4.1 Activity Map

An activity map identifies relevant activities and their interrelationships pertinent to the design of solution to a problem. The most important activities are those which include interaction with the reference system, but this approach can also model other activities.

Activities define the context of use, the way that individual tasks are combined into more complex and interdependent tasks. After a short brainstorming session the identified important activities are: Automatic detection; User updates; Certificate Manager; Privacy manager; Storage and Notification Manager. They are represented in figure 8.

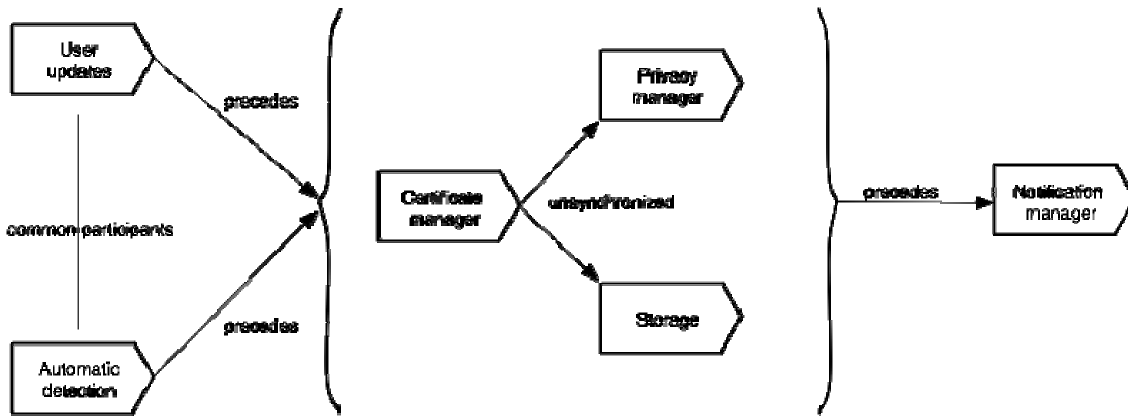


Figure 10 - Activity Map

4.2 Activity Profiles

This chapter will describe the activities presented in the activity map in figure 8. Each will be described in terms of its purpose, place and time, participation and performance.

4.2.1 User Updates

Purpose: User modifies profile, adds contacts, changes privacy or sets a manual status.

Place and Time: Can be done at any place or time, but the application needs to have an internet connection.

Participation: User interacts with the mobile device through available interfaces (touch, keyboard, pointer, etc).

Performance: Information is stored on the mobile device database and synchronization starts through certification exchange as soon as internet connection is available.

4.2.2 Automatic Detection

Purpose: Detects user activities automatically providing awareness information to all contacts.

Place and Time: User can set a periodic interval for activity recognition.

Participation: Activities are recognized through mobile device sensors (e.g. accelerometer, sound) and even from phone properties (e.g. battery level).

Performance: Sensors are checked in a particular sequence. If low battery is detected, automatic detection is deactivated.

4.2.3 Certificate Manager

Purpose: Create, verify and managed the exchange of certificates. The use of certificates avoids constant authentication and adds security to the transactions.

Place and Time: Every request starts with a certificate exchange.

Participation: Certificates are received through http requests to the cloud.

Performance: If the certificate is validated, then the requested operation is provided.

4.2.4 Privacy Manager

Purpose: Grant that data is visible only by users with appropriate permissions.

Place and Time: Requested every time that contact information is requested.

Participation: Levels of privacy are stored in a database and then compared for each transfer.

Performance: After those validations, data requests are pooled in a queue

4.2.5 Storage

Purpose: Data is stored on the cloud using Google Datacenter. Also, mobile devices use local database to store information providing offline access.

Place and Time: Every request, notification or settings update needs to be stored.

Participation: Google datacenter on the cloud and SQL on the mobile device.

Performance: After the data storage, the information will need to be dispatched through the notification manager.

4.2.6 Notification Manager

Purpose: Maintain all system synchronized between changes.

Place and Time: Always after a change occurs, whether automatic or manual.

Participation: These activities will use a pool to dispatch each notification.

Performance: Notifications need validation, and must be retried in case of error.

4.3 Participation Map

A participation map is a way to model an activity through a simple map of the playing field, a representation of the participants and their relationships with each other. Figure 9 represents the participation map of the application.

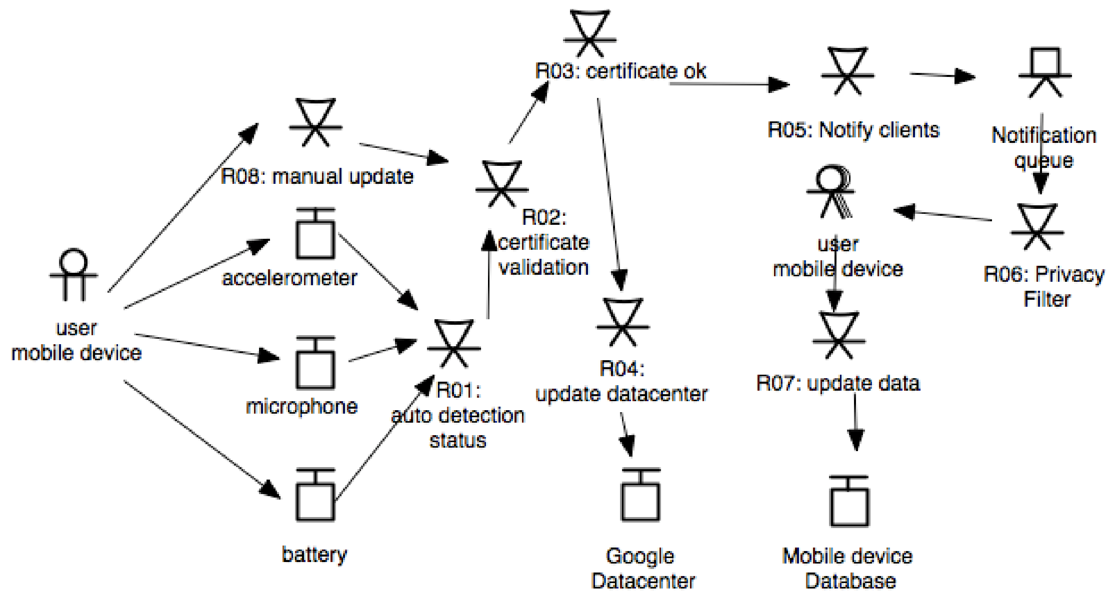


Figure 11 - Participation Map

From this map, it's clear that the role R03 has a major importance in all sequences as it is the first link between client and server. This makes intuitive sense since the application operates through requests to the server and only after successful validation can desired requests be completed.

4.4 Role Profiles

4.4.1 R01 Auto detection status

Context: Using sensors (e.g.: accelerometer, microphone) and phone characteristics to capture user activities in order to set a status.

Characteristics: Batch process independent of user interaction, it detects activities at a preset time interval.

Criteria: Attention to battery consumption.

4.4.2 R02 Certificate validation

Context: Receives a certificate from the last transaction identifying if is a secure and correct client.

Characteristics: Certificate replaces login authentication.

Criteria: Security issues must be tolerable to failures but secure.

4.4.3 R03 Certificate ok

Context: Match certificates prior to executing requests.

Characteristics: Trivial match process.

Criteria: Needs a valid certificate before executing request.

4.4.4 R04 Update datacenter

Context: Stores information received from client in the cloud.

Characteristics: Information will be stored in Google Datacenter.

Criteria: Information is stored and if required a notification is dispatched.

4.4.5 R05 Notify clients

Context: The notify process will be based on a queue. All clients are notified as soon as changes occur in the cloud, maintaining (in real time) all information up to date.

Characteristics: The queue will be built on Google Datastore. The notify process will use XMPP (Extensible Messaging and Presence Protocol) which is an open technology for real-time communication.

Criteria: The queue must be tolerable to errors, and to manage retries. The Jabber ID use in XMPP communication is a part of the certificate exchange.

4.4.6 R06 Privacy filter

Context: Ensure that information is provided to contact who have permission to view it.

Characteristics: Ensure privacy will be checked before notifying clients.

Criteria: Information must be inside privacy level.

4.4.7 R07 Update data

Context: Update information in mobile phone database when notifications are received.

Characteristics: Using SQL database available on android system.

Criteria: Notification may cause new request on the client to get more specific information.

4.4.8 R08 Manual update

Context: Whenever user makes a change on the client. Changes will be communicated to server using a specific request containing a certificate and also saved in database.

Characteristics: **U**pdates are made through the mobile device interaction (e.g. pointer, touch, keyboard, etc).

Criteria: The process must be similar to the processes used in current mobile devices.

4.5 Use Case Model

The use case model presents a graphical overview of the functionality provided by a system in terms of actors, their goals, and any dependencies between individual use cases.

This analysis focused on the analysis of the system as an actor without covering in depth the users as actors. The study of users' actions would be valuable, but due to time limitations, this study focused mainly on system behavior.

The use case model is represented in following figures:

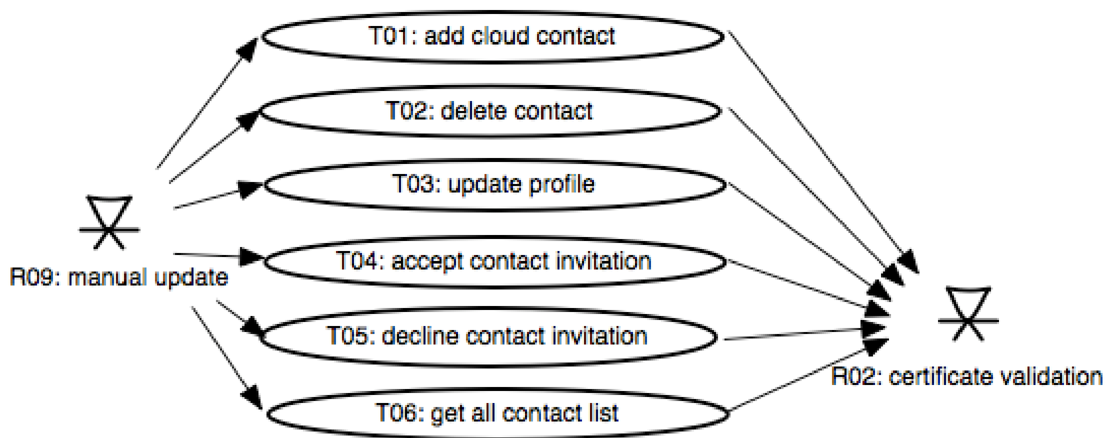


Figure 12 - Manual update role task cases

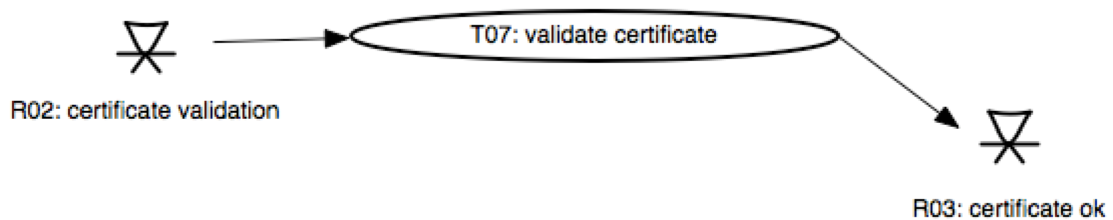


Figure 13 - Certificate validation role task cases

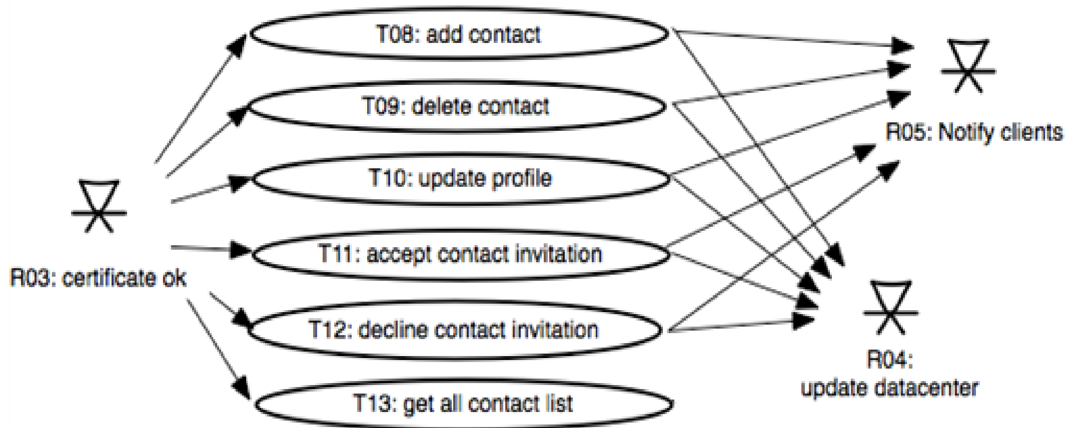


Figure 14 - Certificate ok role task cases

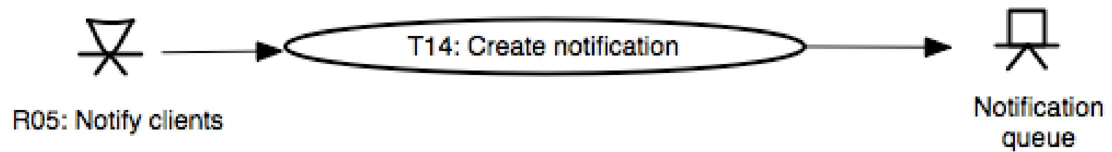


Figure 15 - Create notification role task cases

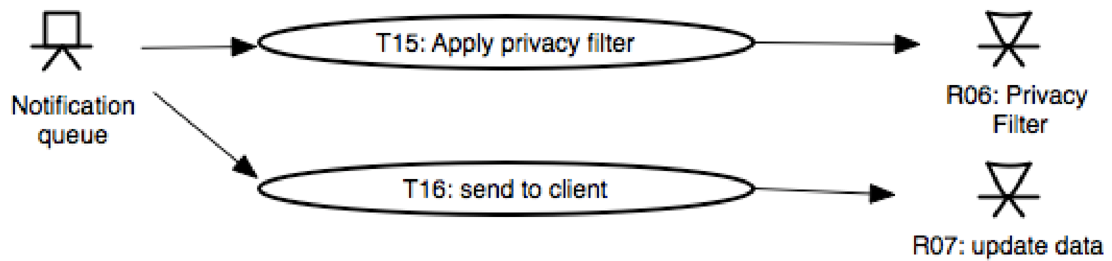


Figure 16 - Notification queue role task cases

4.6 Task Cases

The following tables represent in detail the task cases listed above by the use case model diagrams.

T01: Add Contact	
User intentions	**System responsibilities:**
1. User tap to show menu add contact	
	2. Check if device has internet connection
3. User insert contact nickname	
4. User set privacy level	
	4. Build request with certificate
	5. Send request to server

Table 2 - Add contact task case

T02: Delete Contact	
User intentions	**System responsibilities:**
1. User tap to show contact options	
	2. Check if device has internet connection
3. User select delete options	
	4. Build Request with certificate
	5. Send request to server
	6. Store changes in device database
7. Contact list is updated	

Table 3 - Delete contact task case

T03: Update profile	
User intentions	**System responsibilities:**
1. User tap to show edit menu	
	2. Check if device has internet connection
3. User updates profile	
	4. Build request with certificate
	5. Send request to server
	6. Store changes in device's database
7. User profile is updated	

Table 4 - Update profile task case

T04: Accept contact invitation	
User intentions	**System responsibilities:**
1. User tap in notification invitation	
	2. Check if device has internet connection
	3. Show invitation form
4. User accepts invitation	
	5. Build request with certificate
	6. Send request to server
	7. Receive contact from server
8. Show contact information	

Table 5 - Accept contact invitation task case

T05: Decline contact invitation	
User intentions	**System responsibilities:**
1. user tap in notification invitation	
	2. Check if device has internet connection
	3. Show invitation form
4. User declines invitation	
	5. Build request with certificate
	6. Send request to server

Table 6 - Decline contact invitation task case

T06: Get all contacts list	
User intentions	**System responsibilities:**
1. Client authenticates	
	2. Check if device has internet connection
	3. Send request to server

Table 7 - Get all contact list task case

T07: Validate Certificate	
User intentions	**System responsibilities:**
1. Certificate validation is sent	
	2. System decrypt certificate
	3. System validate certificate content
	4. System authenticate certificate
5. certificate is approved/declined	

Table 8 - Validate certificate task case

T08: Add Contact	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Create pending invitation
	3. Add to datastore

Table 9 - Add contact task case

T09: Delete Contact	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Delete contact in datastore
	3. Create notification
	4. store notification in datastore

Table 10 - Delete contact task case

T10: Update profile	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Update contact in datastore
	3. Get contact list
	5. Create notification for each contact
	5. Store notifications in database

Table 11 - Update profile task case

T11: Accept contact invitation	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Update invitation in datastore
	3. Get contact information
	4. Send contact information to client

Table 12 - Accept contact invitation task case

T12: Decline contact invitation	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Remove invitation in datastore

Table 13 - Decline contact invitation task case

T13: Get all contact list	
User intentions	**System responsibilities:**
1. Certificate is approved	
	2. Get contact list
	3. send contact list to client

Table 14 - Get all contact list task case

T14: Create notification	
User intentions	**System responsibilities:**
1. Operation done successful	
	2. Get contacts to be notified
	3. Insert new notification in database

Table 15 - Create notification task case

T15: Apply privacy filter	
User intentions	System responsibilities:
1. Operation done successful	
	2. Get contact list
	3. Match privacy level

Table 16 - Apply privacy filter task case

T16: Send Notification	
User intentions	System responsibilities:
	1. Get notifications from datastore
	2. For each get associated JID
	3. Send notification trough XMPP

Table 17 - Send notification task case

This process was developed to identify solutions for the subsequent implementation of the wContact application. Not all tasks were studied in great depth, but the whole system process was decomposed, since the requests to the notifications were well defined in order to advance to the design of the architecture.

4.7 Quality Attribute Workshop

The Quality Attribute Workshop is one way to discover, document and prioritize quality attributes early in a development life cycle. It helps to get maximum possible information about the quality when defining the structure of the system.

4.7.1 Non-functional requirements

R1: System design should be in all aspects similar to current contact management applications, such the application will support an easy and transparent use.

R2: The system should have a high security level, but avoiding user authentications whenever the user needs to execute operations

R3: Notifications, in case of failure, will be repeated at one minute intervals avoiding overload of the system

R4: The system must be always available to access information, but must have an internet connection to perform update operations.

4.7.2 Quality Attributes

A quality attribute is any property of the system which has nothing to do with functionality and they are important while defining system properties, system working conditions and as a driver for architecture design. Through non-functional requirements, the following attributes were chosen:

- Usability
- Security
- Availability

4.7.3 Scenarios

After defining the quality attributes, a set of scenarios that may affect the system are presented in the following tables. For each scenario we describe its stimulus, as its source, the environment, the artifact, the response and the respective response measurement.

Quality Attribute		Usability	
Attribute Concern		Provide easy and transparent usage for a new user	
Scenario	#1	User must be able to make all operations without any problems	
		Stimulus	First use of application
		Stimulus source	User needs to see a contact detail
		Environment	Operation
		Artifact	Mobile device
		Response	User can do it without help at first try
		Response mesure	Correct

Table 18 - Quality attribute scenario #1

Quality Attribute		Security	
Attribute Concern		Communication between client and the cloud	
Scenario	#2	User updates information on the cell phone and system automatic builds a request to the cloud	
		Stimulus	Update info on the cloud
		Stimulus source	User updates information
		Environment	Operation
		Artifact	Mobile device / Internet connection
		Response	Operation authenticates mobile device
		Response mesure	Correct

Table 19 - Quality attribute scenario #2

Quality Attribute		Security	
Attribute Concern		Communication between client and the cloud	
Scenario	#3	Request uses a invalid certificate	
		Stimulus	Update info on the cloud
		Stimulus source	User updates information
		Environment	Operation
		Artifact	Mobile device / Internet connection
		Response	Cloud replies wrong certificate
		Response mesure	Correct

Table 20 - Quality attribute scenario #3

Quality Attribute		Security	
Attribute Concern		Communication between client and the cloud	
Scenario	#4	Request uses a valid but old certificate	
		Stimulus	Update info on the cloud
		Stimulus source	User updates information
		Environment	Design
		Artifact	Mobile device / Internet connection
		Response	System processes synchronization before replying to requested operation
		Response mesure	Correct

Table 21 - Quality attribute scenario #4

Quality Attribute		Availability	
Attribute Concern		User needs to access information in the application at any time at any place	
Scenario	#5	When the user needs some information, the system must deliver this info, even if there is not any internet connection available	
		Stimulus	Wish to see contact details
		Stimulus source	End user
		Environment	Operation
		Artifact	Mobile device
		Response	User can access information
		Response mesure	Correct

Table 22 - Quality attribute scenario #5

Quality Attribute		Availability	
Attribute Concern		User cannot update information if application is limited without internet connection	
Scenario	#6	User updates information on mobile device	
		Stimulus	Update user profile
		Stimulus source	End user
		Environment	Degraded (No internet connection)
		Artifact	Mobile device
		Response	Operation is not allowed due to no internet connection
		Response mesure	Correct

Table 23 - Quality attribute scenario #6

Quality Attribute		Availability	
Attribute Concern		Cloud need to notify client	
Scenario	**#7**	A change occurs in the cloud. This change was put in the notify queue and subsequently dispatched	
		Stimulus	Notify client
		Stimulus source	Cloud
		Environment	Degraded (No internet connection)
		Artifact	Mobile device
		Response	Cloud cannot reach client. Notification maintains and will be called again one minute later
		Response mesure	Correct

Table 24 - Quality attribute scenario #7

4.7.4 Architectural Tactics

The architectural tactics can result in design decisions that influence the response control for a particular quality attribute. After defining the requirements and finding the quality attributes, architectural tactics can be defined for each of these attributes.

4.7.4.1 Security

The quality attribute of security will be assured using the **Resisting Attacks** and **Authenticate users** tactics. Authentication ensures that a user or remote computer is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications can provide authentication.

4.7.4.2 Availability

In order to garantee the quality of this attribute **prevention** through **transactions** was used. A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.

4.7.5 Attribute Driven Design

Based on the selected architectural tactics we must implement a component responsible for authentication and certificate management. This component was called CertificateManager. Security has to be maintained when transmitting information between clients. For this reason a privacy level is associated for each user and detail. A component called PrivacyManager was created in order to assure the proper functioning of sharing data between users.

Availability is maintained through the mirrored data between server and client allowing users to access data even when disconnected from the Internet. Transactions are taken into account in data management from the server side where most of the critical situations are, as multiple clients can access to the same information at the same time.

No usability tactics were chosen. In this case, attention was directed to existing applications to try to ensure that this implementation, design and usability does not deviate from them. By leveraging existing metaphors we expect to avoid creating a major challenge to users.

4.8 Architecture

Application requests are based on a simple client/server application, while notifications are based on publish/subscriber architecture. Client/server communication uses HTTP requests regarding RESTful design principles while publisher/subscriber implements a XMPP communication.

Figure 15 represents the architecture designed to support the application implementation.

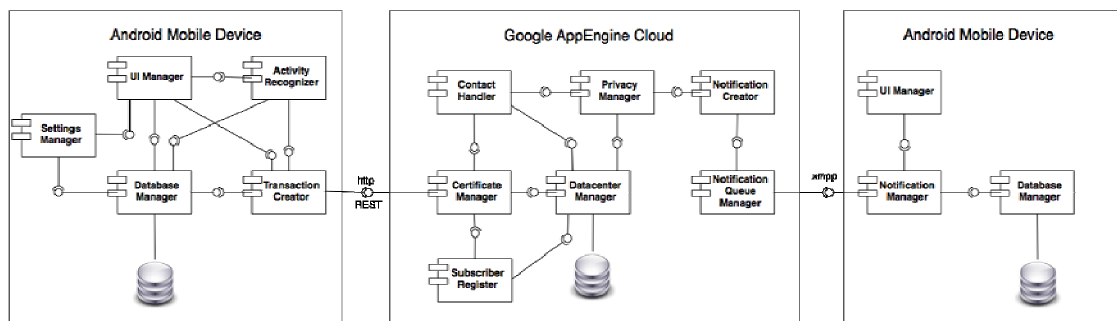


Figure 17 - System architecture

This architecture represents the way that modules are arranged throughout the system, either on the server or the client. The client was divided into two sections so as to better identify their responsibilities in the different processes, both in making server requests and receiving server notifications.

The client application consists of one module *UI Manager* which represents the user interaction. This will be responsible for storing and querying data through the *DatabaseManager* module and the communication to server module trough the *Transaction Creator.* Detection of awareness is represented by the *Activity Recognizer* module. This has the responsibility to update the user interface and to inform the server to store information in its database. The Module *Settings Manager* is an isolated module that does not transmit any information to the server and stores the user system's settings on the device.

On the server side, all requests are verified by the *Certificate Manager* module which delivers information to the *Datacenter Manager* as well as enables the completion of various operations through the Contact Handler. The *Subscriber Register* stores client information about the connection used to deliver notifications. These notifications are created through *Notification Creator* and sent by the *Notification Queue Manager.* The ability of these modules to perform the delivery of the notifications with correct privacy is ensured by the *Privacy Manager* module.

The client receives notification from the module *Notification Manager* which in turn sends information to be stored through the *Database Manager* module, previously described. If user interaction information is needed this is sent to the *UI Manager* module.

This overall system architecture was implemented as described in the following chapter. The use of architectural tactics was necessary to ensure that all modules were well designed and interconnected, while ensuring that no aspect was left behind.

CHAPTER 5 - System Implementation

This chapter described the entire implementation of the application it makes a division between client application and server application. However, before starting this description, an introduction to the choices made to ensure the success of the application is presened.

5.1 Material and Methods

To implement this application various choices have been made. These choices affect how the server and client have been implemented and the way they communicate. This subchapter presents these choices.

5.1.2 Google App Engine

To ensure a scalable system, a cloud server platform was selcted. The choice of Google App Engine makes perfect sense as it has an extensive free usage. The description of the Google App Engine follows in the next paragraphs of this subchapter.

Google App Engine supports running web applications on Google's distributed cloud infrastructure. This helps ensure that software is easy to build, easy to maintain and easy to scale as traffic and data storage needs grow. With this system, application developers do not need to maintain servers. Instead they upload their applications and they become available on the domain appspot.com.

App Engine supports applications written in several languages including a Java runtime environment based on using standard Java technologies, including the JVM, Java servlets and the Java programming language or any other language using JVM-based interpreters or compilers (e.g. JavaScript or Ruby).

In App Engine there are no set-up costs – fees are charged by consumption. Resources, such as storage and bandwidth are measured by the gigabyte and a free starter package with 500 MB of storage and enough CPU and bandwidth to support an efficient app serving around 5 million page views a month is provided.

In summary, App Engine provides:

- dynamic web serving
- persistent storage with queries, sorting and transactions
- automatic scaling and load balancing
- APIs for authenticating users and sending emails using Google Accounts
- task queues for performing work outside the scope of a web request
- scheduled tasks for triggering events at specified times and regulars intervals

The uploaded applications run in a secure environment that provides limited access to the underlying operating system. These environments are called sandboxes. Their goal is to isolate the application in its own secure, reliable environment that is independent of the hardware, operating system and physical location of the web service.

Some disadvantages and limitations are created by the secure sandbox environment:

- An application can only access other computers on the internet through URL fetch and email services.
- Other computers can only connect to the application by making HTTP or HTTPS requests on standard ports.
- An application cannot write to the file system.
- An application can read files, but only files uploaded with the application code.
- The application must use App Engine datastore, memcache or other services for all data that persists between requests.
- Application code only runs in response to web requests, a queued task or a scheduled task, and must return response within 30 seconds in each case.
- A request handler cannot spawn a sub-process or execute code after the response has been sent.

These restrictions and limitations significantly constrain the structure and form of applications that can be developed using this system.

5.1.2.1 Datastore

App Engine provides a distributed data storage service that features a query engine and transactions, but differs from traditional relational databases. Data objects have a type and a set of properties and queries can retrieve objects of a given type filtered and sorted by the values of the properties.

The structure of data entities is provided by and enforced by application code, meaning that entities are "schema-less". The datastore is consistent and uses optimistic concurrency control in which, for example, an update of an entity occurs in a transaction that is retried a fixed a number of times if other processes are trying to update the same entity simultaneously. As in common databases, applications can execute multiple datastore operations in a single transaction which either all succeed or all fail, ensuring the integrity of the data.

5.1.2.2 Scheduled Tasks

An application can perform tasks outside of responding to web requests. These tasks can perform on a schedule that is configured, such as on a daily or hourly basis.

Scheduled tasks are also known as "cron jobs", handled by the Cron service which allows to configure regularly scheduled tasks that operate at defined times or regular intervals. These cron jobs are automatically triggered by the App Engine Cron Service and result in invoking an URL at that time. This request is subjected to the same limits and quotas as a normal HTTP request, including the request time limit.

5.1.3 Android OS

The choice to implement the client application has been based on mobility and on devices that contacts are indispensable: mobile phones. Thus the use of a device that utilizes Java as the programming language was relevant besides all the capacities that are described in the following paragraphs of this subchapter.

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.

Android provides the following features:

- An application framework enabling reuse and replacement of components.
- A Dalvik virtual machine optimized for mobile devices
- An integrated browser based on the open source WebKit engine
- Optimized graphics powered by a custom 3D graphics library
- SQLite for structured data storage
- Media support for common audio, video and still image formats
- GSM telephony
- Bluetooth, EDGE, 3G and WiFi
- Camera, GPS, compass and accelerometer
- Rich development environment including a device emulator

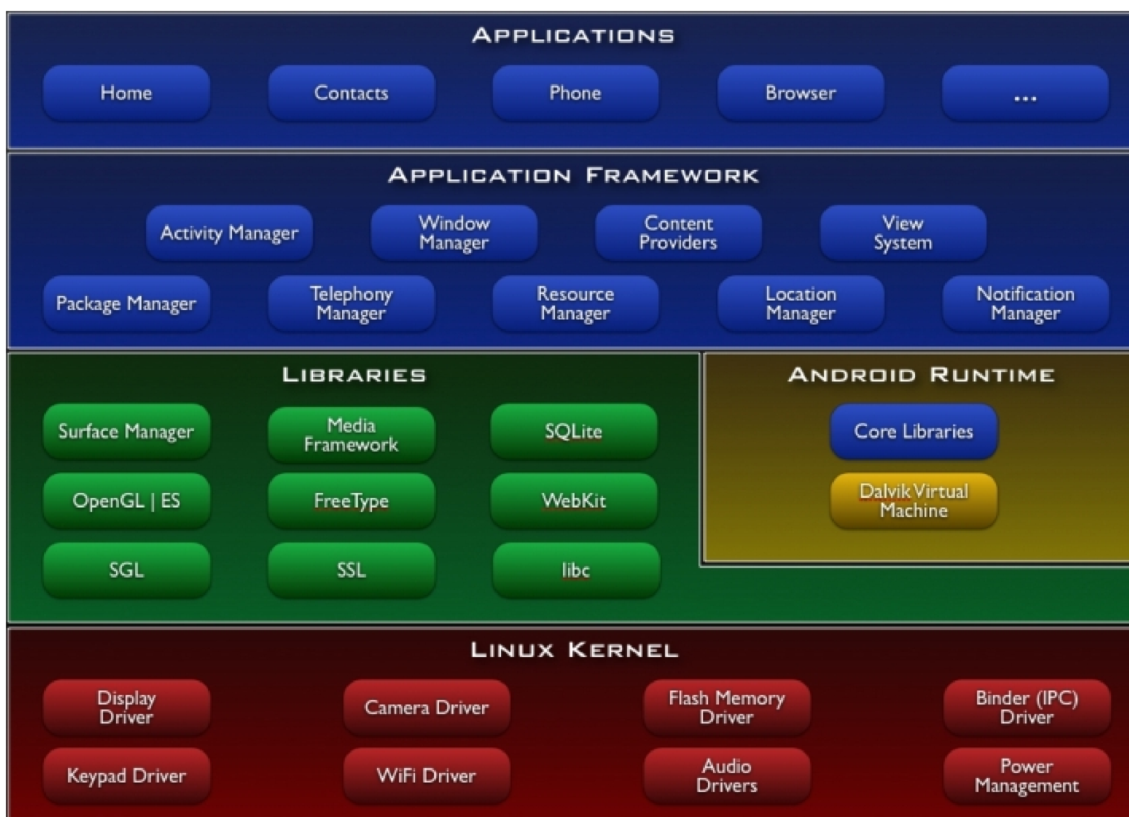


Figure 18 - Android OS

Android offers the ability to build new applications where we are free to take advantage of the device hardware, to access local information, run background services, set alarms, add notifications to the status bar and use other device specific functionality.

The application architecture is designed to simplify the reuse of components where any application can publish its capabilities and any other application may then make use of those capabilities.

Underlying all applications is a set of services and systems, including:

- A rich and extensible set of Views that can be used to build an application, including lists, grids, text boxes, buttons, and even an embeddable web browser
- Content Providers that enable applications to access data from other applications, or to share their own data. This is the only way to share data across applications as there isn't common storage area that all Android packages can access.
- A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files. Externalizing this resources from application code maintain those independently and ensure they are easy to manage.
- A notification Manager that enables all applications to display custom alerts in the status bar. This is a useful way to tell the user that something has happened in the background.

5.1.4 REST (Representational State Transfer)

For developing communication between client and server in the client/server architecture, RESTful techniques were chosen. REST is an architecture style and an analytical description of the existing web architecture. Its web service design has the following characteristics:

- Client –Server: a pull-based interaction style: consuming components pull representations
- Stateless: each request from client to server must contain all the information necessary to understand the request and cannot take advantage of any stored context on the server
- Cache: to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable
- Uniform interface: all resources are accessed with a generic interface (e.g. HTTP GET, POST, PUT, DELETE)
- Named resources> the system is comprised of resources which are named using a URL.
- Interconnected resource representation: the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another
- Layered components – intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

An alternative to REST would be SOAP (Simple Object Access protocol) which is a protocol for XML-based distributed computing that is designed to handle distributed computing environments it is the prevailing standard for web services but is conceptually more difficult as well as harder to develop for and requiring specialized tools.

Since RESTful is much simpler to develop then SOAP as it has a small learning curve and less reliance on tools and its design and philosophy is closer to the web, this has proven to be an excellent choice.

5.1.5 XMPP (Extensible Messaging and Presence Protocol)

For implementation of the communication between publisher/subscriber XMPP was the protocol chosen. The extensible Messaging and Presence Protocol is an open Extensible Markup Language (XML) protocol for near-real-time messaging, presence and request-response services. [29]

XMPP is not wedded to any specific network architecture, however it usually has been implemented via a client-server architecture wherein a client using XMPP accesses a server over a TCP connection, and where servers also communicate with each other over TCP connections.

An XMPP server acts as an intelligent abstraction layer for XMPP communications and its primary responsible are:

- To manage connections from, or sessions for, other entities in the form of XML streams, authorized clients, servers and other entities. Note that an XML stream is a container for the exchange of XML elements between any two entities over a network.
- To route appropriately-addressed XML stanzas among such entities over XML streams. An XML stanza is a discrete semantic unit of structured information that is sent from one entity to another over an XML stream existing at the direct child level of the root element.

Most XMPP-compliant servers also assume responsibility for the storage of a data that is used by clients (e.g. contact lists or presence applications). In this case, the XML data is processed directly by the server itself on behalf of the client.

In case of a XMPP client, most of them connect directly to a server over a TCP connection and use XMPP to take full advantage of the functionality provided by a server and any associated services. Multiple resources, such as devices, can connect simultaneously to a server on behalf of each authorized client, being differentiated by the resource identifier of an XMPP address as defined under Addressing Scheme.

Addressing Scheme defines that the address of an XMPP entity (a network endpoint that can communicate using XMPP) is termed a Jabber Identifier or JID and contains a set of ordered elements, specifically a domain identifier, node identifier, and resource identifier.

In sum, the use of XMPP protocol allows the application to perform its notification system perfectly. Furthermore XMPP is the only notification system compatible with Google App Engine.

5.1.6 Summary

For the server implementation it was chosen to implement and deploy to the Google cloud, App Engine, while the development of client application choice was made on a phone that uses the Android operating system. Combining these choices bring more value to application development as well as both being developed by the same company which allows implementation using same programming language: Java. Together these choices facilitated the overall development of the wContact application.

5.2. System Overview

This application has the purpose of keeping a lifetime organization of user contacts. It is based on a minimal social network where each user is responsible for maintenance of his or her own profile. This profile is shared through user invitations by a simple username.

Contacts as we know today are hard to keep in memory (eg. phone number, or email address) and easy to enter incorrectly where a single letter or number renders the contact completely useless. Using a username (i.e. a pseudonym) is easy to remember. It is the goal of this thesis to provide all necessary contact information following a preset privacy level. Users therefore, do not have any problem to share contact information, once the owner of the profile can choose an adequate privacy level.

Another important goal gives concerns the improvement of the quality communication: the reduction of the number of calls which interrupt other activities. We achieve this goal providing clues to give an idea of what the availability of a user to communicate at that exact moment. The simplest way is to give access to user provide a status quote representing a contact's availability. However this requires a high effort to user to keep it truthful through time. One transparent way of achieving this is using the sensors available in the phone to detect activities and environmental clues about user availability and share this automatically without user intervention.

5.2.1 Authentication / Registration

The application needs user authentication. If the user is registered it will retrieve all contact information and contact list from the server, otherwise a small form where user can choose a user name, password and set an email will be presented. The chosen username is all that the user needs to pass all contact information between other users.

After authentication is successfully achieved, the user can use the application without further authentication, which provides the sensation that application runs locally. However the application aggregates some information (phone unique id, session identifier, user id and also XMPP jabber id) to assure that the user is authenticated and that the information stored locally is always synchronized with the server.

5.2.2 Edit Profile

If user registers a username for first time, application will show the empty profile where user can set his contact information. A full list of possible contacts is shown like:

- Phone numbers
- Email addresses
- Chat addresses
- Postal addresses
- Web sites
- Virtual communities links
- Blogs
- Organization information
- Notes

Each one of these contact details is followed by a privacy level, and a label where user can choose one of the presets or name it in more appropriate way. As soon that user gets his list populated, notifications will be sent to every contact, following the privacy agreement, in order to keep his contact updated for everyone.

5.2.3 Privacy

This privacy functionality is a simple way to warranty that private information is shown only to friends with appropriate permission. It is based in a four levels (Partner, Family, Friend and All) and follows a priority order. The client can only see information that is marked as to all, while on the other side, partner can see all contact information. In a more specific example, family has access to family, friend and all information, while members of this group cannot access information set as to partner.

Each contact detail has an associated privacy level, and a user is added to a contact list is also assigned a privacy level. This arrangement means that this match can be simply done and contact information is successful shared.

5.2.4 Remove User

If a user no longer wants to keep track of a contact, they can simple remove it. This operation will not only remove the profile from your list, but also remove your contact from the contact's list and no notification is sent to the other user. This follows the simple concept: if you do not want to find someone, it is likely you do not want to be found by him or her.

Another simple way to reduce your information to a contact is changing their privacy in your contact list for a lower privacy level. Your information on his list will be automatically updated following the new security settings based on your profile details privacy. For example: if you have a phone number shared as to friends, and you do not trust in one of your contacts in list which has the privacy level of friends, simple change the privacy level to all, and this contact won't have access to your phone number anymore, but will keep in touch with any other details that you have set as share to all

5.2.5. Define Settings

Some settings can be defined in this application. Notifications are one of them. Some processes are default to trigger notifications to users such as:

- contact invitation
- user profile in list updated
- status changed

Notifications can be annoying, mainly if you have a long list of contacts. You can select which notifications do you want receive, and processes will still running once they all run in background.

Another important setting available relates to automatic static detection. You can deactivate automatic status detection, or increase interval of status detection or even choose which sensors to use in status detection. The application uses accelerometer and microphone

to detect user activities and environmental sounds, all these ones are disabled in case of low battery, and the user status becomes one which informs all contacts in list that the user is running out of battery. For more information about this system see chapter 3.2.

5.2.6. Change Status

Users can easily change their status to manual input by using a home screen widget designed for this purpose. When selected, it opens an application with all status's previously created by the user. Status is saved once the probability of repeating a status is very high, and the effort to write every time that a situation changes increases the effort for user interaction. When user changes to a new status, all contacts are notified about this change providing clues to others about his situation.

5.2.7 Remove Profile

User can remove his profile in two ways: remove information from the phone or remove completely the system account

5.2.7.1 Remove from his phone

The user's information is fully removed from the phone's database and maintained on the server. If the user logins again, all information will be received from server. Contact information remains available for all contacts in the list and no change is made on the server side.

5.2.7.2 Cancel account

When the user removes his or her account, it will be removed from the phone and also from the server. All contacts in list will no longer have access to the user's information, the contact details will be removed from all lists of his or her contacts.

5.2.8 Background Process

This application will have an always running process schedule to start as the phone starts up. This background process is responsible for retrieving information from the sensors respecting the user's settings for sensor detection. It will be responsible to set automatic status if this is enabled.

Also, this background process will be responsible to receive notifications from the server, updating the phone's database and sending notifications, once more, respecting the user's settings.

5.2.9 Summary

This chapter describes general approach to the application behavior. It covers the various operations of the system as well as the responsibilities of the user. A more in depth description of the implementation of the implementation follows.

5.3 Server Implementation

The entire server was implemented in Java using the Google App Engine SDK. The server is responsible for receiving requests from clients via HTTP requests, storing information through the datastore and notifying clients, when necessary, through the Cron and XMPP services.

5.3.1 Datastore

A small database was designed as found in figure 17.

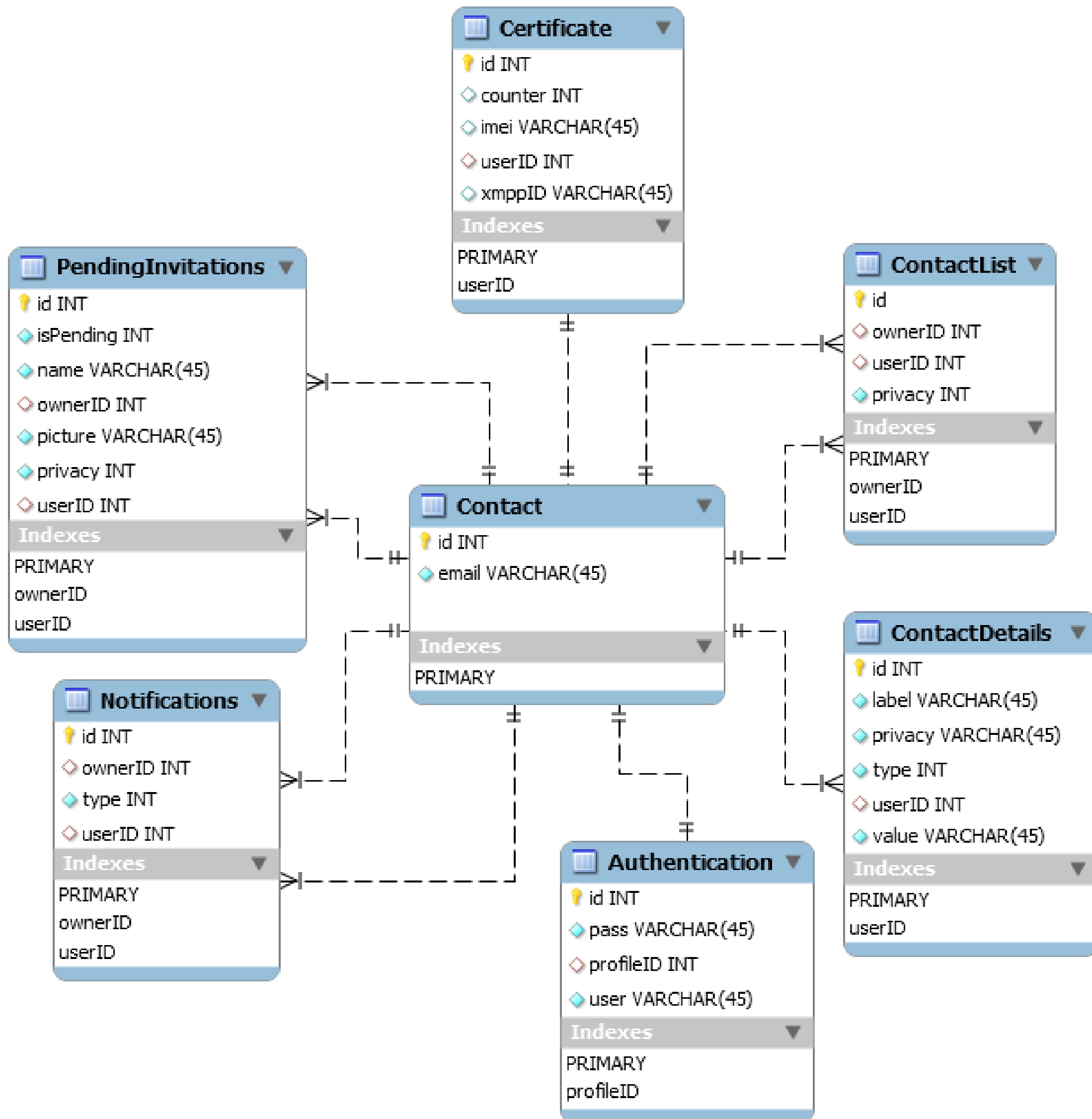


Figure 19 - Database diagram

The implementation of this database is on the Google App Engine datastore which supports the Java Persistence API, a simple programming model for entity persistence. This interface is provided by the DataNucleus Access Platform, an open source implementation of Java persistence standards, which supports the App Engine datastore. Appendix A shows the configuration file that indicates how the Google App Engine uses the datastore through the Access Platform. This file is named 'jdoconfig.xml'.

Requests to the datastore create instances of the PersistenceManager class by using an instance of PersistenceManagerFactory class. Once a Persistence ManagerFactory instance has initialized and the application uses this exclusively by storing it as a static variable. This enables it to use with multiple requests and multiple classes through the use of the singleton design pattern as is shown in figure 19.

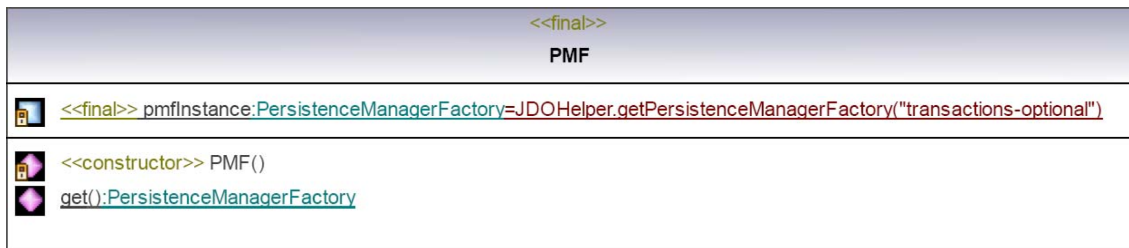


Figure 20 - PMF class

Persistence classes in Google App Engine do not allow extending classes or implementing interfaces, avoiding the use of the hierarchy and the structure without a solid definition. An optimal solution in the wContact application would have entailed building a middle class implementing a structure and using the persistence classes for accessing/storing data only. This amendment was not considered critical due to the relationship between time spent (large) and the resulting improvement (small).

The roles and rationale of the individual database tables are described in the following subsections.

5.3.1.1 Authentication

Authentication of a user in the application consists of a username and password. This information is stored in an authentication table and cross-referenced. Besides this information, each user has one (and only one) unique profile, so login references this profile, using a one-to-one relationship.

5.3.1.2 Certificate

Using certificates was the solution adopted in order to allow users to perform several operations without having to perform repeated authentications. To ensure system integrity, a certificate had to be constrained by several critical components including user identification, a unique device identifier and an incremental handler to ensure that the user making the request is the same individual. With this implementation the application can make an assessment of synchronization state of the device.

Another aspect that would be useful to put in the certificate would be the Jabber ID, the XMPP connection identifier. However, as the target devices are mobile, it was considered that they would lose or gain access to the Internet frequently, causing the JID to also change. One solution would be to make a request to server whenever it changes stating this has occurred. Another method, less bandwidth expensive but still functional, would involve embedding this identifier in the certificate. This was the select solution.

Therefore the certificate contains the following elements:

- User identifier
- Device identifier
- Incremental identifier
- Jabber Id

Since each user only has a unique identity that is associated with a single profile, it is appropriate that a profile has a one-to-one relationship with the certificate table as identified in figure 17.

5.3.1.3 Contact

This table was created to avoid linking the contact details directly to the authentication table, since a contact can change radically over time. However, for the purpose of this thesis this table is only responsible for creating an identifier for the profile.

5.3.1.4 Contact Details

This table is responsible for storing all the contact information. Every detail has a handle type that allows for the information to be differentiated. The mapping between codes and contact types used in this thesis are shown in table 25

Type	Description
0	Name
1	Phone numbers
2	Email addresses
3	Chat usernames
4	Postal addresses
5	Organizations details
6	Notes
7	Birthday
8	Picture
9	Status
10	Websites urls
11	Virtual Communities
12	Blogs urls

Table 25 - Contact details types

5.3.1.5 Contact List

The contact list is defined by a list of user profiles, which means an association between the user profile and all the profiles that have been added. Beside this association, a property named privacy qualifies the sharing relationship between the user and each profile.

All contact details have a label allowing its value to be associated with a context. Beyond this context, each detail is an assigned a privacy level. Although privacy was not deeply explored in this work, a basic discussion of its implications is included in section 5.3.5.

5.3.1.6 Notifications

To ensure synchronization between the server and client, when a change occurs on the server side a notification is created and stored in a database table named *notifications*. These notifications are composed of the identifier of the individual responsible for the changes (the

owner) and the identifier of the contact to be notified (the *user*). In addition, notifications have a type identifier whose meaning can be found in table 26:

Type	Description
0	User has accepted invitation
1	User profile as updated
2	Privacy changed

Table 26 - Notifications types

Once the notification is delivered to the client application, the notification is removed from the datastore.

5.3.1.7 Pending Invitations

The contact list is filled through invitations to other users requesting the sharing of their profiles. Once the user indicates his or her intent to add a contact to the list the PendingInvitations table becomes populated with the following data:

- Owner – the identifier of the requester
- User – the identifier of the addressee
- Name – requester name
- Picture – requester picture
- Privacy – privacy level to be assigned

Invitations are similar to notifications and use the same channel, but contain information about the applicant. This is because during a notification the user already has information about the applicant, while during an invitation, the user does not have this information and is requesting it.

Invitations are not removed immediately after delivery. Instead a flag is raised identifying the delivery status. When an invitation is delivered to another user, this flag is updated and only after the answer is received is the invitation removed from database.

5.3.2 Servlets

Requests from the client application comply with the RESTful architecture style. This implementation has security issues, mainly being vulnerable to man-in-the-middle attacks. One good solution to this problem when working with REST is SSL, the standard for secure transfer on the web SSL. Another solution is to require the hash of the body signed via a private key. These techniques were not explored in this thesis.

In order to handle all requests from clients a set of servlets were created and configure in 'web.xml' file available in Appendix A:

- AuthenticationServlet
- ContactServlet
- ContactListServlet
- NotifyServlet
- PendingInvitationServlet
- SyncServlet
- XMPPServlet

A UML class model can be found in Appendix C, and the table 27 summarizes all servlets responsibilities and operators:

Servlet name	GET	POST	PUT	DELETE	Represents
AuthenticationServlet	✔	✘	✔	✘	Sign in/Registration
ContactServlet	✔	✔	✘	✔	User profiles
ContactListServlet	✔	✔	✔	✔	List of contacts
NotifyServlet	✔	✘	✘	✘	Notifications process
PendingInvitationsServlet	✘	✘	✔	✔	Invitations process
SyncServlet	✘	✔	✘	✘	Synchronization process
XMPPServlet	✘	✔	✘	✘	Communication from client

Table 27 – Web services description

A more detailed overview of the servlets is presented in the following subsections.

5.3.2.1 Authentication Servlet

This servlet is responsible for two functions:

1. Authenticating a particular user via a certificate so that future operations can be performed without disruption.
2. Registration of a new user.

In the case of authentication, the servlet receives a GET request with two parameters, username and password, and matches those against those in the database. If a valid match is found, a certificate is created and sent to the client. Its functionality can be found in Appendix E.

When this servlet receives a PUT request, this means that it is receiving a request to register a new user. This request has three parameters: username, password and Google Account email. Note that Google Account password is never transmitted to server since the use of Google Accounts is only needed for client-side authentication and the XMPP service used to receive notifications on the mobile device.

The registration process verifies the non-preexistence of the username or email to ensure the uniqueness of users. If no problems occur, the creation of a new user is the result of this operation. A diagram of this sequence in Appendix E provides a better analysis of this operation.

5.3.2.2 Contact Servlet

The ContactServlet takes care of user profiles. It allows them to be examined, edited and even removed. Profile creation is handled solely by the registration process ensuring that each profile is associated to a single user and vice versa.

A request from the client to view a profile can has a number of parameters. These include the identifier of the profile and the privacy-level that client application requests. If these parameters do not exist, it means that the user is accessing his or her own profile so that the required identifier is present in the certificate and no privacy level is required.

DELETE requests to this servlet do not have any parameters, since deleting profiles can only be achieved by their owners and certificate already expresses user identifiers. In this way, profile integrity is guaranteed.

To update a profile this servlet receives all the profile details through a parameter existing in the UPDATE request which is transmitted in XML. This xml string is decomposed by filling out a list of details allowing them to update the datastore.

The full description of these operations can be found in the sequence diagrams in Appendix E.

5.3.2.3 Contact List Servlet

This servlet respond to all HTTP request types (as can be seen in table 27). These operations correspond to getting the contacts list, updating a contact in the list, inserting a new contact in the list and deleting a contact from the list.

Adding a contact is based on invitations. Thus when the servlet receives a PUT request it does not add a contact directly in to the contact list. Instead, it creates an invitation notification that will be sent to the contact to be added. Only after this approval, will the contact be added to the contact list.

DELETE requests will remove a named contact from the contact list, and as the lists are bidirectional, this change will be propagated to deleted contact's own list. This implements the concept that if you do not want to see someone, you certainly you do not want to be found by them. It also ensures that an individual user retains complete control over their contact details in the system – they can delete them from any user at any time simply by removing that user from their contact list.

Another operation that receives special attention is in incoming POST request. The only update available to the contacts in list is changing the level of privacy. Contacts updates are done through the profile update in which each user is responsible for each update.

Further information on requests to this servlet can be seen in Appendix E.

5.3.2.4 Notify Servlet

Notifications are dispatched one a minute by default. And as the Google App Engine server relies on http requests is not possible to set a timer or launch a thread to handle this kind of intermittent process. To attain this functionality a scheduled system called Cron is used. The Cron service allows developers to configure tasks to operate at well defined regular intervals.

The file 'cron.xml' creates and configures the Cron service for wContact. It is shown in Appendix A and produces GET requests to the NotifyServlet every minute. This servlet then reads the available notifications and propagates them to client applications using XMPP service available.

Notification and PendingInvitations are consulted and a Jabber ID is obtained from the certificate of each user. Notifications are simply dispatched while processing invitations

requires a check to the flag to known if it is an invitation that has been delivered or not. Invitations are dispatched only if still pending.

The sequence diagram of the entire process can be seen in appendix E.

5.3.2.5 Pending Invitations Servlet

This web service receives responses to invitations. Only two answers are possible, so this servlet only serves two types of request: PUT and DELETE. The first accepts while the second rejects an invitation.

Both of the requests will remove the invitation from the PendingInvitations table, but the accept process will add the user to the contact list of the owner and consequently add the owner to the user contacts list (since the lists are bidirectional).

5.3.2.6 Synchronization Servlet

When a synchronization failure is detected by the client application, it needs to get an updated certificate in order to perform operations. This existing certificate is only good for such an updating operation. This is achieved through a POST request include the old certificate skipping incremental identifier validation and replying the actual certificate existing in the certificate table.

The intent of this web service implementation was to verify alterations by consulting a history table on the client and the server in order to identify changes that needed to be made. However, due to time limitations for application implementation in this thesis, this feature was not implemented in full. The simple implementation used in this work entails updating all information on the client when an updated certificate is required.

5.3.2.7 XMPP Servlet

The client application can send chat messages to App Engine application. For server application receive chat messages, the XMPP message service must be enabled in the app's configuration file: 'appengine-web.xml' as is in Appendix A. With the XMPP service enabled, when the App Engine application receives a chat message it makes an HTTP POST request to the following path: '/_ah/xmpp/message/chat/' which is mapped to the XMPPServlet class shown in appendix A.

The client application sends a chat message to the server when it receives a notification from it with the objective of validating that the notification was properly delivered. Since there are two types of notifications: notifications and invitations, an identifier is added to handle the notification so that the system can properly distinguish which notification is being acknowledged.

5.3.3 Cloud Response

The CloudResponse class was created to manage responses to requests that were made to web services. Responses from web services are held in xml format containing two distinct parts: a header containing the updated certificate and the message body containing all the information requested by the client application.

In the case of the occurrence of an error on the server while processing a request, CloudResponse creates a fictitious certificate stating that an error occurred, and the message body will be the description of the error itself. This is composed of a code and a description.

More information about this class and process is available in appendix C and E.

5.3.4 Certificate Manager

The certificate manager has the ultimate responsibility of ensuring the authenticity and integrity of the certificates. The CertificateManager class receives request objects from client and gets the certificate from the header of http request. This data is then compared with certificates existing in the datastore.

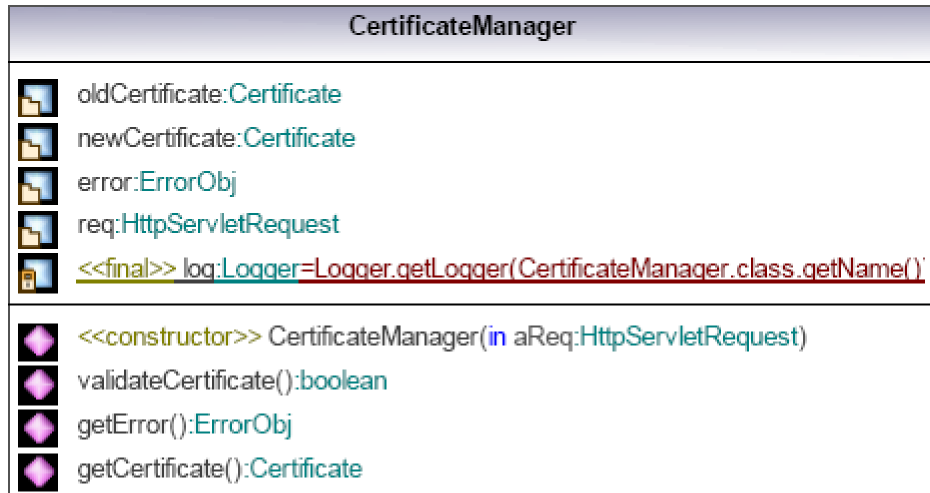


Figure 21 - CertificateManager class

Figure 19 represents the CertificateManager. Although two certificates exist in this class, it does not hold the client and server certificate. When the client sends the certificate that exists in its database and it also sends data to create the new certificate. The possible data to be changed include the device identifier and the Jabber ID from the XMPP connection.

After the certificate manager validates the old certificate (the certificate existing in client database) it fills in the new certificate with the user identifier and the incremental identifier. Thus, the header of CloudResponse is filled and certificate is updated in datastore.

5.3.5 Privacy Manager

Privacy is a complex subject and a nuanced examination of this topic is beyond the scope of this thesis. Instead, a simple and functional mechanism was implemented. Four different levels of privacy are supported: partners, family, friend and all. Each level was associated with a numerical value as shown in table 29.

Level	Description
0	Partner
1	Family
2	Friends
3	All

Table 28 - Privacy levels

In order to maintain simplicity, the system does not allow the selection of multiple levels of privacy. Instead, a hierarchy of privacy as shown in figure 20 was implemented, where partner can have access to all the information and the 'all' can only see what is the least sensitive information.

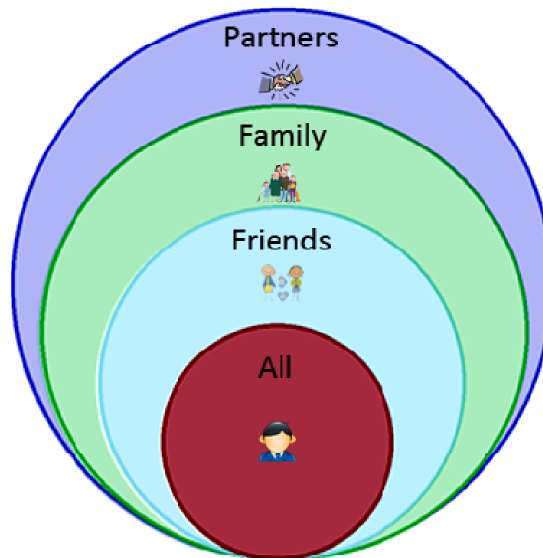


Figure 22 - Privacy representation

To implement the operation of privacy management in the sharing of the contact information a class named PrivacyManager was created. This class ensures that information is shared to those with access and limits access to those who without permission. This operation is simple and is based in a comparison between the privacy levels associated with each contact with the privacy level associated with each contact detail. If the associated privacy level in the contact details is less or equal to the privacy level associated in the contact list, the contact detail is deemed visible and sent to the client.

5.3.6. Summary

This chapter describes the server implementation. It was deployed to the Google App Engine servers and will handle HTTP requests from client application. Notifications, in turn, are carried out through a Cron service that runs every minute, being delivered to the application client through a XMPP channel using GTalk server.

5.4 Client Implementation

The wContact client application was implemented in Java using the Android SDK. This application has a user interface allowing easy access to information which is stored in a local database in device. This is synchronized with a server via HTTP requests and it is necessary that the phone has an internet connection for full functionality.

For automatic detection of awareness information, sensors on the device (accelerometer and microphone) are used. For receiving notifications from server a XMPP client is implemented using the Smack library. All this systems are explained in detail in the following subchapters.

5.4.1 Database

The implementation of the database to the client application was designed as a mirror of the database server designed as previously shown in figure 17. This time the database was implemented in SQLite, as provided by Android's framework.

In order to universalize access to the database and ensure data security, a hierarchy was created as shown in figure 21 and represented in detail in Appendix D. The DBAbstract class implements access to the database through transactions and full operation can be seen in Appendix F.

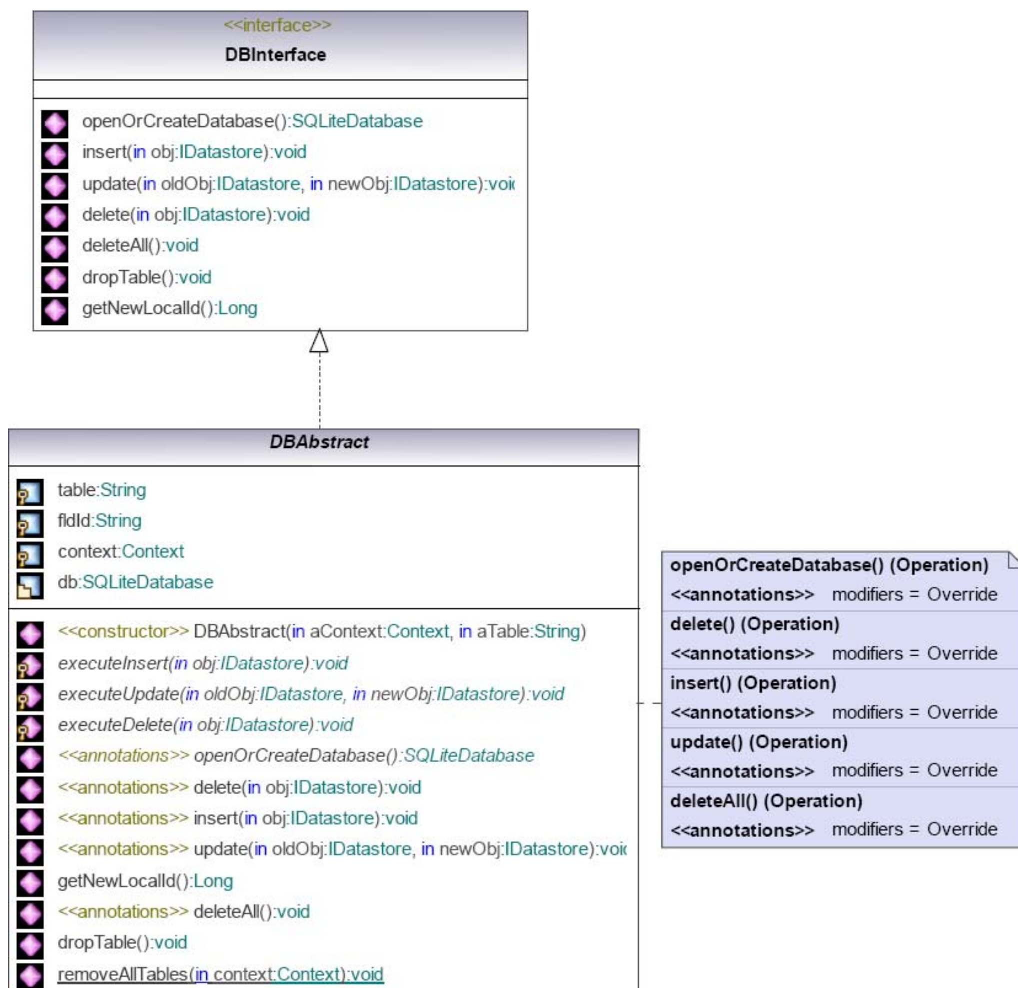


Figure 23 - Database hierarchy classes

All objects that use the database will extend DBAbstract class in order to access the database:

- DBCertificate
- DBContact
- DBContactDetails
- DBContactList
- DBNotifications
- DBPendingInvitations
- DBSettings
- DBStatus

Comparing this list with the list of tables on the server it can be seen that there are two new tables: settings and status. These tables are responsible for information storage by user configurations and have no connection with existing tables, so these remain unchanged.

Furthermore, an interface was created to distinguish which objects have access to the database. This interface was called IDatastore and is represented in the figure 22:

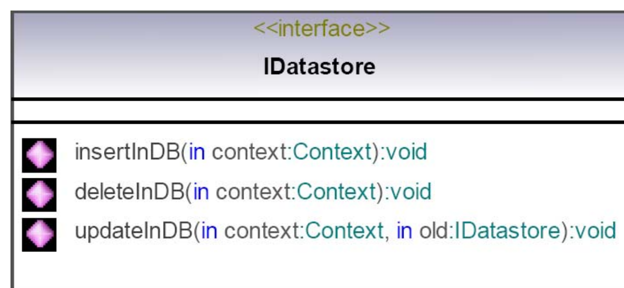


Figure 24 - IDatastore interface

Logically the objects that implement this interface are listed below and their hierarchy can be consulted in Appendix D. Whenever these objects need access to information stored in the database they will, through the methods defined by the interface, perform calls to the classes described above:

- Certificate
- Contact
- ContactDetails
- ContactListItem
- Notification
- PendingInvitation
- Settings
- Status

5.4.2 Layout

The implementation of the layout respects an interface named IActivityView shown in figure 23:

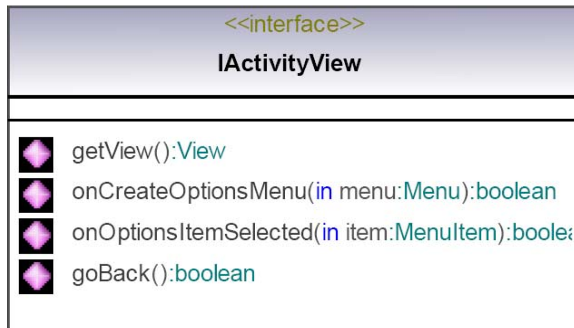


Figure 25 - IActivityView

It is ensured that the various parameters take into account the user activity and are respectful of the responsibility of each element. The whole hierarchy of the layout can be seen in Appendix D.

The methods represented by this interface refer to the standard popup menu of the operating system, as triggered by the back button in the device. The method getView will be called whenever it is necessary to build the layout. The activity stores the actual visible layout in a variable and delivers it whenever one of these operations is called.

Four independent layouts were developed:

- AuthenticationLayout
- RegistrationLayout
- StatusLayout
- Main

5.4.2.1 Authentication Layout

This layout allows the authentication of users. It consists in two input text fields: username and password, and two buttons: sign in and register.

The sign in button will validate authentication first on the device, e.g. invalid characters or empty fields, and then will make a request to server to check the data entered by the user. This operation is performed by an object created to handle the authentication data called Authentication. The authentication process runs in a new thread (*AuthenticationProcess*) freeing the layout to provide information to the user through a *ProcessDialog,* which is a dialog showing a progress indicator and a text message. If the data is correct it then receives and fills the database with all information from contacts received from the server and changes the view to the main view. The whole process is available in sequence diagrams at appendix F.

In other hand, if user does not have access, he or she should access the registration view.

5.4.2.2 Registration Layout

As authentication this view will be composed by a set of input text and two buttons: cancel and register. The information required for registration is based on a username, a password that the user needs to retype it and an email address.

As the server was deployed to Google App Engine, and the use of Google Apps domains in XMPP addresses is not yet supported for applications, a workaround had to be developed by using user's Google Account. This allows authentication on a GTalk server and for the device maintain a connection with the server to receive further notifications. Because of this limitation this need became essential to introduce Google Account data in the registration form.

In a similar way as the authentication, the registration process occurs in a different thread releasing the layout to present the *ProcessDialog.* The following validations are performed before registration is finalized:

- No field is left empty (validation on device)
- No invalid characters introduced (validation on device)
- Passwords are equals (validation on device)
- Google Account is correct (GTalk Server validation)
- Username doesn't exist (server validation)
- Email is not associated with other contact (server validation)

With registration completed successfully, the application receives the newly created profile for the user and writes into the database, and then the view is updated to the main view. The Registration process can be analyzed in appendix F.

5.4.2.3 Status Layout

The status layout allows for an easy way to manually change the status. This view is composed of three distinct parts:

1. The presentation of the current state and its mode: manual or automatic.
2. A input text allowing the introduction of a new status from a drop down list and allowing the association of a privacy level
3. A list of previously used status to facilitate reuse of the same with the possibility to change the associated privacy

Once the user defines a new status this view closes and returns to the previous one. If the user does not intend to change the status, the view can be removed by pressing the back button of the device.

5.4.2.4 Main Layout

The main layout corresponds to the main use of the application. This consists of four tabs which are:

- Profile Tab
- History Tab
- Contacts Tab
- Settings Tab

A new interface called *ItabInMain* was created and extended the previous *IActivityView* as shown in figure 24

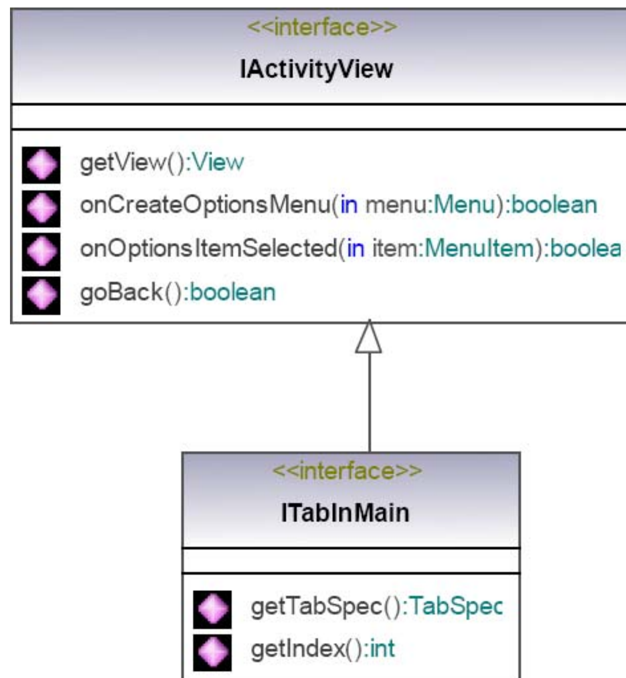


Figure 26 - Tabs hierarchy

This interface ensures that the main layout can access the tab specifications, essential for the layout design, as well as its identifier.

5.4.2.4.1 Profile Tab

The profile tab is responsible for displaying and maintaining the profile of the user. Thus two distinct ways of maintaining this layout were needed: in edit mode or in viewing mode. Depending on which mode is active the popup menu options will vary where options of revert and save are shown while profile is in edit mode, and edit and remove profile when it is in viewing mode.

Remove profile will show a dialog where users can remove their profiles locally and also in the server. It is not possible remove from the server and leave on the device. Removing from server will remove the ability of the user re-authenticate in the application, losing all existing information.

Edit Mode

When editing, contact details can be defined. Its layout is similar to the introduction of a new contact by the existing contacts application on the Android system. In order to implement this user interface, as shown in appendix G, several modules were designed. In order to maintain universality a class *ProfileItem* was created, which extends the *IActivityView* and *IContactDetailsItem* interfaces.

The first module presented in this layout is the *PName* allowing the introduction of username and avatar selection through a dialog that is presented after clicking the image. The next module to be presented is the status which is based in one text view displaying actual status and a button which allows to change the status. When this button is pressed a dialog is presented with the same layout built described in the section 5.4.2.3.

Then a list of *ProfileItem* fills the rest of the user interface by adding one for each type of detail used in the application. Each ProfileItem has the feature to add more information as the user wishes to each detail type. Each of this information line is implemented through an object named *ProfileItemLine.* This line will be represented by a label, a privacy level and its value and will be associated as a *ContactDetails* which will be introduced in its table when the contact will be saved.

Viewing Mode

The user interface when is in consultant mode consists of a list of all existing contact details in the database. So it can be represented to the user, a new class named *ContactDetailsView* was created enabling its reuse whenever it is necessary to show the profile of a particular user.

As the user's name, its image and status have no label, these data are handled in different way from all other kinds of details. These items are individually designed as shown in appendix G. On the other hand all other details have similar properties and easily allow the construction of the list of details to be shown to the user.

In order to use the contact details directly to build the user interface, a method called getView has been added to the *ListContactDetails* class which returns the view of contact details list that can be easily added to the *ListView* implemented by the profile tab.

5.4.2.4.2 History Tab

The history tab is responsible for all exposing all the activity happened in the device from the received calls to the different types of notifications. The call history is consulted by the contact application database. In order to access this information it is necessary to register permission in the AndoridManifest file as outlined in appendix B.

The collected information in this database is gathered in a single list with the information received from the application database through the tables *Notifications* and *PendingInvitations.* In order to manage these different types of information in a unique list, an interface named IHistoryLine was created and consists in getting an image, a message indicating the action and even a date of the event. After the data is correctly collected and the list filled, is sorted by date of the most recent to oldest. This sort was accomplished through a *Comparable* interface.

As the number of notifications can reach undesirable amounts compromising the functionality of history, a filter process was created offering the user a choice between different kinds of actions: call, notifications or invitations. This choice is maintained in the *Settings* table of the application database allowing to respect the user choices whenever he or she wishes to consult the history.

Through the menu, the user can remove all the history or only one kind of the elements presented in the history list. This menu is implemented through the methods defined by the interface IActivityView.

5.4.2.4.3 Contacts Tab

The contacts tab provides a list of contacts associated with the user and previously received from the server. This way no request is made to the server reducing considerably the time that it takes to complete and present the list to the user.

Each list item is composed of the user, his name, current status and a small icon informing the privacy level of that contact. In the case when a user does not have a status, one of the details existing in contact will be displayed in its place.

User can interact with this list in two ways: a simple click on one of the contacts or a long click. In the first case a full list of the contact details is presented as it is shown in the user profile. To obtain this view the previously created object, *ContactDetailsView,* is called. This operation can be reverted by pressing the back button of the device returning to the contact list view.

The second interaction method will display a list of available operations: view details, remove contact and change privacy. The view details option will take the same process as a simple click. The remove contact will make a request to server trough the CloudContact object resulting in the elimination of contact from the list. The last operation will display a dialog which allows to user select from a drop down list a new privacy level. Changing privacy level means that the information that the user shares to that contact will respect the new privacy selected. This operation does not change the privacy icon on his list, but in the contact list of the contact that privacy was changed.

When the user starts using the application and has no contact in list, a button will be displayed indicating the addition of a new contact. This addition will be made through a dialog that allows the introduction of contact’s nickname to be added as well as a privacy level. When user already has contacts in list, the user can continue to add contact through the menu which is implemented through the respective methods existing in the IActivityView which each tab implements.

5.4.2.4.4 Settings Tab

The Settings tab is meant to allow the user to configure the required settings. The available settings are:

- Enable or disable the notifications in the status bar
- Enable or disable the automatic status detection
- Set privacy for automatic status detection
- Set time interval for automatic status detection
- Select which sensors to be used for automatic status detection
- Set account data
- Set Google Account data

The default settings have notifications enable for invitation and replies, with contact updates notification disabled. Automatic status detection is on using all sensors available with a time interval for 5 minutes and an associated privacy to be shared to friends.

5.4.3 Cloud

Access to the cloud was implemented using a hierarchy similar to the database, and is shown in figure 25. *CloudAbstract* uses *RESTRequest* to make request to server. The methods *insert*, *get*, *update* and *delete* from *CloudAbstract* make calls to the methods *doPut*, *doGet*, *doPost* an *doDelete* existing in *RESTRequest*.

It is the responsibility of the classes that extend CloudAbstract to define the parameters to be sent to the server as the web service to use. The entire process is transparent between all objects. The objects defined for accessing the server are:

- CloudAuthentication
- CloudContact
- CloudContactList
- CloudInvitation
- CloudPendingInvitation
- CloudRescue
- CloudSync

All requests from the client use a certificate in order to be authenticated on the server. The certificate is composed by the certificate previously received from server, plus the existing XMPP connection Jabber ID and current ID from device. This information is put together by the RESTRequest class and then is encapsulated in the *Authorization* header of the http request.

In order to be able to manipulate the responses received from the server as universal as the requests are made, a class named *CloudResponse* was created as shown in figure 26. This class receives the response in xml from server identifying the certificate and detecting if an error occurred on server.

If no errors were detected the elements can get the body of the message which in turn are converted to their objects. In order to universalize this transformation an interface named *IXMLCapable* was created and can be found in appendix D.

In case of error, and if is a specific error stating that the certificate is old and needs to be updated, this same object is responsible for using *CloudSync* to receive the existing certificate on the server and start the synchronization process.

If no errors were detected the elements can get the body of the message which in turn are converted to their objects. In order to universalize this transformation an interface named *IXMLCapable* was created and can be found in appendix D.

In case of error, and if is a specific error stating that the certificate is old and needs to be updated, this same object is responsible for using *CloudSync* to receive the existing certificate on the server and start the synchronization process.

The synchronization process will remove all information about contacts from the database and will replace it with the new data from server. This would not be the desired solution, but due to time limitation it was implemented as a fast and functional solution.

Once the synchronization ends the object *CloudRescue* is called that will use the latest request information stored in *CloudAbstract* in order that the system that returned the old certificate can finally deliver its request.

Figure 27 - Cloud access hierarchy

Figure 28 - CloudResponse class

5.4.4 Contact

Each contact is based on profile sharing. While this is associated with the *Contact* object, all information on this profile is base on a list of *ContactDetails.* These details comprise a label, an identifier type an associated privacy and its value.

A class named *ListContactDetails* was created and extended from the java util *List* interface. Thus allows us to define a better access to details list and some details such as name and picture can be obtained directly. Also a list of a specific type can be obtained by passing the type identifier. The type of contact details used to implement this application was described on section 5.3.1.4.

The object *Contact* accesses the database through the *DBContact* which just implements the way the object uses the database through the connection and transaction monitoring responsibility of its super class *DBAbstract.* The *ContactDetails* object has a similar process as well as all others objects that need access to the database.

When it is necessary to communicate with the server, (ie if a profile is updated), the *Contact* object will use *CloudContact* in order to execute the request to server. This class only defines the web service which will be made the request as the parameters leaving the run task to their super class. For the purpose of *ContactDetails* object, this does not need to communicate with the server since it is the responsibility of *Contact* to send its list of details. This list of details is obtained a converted into xml in order to be sent to the server.

5.4.5 Status

In order to simplify and universalize access to information in a contact, the status was implemented as a *ContactDetails* without the need to associate a label and being associated with a privacy (set when is set a new manual status or globally for all automatically detected status). In this way this is available to other users as any other details of the contact.

However, when it necessary to obtain a status display to the user (as in the home screen widget or in the manual status change layout) the system first checks the settings to know if automatic status detection is on. If it is confirmed then status is obtained through the *ServiceBridge* object which gives the latest status detected from sensors. On the other hand if it is deactivated will query the database through the *DBStatus* object, which equally need only to inform how the table will be accessed by leaving the transaction and the connection to its super class, in order to obtain the current status.

5.4.6 Background Service

The application requires a service running in background in order to perform two important but distinct operations. A background service in the Android operating system is created using Service object and must be declared in the AndroidManifest.xml as shown in appendix B.

The first responsibility of this service is to ensure the communication with the server via the XMPP connection. The system authenticates the user through their Google Account previously defined and stored in the *Settings* table in the database, and creates a chat to the server Jabber ID: 'w-contact@appspot.com'. This way the client application will receive all the notifications from the server as explained later in the section 5.4.7.

The second responsibility of this service will be the awareness detection. A new thread is created to render the sensors independently and without affecting the connection previously established with the server. The whole process of awareness detection is described later in section 5.4.8

Because this service is indispensable for the good operation of the application there was the need to ensure that this service is always running. To achieve this goal the service was started in the boot of the mobile device. A class named *BootUp* was created and registered in AndroidManifest, available in appendix B, as a receiver taking action of BOOT_COMPLETED.

5.4.7 Notifications and Invitations

Notifications from the server are received via a XMPP chat pre opened with the server through the background service. This chat session is registered with a listener which is triggered whenever a message is received.

The system distinguishes from the possible notifications and takes the respective action. If an invitation is received a response layout is shown to the user where he can accept, reject or even change the proposed privacy level. This layout is shown through an *Intent* call to the wContact application carrying the flag that allows to the system identify the request and show the desired view.

In case of a notification of an invitation response, contact update or privacy change, the application will request from the server the information of the contact stored in database. Depending on the type of notifications, user settings will be checked to see if there is a need to display a notification in the status bar. If there is, a notification is created using the *NotificationManager* object provided by the Android framework. The process can be found in Appendix F.

Notifications received from the server are stored in the database in order to be shown to the users through the history tab. The last notification is also shown in the home screen widget.

5.4.8 Awareness

The BackroundService is responsible for automatic status detection (or awareness). Thus checks the settings of the user and if automatic detection is active it starts awareness detection.

Detection of awareness begins with the movement detection through the accelerometer sensor. If status was not detected, it moves on to the detection of environmental sound. To use these sensors it is necessary to check the user settings to ensure that the user is allowed the use of the respective sensors.

After this process and a status is detected, the battery level will be consulted and if it is below a certain percentage the status will be updated to 'low battery' disabling the awareness detection.

5.4.8.1 Accelerometer

A listener is registered on the SensorManager provided by the Android framework, and whenever a change occurs, it is called. Then a check is made in order to assure that the sensor that triggered the change was the accelerometer and then the axis data are stored in a collection. This process occurs over 30 seconds and when time is over the data is analyzed returning a status in case of a valid status detected.

5.4.8.2 Microphone

Environmental sound is recorded through a MediaRecorder object existing in Android framework. Before using it permission is added to AndroidMacifest.xml as represented in appendix B. Then a configuration of the recorder is made setting the audio source, output format and audio recorder.

Media recorder will record the sound to an output file and this is the only way that it works. Thus it is necessary to ensure that a file exists and the application has write permissions. The management of this file is the responsibility of this module and it needs permission to write in external data on the AndroidManifest.xml file as shown in appendix B. A file named 'a.3gp' is created and provided to MediaRecorder as the output file which will record the environment sound for 30 seconds.

During this recording at every second the maximum amplitude is requested and stored in a collection. Once the time expires the created file is deleted and the collection is analyzed to check if a new status is detected.

5.4.8.3 Battery

To have access to information from battery system needs to use a *broadcast receiver*. *Broadcast receiver* is a component that does nothing but receive and react to broadcast announcements. When the battery level is changed this is announced via broadcast. The value is stored in a global variable and used to compare with the pre defined value for low battery.

5.4.9 Widget

A screen widget was implemented as a simple UI component that users can place on the main home screen of their phone and offers simple information display and interaction. In this system, the widget is used to display the user's currently recorded status and also possesses a toggle button which enables them to switch between automatic and manual sharing of status.

Users can also tap on the status text and use it to select from a list of common and previous statuses (e.g. terms such as "Working"). This widget can be seen in Figure 27.

Figure 29- Home screen widget

Also in this widget shows a feed of the received notifications. An example is present in appendix G.

For the implementation of this widget a receiver must be inserted in configuration file of the application as represented in appendix B, and to the user interacting with the widget actions are also declared in the receiver.

A standard action exists and is the widget update that occurs at a preset time or whenever the widget is added to the screen. The other two existing actions refer to the user interaction with the widget. The first refers to the button that lets you to switch the mode of sharing status between automatic and manual, and the other is related when user taps on text opening a simple UI that allows the status selection UI. In this case the wContact will be call through an *Intent* passing an identifier variable indicating that is to show the status chooser layout. The description of this process is in appendix F, whereas their UIs can be found in appendix G.

5.4.10 Summary

This chapter described the implementation of the client application. This was developed for the Android operating system and has a simple UI that allows users to manage contacts. Communication with the server is performed through certificate validations, which are encapsulated in the header of HTTP requests. The contacts in list are updated through notifications received by a channel implemented using the XMPP Smack library. The application uses phone sensors to detect awareness information and this is shared as part of user's profile in the system.

CHAPTER 6 - Conclusion

The development of the wContact application represents a new direction for the management of contacts. Essentially, it is based on a model where the responsibility for taking care of their personal contact information is delegated to each user. No effort is made to keep contact information relating to other users updated and reliable as social-network connections between contacts seamlessly achieve this for each user. The layout of the application was closely modeled on existing contact management solutions in order to minimize the difficulties the new system might present in terms of usability.

Some of the major problems encountered while developing this application relate to limitations of the Google App Engine:

- XMPP limitation – Google App Engine provides a XMPP client for application that is deployed with IDs such in the form "app-id@appspot.com". The specification indicates that applications can also use custom addresses (e.g. "anything@app-id.appspotchat.com") as part of their functionality – such addresses can be used to represent individual user accounts in an App Engine application such as wContact. However, the use of Google Apps domains in XMPP addresses is not yet supported for applications. As wContact using Google App Engine, the GTalk XMPP server was mandatory, but as XMPP domains are not available for applications, the current system requires users to register with own Google Accounts in order to implement the notifications features. This is undesirable and should be changed as soon as Google App Engine implements custom XMPP domains for apps.
- Upload files – An application deployed in Google App Engine Servers cannot write to the file system, i.e., an application can read files, but only files uploaded with the application code. This limitation brought a major problem relating to user sharing profile photos as the application was not able to support picture upload. In order to circumvent this problem, the wContact application offers a list of avatars as a substitute for profile pictures.
- Web services response time – Another limitation from Google App Engine that may cause problems for wContact is the fact that each request to a web server in Google App Engine must respond within 30 seconds. If information to be retrieved from the datastore is overly large, the response time may be exceeded. In wContact, this may occur where there are extremely long lists of contacts. No stress tests were conducted to explore this issue as part of this thesis work.

Despite these limitations, Google App Engine is an affordable, rapid and convenient way to build and host an application and ensure that it is scalable.

The Android platform was an excellent choice on which to develop the wContact client application. On this platform, it is simple to implement, code and perform UI layout. Small restrictions were found relating to limitations in accessing databases or UI layout elements in child threads, but these were easily solved by passing handlers to child threads which give back operation results in order to show in layout or be managed by the database.

Allowing the user to share a status may yield an improvement in the quality of communication – it can serve as a low-impact channel for users to assess the suitability of

initiating a communication. In wContact, a manual status can be easily set and a UI enables previous status strings to be re-used, thereby reducing the effort of user interaction. wContact also explored automatic status detection in order to remove user intervention and provide a level of up-to-date awareness.

Status was shown on a screen widget. Besides keeping users alerted about their own status it also delivered UI support to switch between automatic and manual status and easy access to the UI to set a manual status. Future developments to this widget should add UI elements that provide access to the status history and to the contact list.

The automatic status detection was developed focusing on activities, rather than location, a topic that has received more attention in literature [6]. The benefits of focusing on activities include reduced privacy implications and directly appealing to users' ability to infer conclusions based on their own contextual knowledge. For example, at 2am, seeing that one friend is in a quiet environment, whilst another is in loud environment is probably sufficient to make useful inferences about their likely activities: one is probably sleeping, while the other is awake and likely to be socializing.

However, there are many avenues for future work on this topic. These include improving the existing recognition algorithms and also incorporating other sensors such as digital compasses, which may be capable of providing robust predictions. Similarly, greater awareness of device state, such as the ring profiles status as the ring profiles shared in Connecto [2], should be incorporated into the application. The addition of such capabilities may enable the detection of more nuanced activities such as "eating" or "meetings". However, a tradeoff is anticipated between the usefulness of particular classified statuses' and the accuracy with which they can be determined and this thesis suggests that, ultimately, there may be a place for very general classifications which users are required to interpret based on contextual knowledge.

The usage of automatic detection via the device sensors may significantly drain the battery. To counteract this, the use of the accelerometer and microphone sensors and respective analyzers was set to a periodicity of 5 minutes by default. Users could set longer intervals in to further reduce the impact on battery life, but this would be at the cost of reducing the accuracy and relevance of detected statuses. This issue should be explored more thoroughly in future work.

Due to time limitations during the wContact development, process synchronization between client and cloud was implemented simply; it is a potentially complex topic which is not a core element of the thesis work. The current implementation operates as follows: when the client detects that an update is required (through the certificate) it simply gets all information from cloud and replaces the information on the client. In order to assure that no information is lost, when the client does not have an internet connection features requiring transmission of data from the client to the cloud are disabled (a message indicates this is due to a lack on internet connection). Consequently, a user can still use the application without an Internet connection (for instance to access stored contact information) but cannot make changes (for instance, updating their profile information).

An important area of future work in wContact is privacy management. Privacy has become a hot topic in social networks and attracted significant attention in the research community. In wContact, the implementation of privacy controls is simple (based on four classifications of contacts) as this topic is beyond scope of the thesis. However, further developments of this model would be required to ensure that wContact is a flexible, powerful and usable contact management system.

Contact addition based on invitations is a useful tool that allows user to set privacy levels prior to sharing detailed information with others. However, it requires some improvement. One current limitation is that invitations are only possible to existing wContact users. In the future, users should be able to invite non-users through e-mail to join and register on the wContact application.

In order to make application fully accessible to everyone, a web platform version of wContact should be developed. This will attract users who do not own smart phones and provide those who do a multi-channel mechanism to access a shared database of contact information. Similarly, the client application should also be implemented on other smartphones such as iPhone and Symbian. Finally, wContact could also be extended through integration with third party services that need contact information, such as email accounts, virtual communities and social networking services.

In summary, the system was implemented as expected, and realized the distributed management of contacts. The application fundamentals consist of the sharing of profiles by users as well as their maintenance. As maintenance is the responsibility of the owner of each profile, it minimizes the possibility that the contacts on the list get out of date.

Another advantage of being the owner responsible for their profile is that it allows him or her to fill the information with various channels with which he can be contacted, thereby increasing the choice for users seeking to make contact.

A final benefit of the system relates to the automatically detected awareness information. By providing access to clues about others relating to availability wContact helps users choose the appropriate channel and appropriate time to communicate. In this way, the system aims to improve the quality of communication between human beings.

Bibliography

1. *Android Developers*. (n.d.). Retrieved from http://developer.android.com/guide/basics/what-is-android.html

2. Barkhuus, L., Brown, B., Bell, M., Sherwood, S., Hall, M., & Chalmers, M. (2008). From awareness to repartee. *CHI'08* (pp. 497-506). New York: ACM.

3. Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice, Second Edition.* Adison Wesley.

4. Chohan, N. B. (2009). AppScale: Scalable and Open AppEngine Application Development and Deployment. *International Conference on Cloud Computing.*

5. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., et al. (2009). AppScale: Scalable and Open AppEngine Application Development and Deployment. *CLOUDCOMP.*

6. Consolvo, S., Smith, I. E., Matthews, T., LaMarca, A., Tabert, J., & Powledge, P. (2005). Location disclosure to social relations: why, when, & what people want to share. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 81-90). Portland, Orlando, USA: CHI '05. ACM, New York.

7. Constantine, L. (2006). Activity Modeling: Toward a Pragmatic Integration of Activity Theory with Usage-Centered Design.

8. Constantine, L. (2005). User, Roles and Personas.

9. Danninger, M., Flaherty, G., Bernardin, K., Ekenel, H., Kohler, T., Malkin, R., et al. (2005). The Connector: facilitating context-aware communication. *7th International Conference on Multimodal Interfaces* (pp. 69-75). ICMI'05. ACM.

10. Erickson, T., & Kellog, W. A. (2000). Social translucence: an approach to designing systems that support social processes. *TOCHI*, (pp. 59-83).

11. Fogarty, J., Lai, J., & Christensen. (2004). Presence versus availability: the design and evaluation of a context-aware communication client. *Int. J. Hum.-Comput. Stud. 61, 3*, (pp. 299-317).

12. Google App Engine. (n.d.). Retrieved from http://code.google.com/appengine/docs/whatisgoogleappengine.html

13. Gross.R., & Acquissti, A. (2005). Information Revelation and Privacy in Online Social Networks. *Workshop onPrivacy in the Electronic Society.* ACM.

14. Gupta, S., & Kim, H. (2004). Virtual Community: Concepts, Implications, and Future Research Directions. *Proceedings of the Tenth Americas Conference on Information Systems*, (pp. 2679-2686). New York.

15. Heyer, C., Brereton, M., & Viller, S. (2008). Cross-channel mobile social software: an empirical study. *Proceeding of the Twenty-Sixth Annual SIGCHI Conference on Human Factors in Computing Systems* (pp. 1525-1534). Florence, Italy: ACM, New York, NY.

16. Horrigan, J. B. (2004). Online Communities: Networks That Nature Long-Distance Realationships and Local Ties. *Pew Internet and American Life Project.*

17. Hudson, S., Fogarty, J., Atkeson, C., Avrahami, D. F., Kiesler, S., Lee, J., et al. (2003). Predicting human interruptibility with sensors: aWizard of Oz feasibility study. *CHI* (pp. 257-264). ACM.

18. Lenk, A. K. (2009). What's inside the Cloud? An architectural map of the Cloud landscape. *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing* (pp. 23-31). Washington: International Conference on Software Engineering.

19. Liccope, C., & Heurtin, J. (2001). Managing One's Availability to Telephone Communication Through Mobile Phones: A French Case Study of the Development Dynamics of Mobile Phone Use. *Personal Ubiquitous Comput. 5, 2*, (pp. 99-108).

20. Mihalic, K., & Tscheligi, M. 2. (2007). 'Divert: mother-in-law': representing and evaluating social context on mobile devices. *Proceedings of the 9th international Conference on Human Computer interaction with Mobile Devices and Services* (pp. 257-264). Singapore: ACM.

21. Milewski, A. E., & Smith, T. M. (2000). Providing presence cues to telephone users. *CSCW'00.* ACM.

22. Morgan, R. M., & Hunt, S. D. (1994). The Commitment-Trust Theory of Relationship Markting. *Journal of Marketing*, (pp. 20-38).

23. Murray-Smith, R., Ramsay, A., Garrod, S., Jackson, M., & Musizza, B. (2007). Gait alignment in mobile phone conversations. *MobileHCI* (pp. 214-221). ACM.

24. Object Management Architecture™. (n.d.). Retrieved 2010 йил 08-06 from OMG: http://www.omg.org/oma/

25. Pedersen, E. R. (2001). Calls.calm: enabling caller and callee to collaborate. *CHI'01 Extended Abstracts* (pp. 235-236). New York: CHI'01. ACM.

26. Porter, C. E. (2004). A Typology of Virtual Communities: A Multi-Disciplinary Foundation for Future Research. *JCMC.*

27. Reily, K., Ludford, P. J., & Terveen, L. (2008). Sharescape: an interface for place annotation. *Proceedings of the 5th Nordic Conference on Human-Computer interaction: Building Bridges* (pp. 326-333). Lund, Sweden: ACM, New York, NY.

28. Richardson, L., & Ruby, S. (2007). *RESTful Web Services.* O'Reilly Media.

29. Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core.

30. Schwanen, T., & Kwan, M. (2008). The Internet, mobile phone and space-time constraints. *Geoforum*, (pp. 1362-1377).

31. Smith, M. A. (2000). Some social implications of ubiquitous wireless networks. . *SIGMOBILE Mob. Comput. Commun.* , (pp. 25-36).

32. Thomas, D. (2008). Enabling application agility: software as a service, cloud computing and dynamic languages. *Journal of Object Technology*, (pp. 29–32).

33. Towards Open Grid Services Architecture. (n.d.). Retrieved 2010 йил 08-06 from Globus: http://www.globus.org/ogsa/

34. Tuckman, B. W. (1965). Developmental Sequence in Small Groups. *Psychological Bulletin*, (pp. 384-399).

35. Wang, J., & Canny, J. (2006). End-user place annotation on mobile devices: a comparative study. *CHI '06 Extended Abstracts on Human Factors in Computing Systems* (pp. 1493-1498). Montréal, Québec, Canada: CHI '06. ACM, New York, NY.

36. Whittaker, S., Jones, Q., & Terveen, L. (2002). Contact Management: Identifying Contacts to Support Long-Term Communication. *CSCW*, (pp. 216-225).

37. Wind, Y., & Mahajan, V. (2002). Converging Marketing. *Journal of Interactive Marketing*, (pp. 64-79).

38. Zhang, X., Schiffman, J., Gibbs, S., Kunjithapatham, A., & Jeong, S. (2009). Securing elastic applications on mobile devices for cloud computing. *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, (pp. 127-134). Chicago.

wContact

Improving Social Communication

APPENDICES

Table of Contents

APPENDIX A –Serever Configuration Files 5

Jdoconfig.xml 6

Web.xml 6

Cron.xml 7

Appengine-web.xml 7

APPENDIX B – Client Configuration File 8

AndroidManifest.xml 9

APPENDIX C – Server UML Classes 10

DataStore 11

Servlets 12

CloudResponse 13

CertificateManager 13

NotificationProcess 14

PrivacyManager 15

APPENDIX D – Client UML Classes 16

Database 17

Layout 18

Layout – Profile Tab 19

Layout – HistoryTab 20

Layout - ContactsTab 21

Layout – SettingsTab 22

Cloud 23

Certificate 24

Authentication 25

Contact 26

Notifications and Invitations 27

BackgroundService 28

Awareness 29

Status 30

APPENDIX E - Server Sequence Diagrams 31

Contact – save() 32

Contact – delete() 33

Notification – createNotification() 33

ContactList - updatePrivacyInContactList() 34

ContactList – getList() 34

AuthenticationServlet GET Request 35
AuthenticationServlet PUT Request 36
ContactServlet GET Request 37
ContactServlet POST Request 38
ContactServlet DELETE Request 39
ContactListServlet GET Request 40
ContactListServlet POST Request 41
ContactListServlet PUT Request 42
ContactListServlet DELETE Request 43
NotifyServlet GET Request 44
PendingInvitationServlet PUT Request 45
PendingInvitationsServlet DELETE Request 46
SyncServlet POST Request 47
XMPPServlet POST Request 48
CloudResponse 49
setErrorResponse() 49
sendResponse() 49
CertificateManager – constructor 50
CertificateManager – validateCertificate() 51
NotificationManager – sendInvitationNotification() 52
NotificationManager – sendNotification() 53
PrivacyManager – applyFilterInContact 54
APPENDIX F – Client Sequence Diagrams 55
DBAbstract – delete() 56
DBAbstract – insert() 57
DBAbstract – update() 58
DBAbstract – newLocalId() 59
CloudAbstract-update() 60
RESTRequest-doPut() 61
CloudAbstract - get() 62
RESTRequest – doGet() 63
CloudAbstract- insert() 64
RESTRequest – doPut() 65
CloudAbstract – delete() 66
RESTRequest – doDelete() 67
CloudResponse 68

Certificate – save() 69
AuthenticationLayout – signin() 70
AuthenticationProcess – run() 71
Authentication – ValidateOnCloud() 72
RegistrationLayout – regist() 73
RegistrationProcess – run() 74
ProfileTab – saveProfile() 75
ProfileItem – updateContactDetails() 75
SaveProfileProcess – run() 76
HistoryTab – populateCallList() 77
HistoryTab – populateNotificationsList() 78
HistoryTab – populateInvitationsList() 79
Sort history list 79
SettingsTab - activateAutomaticStatusDetection 80
SettingsTab - deactivateAutomaticStatusDetection 81
Tab creation 82
Contact – save() 82
Contact – saveOnCloud() 83
Contact – insertInDB() 83
ListContactDetails – getName() 84
PendingInvitations – reject() 85
PendingInvitations – accept() 86
BackgroundService – onCreate() 87
BackgroundService – connect() 88
BackgroundService – updateWidgetFeed() 89
BackgroundService – showNotificaton() 90
BootUp – onReceive() 90
AwarenessProcess – run() 91
AccelerometerProcess – run() 92
MicrophoneProcess – run() 93
Status – getStatus() 94
Status – save() 95
APPENDIX G – Application Screenshots 96
APPENDIX H – User Research Survey 103

APPENDIX A

Server Configuration Files

Jdoconfig.xml

```
<?xml version="1.0" encoding="utf-8"?>
<jdoconfig xmlns="http://java.sun.com/xml/ns/jdo/jdoconfig"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://java.sun.com/xml/ns/jdo/jdoconfig">

   <persistence-manager-factory name="transactions-optional">
       <property name="javax.jdo.PersistenceManagerFactoryClass"
           value="org.datanucleus.store.appengine.jdo.DatastoreJDOPersistenceManagerFactory"/>
       <property name="javax.jdo.option.ConnectionURL" value="appengine"/>
       <property name="javax.jdo.option.NontransactionalRead" value="true"/>
       <property name="javax.jdo.option.NontransactionalWrite" value="true"/>
       <property name="javax.jdo.option.RetainValues" value="true"/>
       <property name="datanucleus.appengine.autoCreateDatastoreTxns" value="true"/>
       <property name="datanucleus.appengine.datastoreReadConsistency" value="EVENTUAL" />
   </persistence-manager-factory>
</jdoconfig>
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Servlets -->
  <servlet>
    <servlet-name>greetServlet</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.GreetingServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>greetServlet</servlet-name>
    <url-pattern>/wcontact/greet</url-pattern>
  </servlet-mapping>

   <servlet>
    <servlet-name>Authentication</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.AuthenticationServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Authentication</servlet-name>
    <url-pattern>/Authentication</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>Contact</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.ContactServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Contact</servlet-name>
    <url-pattern>/Contact</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>Sync</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.SyncServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Sync</servlet-name>
    <url-pattern>/Sync</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>ContactList</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.ContactListServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ContactList</servlet-name>
    <url-pattern>/ContactList</url-pattern>
  </servlet-mapping>

   <servlet>
    <servlet-name>PendingInvitations</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.PendingInvitationServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>PendingInvitations</servlet-name>
    <url-pattern>/PendingInvitations</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>notify</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.NotifyServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>notify</servlet-name>
```

```
    <url-pattern>/notify</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>xmpp</servlet-name>
    <servlet-class>ns.appengine.wcontact.server.servlet.XMPPServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>xmpp</servlet-name>
    <url-pattern>/_ah/xmpp/message/chat/</url-pattern>
  </servlet-mapping>


  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>WContact.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/notify</url>
    <description>Check and dispatch notifications every minute</description>
    <schedule>every 1 minutes</schedule>
  </cron>
</cronentries>
```

Appengine-web.xml

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
<application>w-contact</application>
<version>1</version>
<!-- Configure java.util.logging -->
<system-properties>
<property name="java.util.logging.config.file" value="WEB-INF/logging.properties"/>
</system-properties>
<inbound-services>
     <service>xmpp_message</service>
  </inbound-services>
</appengine-web-app>
```

APPENDIX B

Client Configuration Files

AndroidManifest.xml

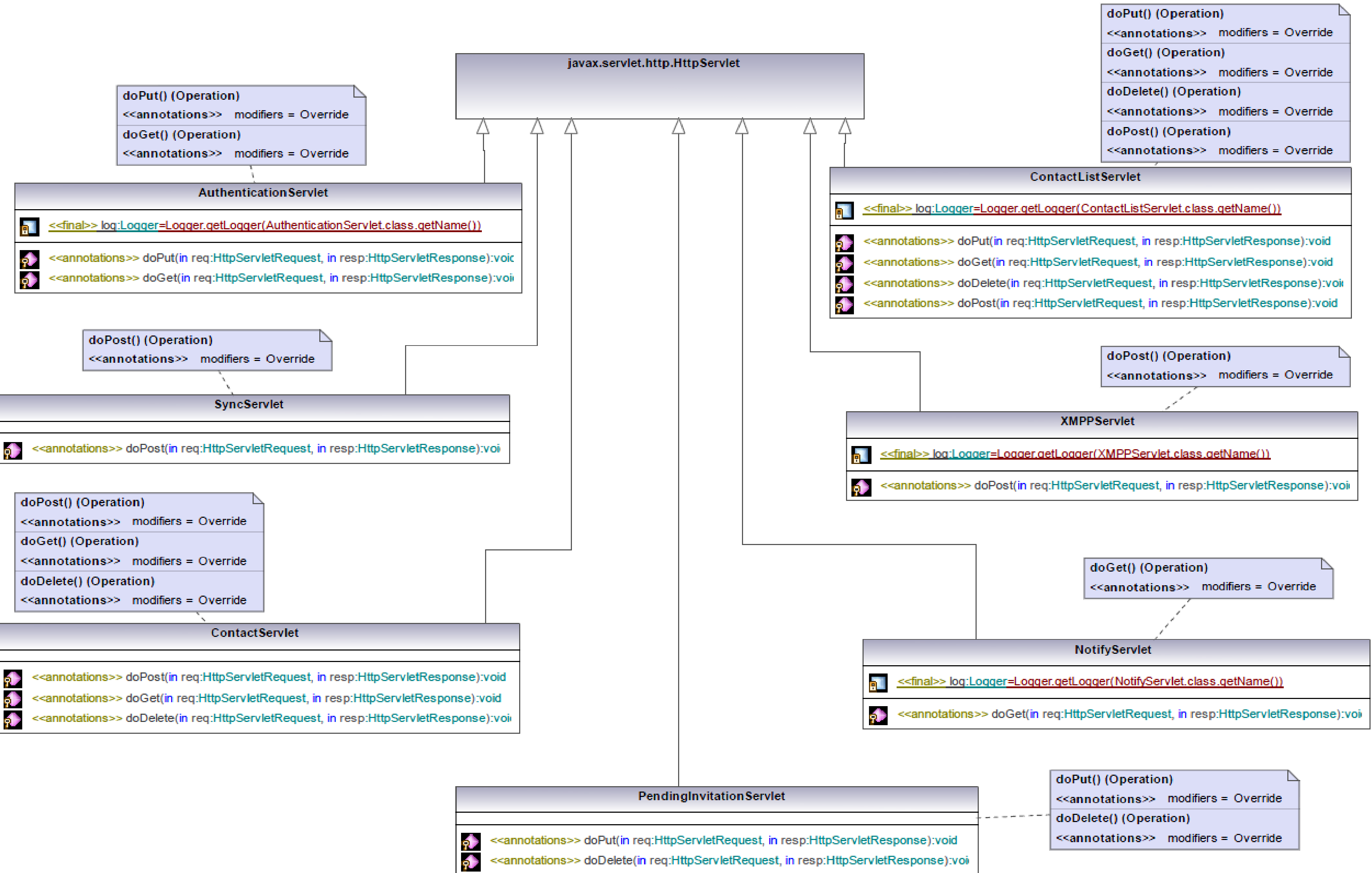
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ns.android.wcontact"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
  <application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".WContact"
          android:label="@string/app_name"
          android:screenOrientation="portrait">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <service android:name=".service.BackgroundService" />
    <receiver android:enabled="true" android:name=".xmpp.BootUp"
          android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
          <intent-filter>
                  <action android:name="android.intent.action.BOOT_COMPLETED" />
                  <category android:name="android.intent.category.DEFAULT" />
          </intent-filter>
    </receiver>
    <receiver android:name=".widget.WContactWidget" android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
        <action android:name=".widget.WContactWidget.ACTION_WIDGET_STATUS_BTN"/>
         <action android:name=".widget.WContactWidget.ACTION_WIDGET_AUTOMATIC_BTN"/>
      </intent-filter>
      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/wcontact_widget" />
    </receiver>
  </application>
  <uses-sdk android:minSdkVersion="4" />

</manifest>
```

APPENDIX C

Server UML Classes

DataStore

Servlets



CloudResponse

<<interface>>
XMLCapable
toXML():String
getXMLElements(in doc:Document):Elemer

XMLCapableAbstract
<<final>> log:Logger=Logger.getLogger(XMLCapableAbstract.class.getName())
toXML():String
getXMLElements(in doc:Document):Element

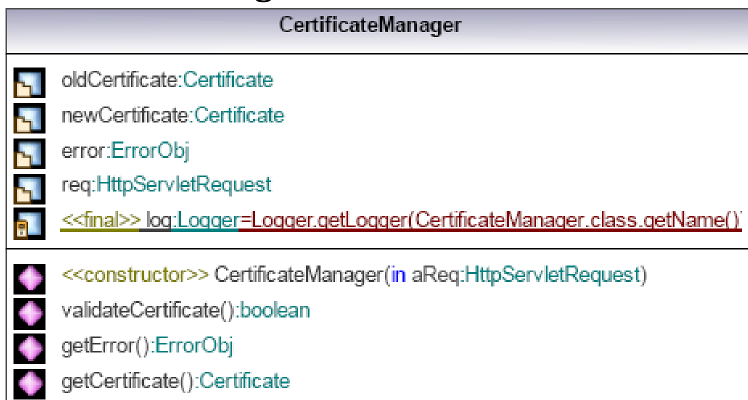
getXMLElements() (Operation)
<<annotations>> modifiers = Override

ErrorObj
<<final>> log:Logger=Logger.getLogger(ErrorObj.class.getName())
code:int
value:String
<<final>> INVALID_PARAMETERS:int=1100
<<final>> INVALID_PARAMETERS_DESC:String="Invalid parameters!"
<<final>> INVALID_CERTIFICATE:int=3100
<<final>> INVALID_CERTIFICATE_DESC:String="Invalid certificate!"
<<final>> INEXISTENT_CERTIFICATE:int=3200
<<final>> INEXISTENT_CERTIFICATE_DESC:String="Certificate doesn't exist."
<<final>> OLD_CERTIFICATE:int=3300
<<final>> OLD_CERTIFICATE_DESC:String="Certificate is old. Update is needed."
<<final>> WRONG_CERTIFICATE:int=3400
<<final>> WRONG_CERTIFICATE_DESC:String="Wrong certificate"
<<final>> EMAIL_EXISTS:int=5100
<<final>> EMAIL_EXISTS_DESC:String="Email already registered"
<<final>> USERNAME_EXISTS:int=5200
<<final>> USERNAME_EXISTS_DESC:String="Username already registered"
<<final>> USER_PASS_MISMATCH:int=5300
<<final>> USER_PASS_MISMATCH_DESC:String="User and pass doesn't match"
<<final>> USERNAME_NOT_EXISTS:int=5400
<<final>> USERNAME_NOT_EXISTS_DESC:String="Username doesn't exist"
<<final>> CONTACT_NOT_EXISTS:int=7100
<<final>> CONTACT_NOT_EXISTS_DESC:String="Contact doesn't exist in server"
<<final>> CONTACT_NOT_EXISTS_IN_LIST:int=7200
<<final>> CONTACT_NOT_EXISTS_IN_LIST_DESC:String="Contact doesn't exist in
<<final>> CONTACT_EXIST_IN_LIST:int=7300
<<final>> CONTACT_EXIST_IN_LIST_DESC:String="Contact already exists in list"
<<final>> INVITATION_REMOVED:int=7400
<<final>> INVITATION_REMOVED_DESC:String="Invitation was removed"
<<final>> XMPP_INVALID_ARG:int=9100
<<final>> XMPP_INVALID_ARG_DESC:String="Invalid argument"
UNDEFINED:String="Undefined error!"
<<constructor>> ErrorObj(in aCode:int)
getCode():int
getValue():String
<<annotations>> getXMLElements(in doc:Document):Element

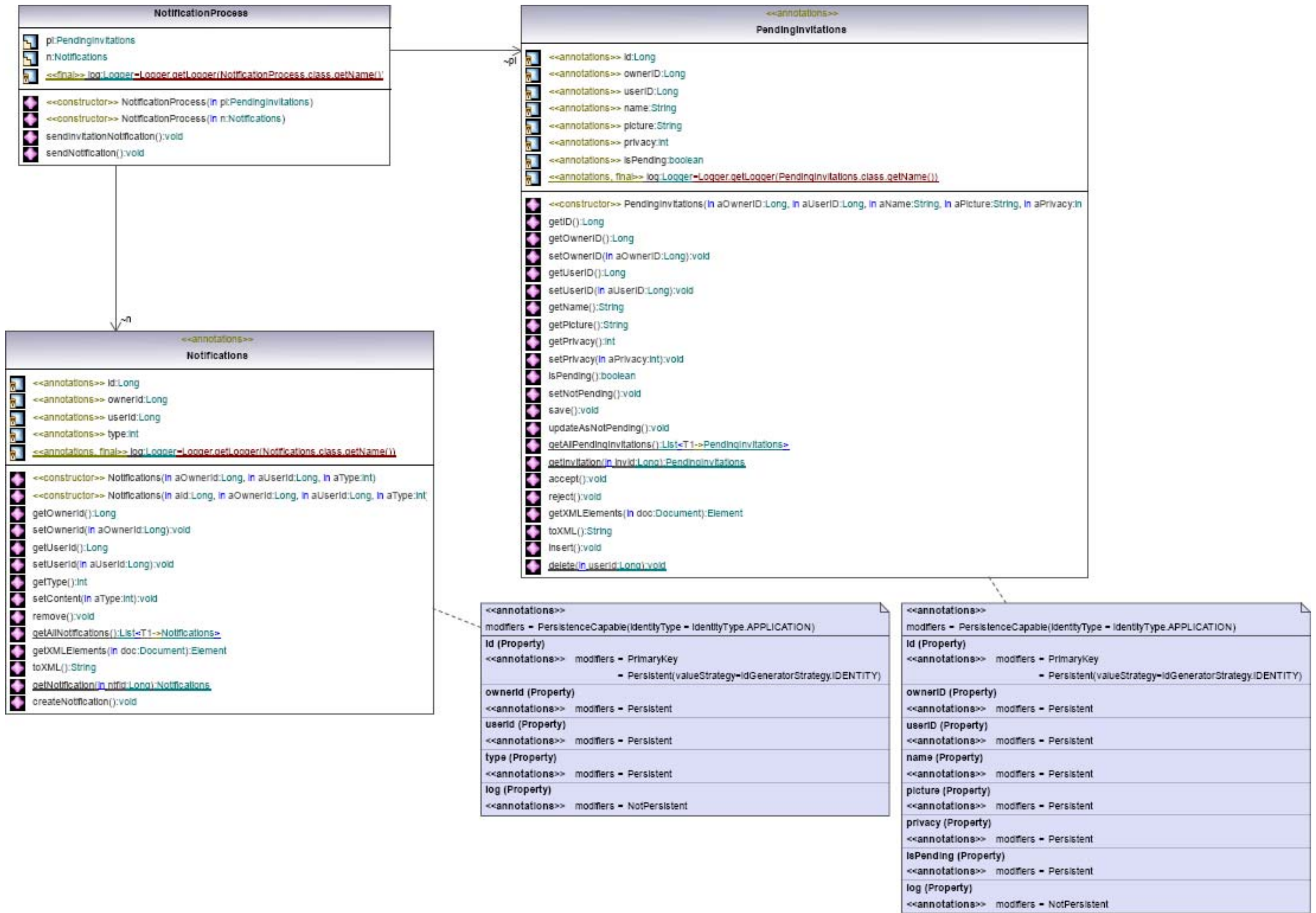
CloudResponse
certificate:Certificate
body:Object
pw:PrintWriter
<<constructor>> CloudResponse(in resp:HttpServletResponse
setResponse(in aCertificate:Certificate, in aBody:Object):void
setErrorResponse(in error:ErrorObj):void
<<annotations>> getXMLElements(in doc:Document):Element
sendResponse():void

getXMLElements() (Operation)
<<annotations>> modifiers = Override

CertificateManager

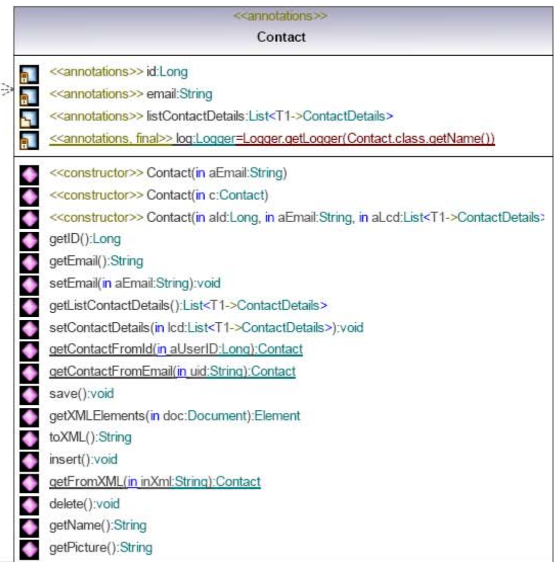


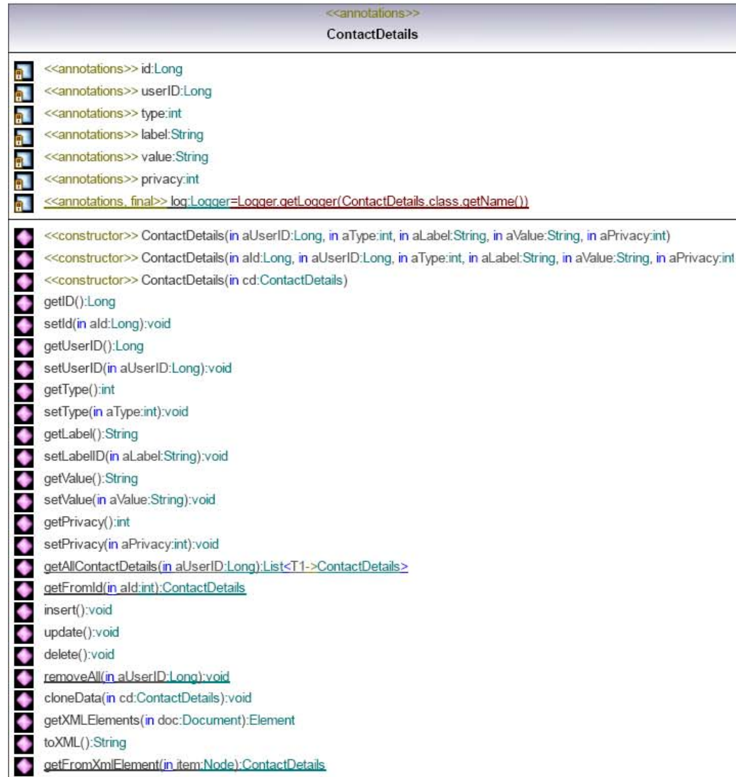
NotificationProcess



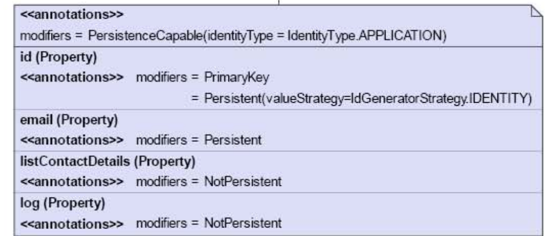
PrivacyManager

<<use>>





~listContactDetails



APPENDIX D

Client UML Classes

Database

Layout

Layout – Profile Tab

<<interface>>
IActivityView

- getView():View
- onCreateOptionsMenu(in menu:Menu):boolean
- onOptionsItemSelected(in item:MenuItem):boolea
- goBack():boolean

<<interface>>
ITabInMain

- getTabSpec():TabSpec
- getIndex():int

<<interface>>
IContactDetailsItem

- updateContactDetails(in lcd:ListContactDetails):voi

updateContactDetails() (Operation)
<<annotations>> modifiers = Override
onCreateOptionsMenu() (Operation)
<<annotations>> modifiers = Override
onOptionsItemSelected() (Operation)
<<annotations>> modifiers = Override
goBack() (Operation)
<<annotations>> modifiers = Override

ProfileTab

- parent:WContact
- spec:TabSpec
- tHost:TabHost
- index:int=0
- contact:Contact
- isEditable:boolean=false
- rlStatus:RelativeLayout
- tvValue:TextView
- ll:LinearLayout
- sv:ScrollView
- listProfileItems:List<T1->ProfileItem>
- pd:ProgressDialog
- saveHandler:Handler=new Handler() { public void handleMessage(android.os.Message msg) { pd.dismiss()

- <<constructor>> ProfileTab(in aParent:WContact, in aTHost:TabHost, in editable:boolean)
- <<annotations>> createTabContent(in tag:String):View
- getTabSpec():TabSpec
- <<annotations>> getIndex():int
- getEditableView():View
- getNormalView():View
- <<annotations>> getView():View
- <<annotations>> onCreateOptionsMenu(in menu:Menu):boolean
- <<annotations>> onOptionsItemSelected(in item:MenuItem):boolean
- revertChanges():void
- saveProfile():void
- <<annotations>> goBack():boolean

~listProfileItems

ProfileItem

- activity:Activity
- lines:List<T1->ProfileItemLine>
- mainRelativeLayoutId:int
- captionLayoutId:int
- captionText:String
- type:int
- contact:Contact
- lcd:List<T1->ContactDetails>
- llmain:LinearLayout

- getNewLine(in cd:ContactDetails):ProfileItemLine
- <<constructor>> ProfileItem(in aActivity:Activity, in aType:int, in aContact:Contact)
- getType():int
- getView():View
- getMainRelativeLayoutParams():RelativeLayout.LayoutParams
- addLine(in pel:ProfileItemLine):void
- removeLine(in aId:int):void
- <<annotations>> updateContactDetails(in lcd:ListContactDetails):void
- getActivity():Activity
- getID():int
- getLines():List<T1->ProfileItemLine>
- <<annotations>> onCreateOptionsMenu(in menu:Menu):boolean
- <<annotations>> onOptionsItemSelected(in item:MenuItem):boolean
- <<annotations>> goBack():boolean

-parent

createTabContent() (Operation)
<<annotations>> modifiers = Override
getIndex() (Operation)
<<annotations>> modifiers = Override
getView() (Operation)
<<annotations>> modifiers = Override
onCreateOptionsMenu() (Operation)
<<annotations>> modifiers = Override
onOptionsItemSelected() (Operation)
<<annotations>> modifiers = Override
goBack() (Operation)
<<annotations>> modifiers = Override

onCreateOptionsMenu() (Operation)
<<annotations>> modifiers = Override
onOptionsItemSelected() (Operation)
<<annotations>> modifiers = Override
goBack() (Operation)
<<annotations>> modifiers = Override

~lProfileItems

-contact

#contact

#lines

deleteInDB() (Operation)
<<annotations>> modifiers = Override
insertInDB() (Operation)
<<annotations>> modifiers = Override
updateInDB() (Operation)
<<annotations>> modifiers = Override
fromXml() (Operation)
<<annotations>> modifiers = Override
toXml() (Operation)
<<annotations>> modifiers = Override
getXmlElements() (Operation)
<<annotations>> modifiers = Override

Contact

- id:Long
- email:String
- type:int
- listContactDetails:ListContactDetails

- <<constructor>> Contact(in aId:Long, in aEmail:String, in aListContactDetails:ListContactDetails
- <<constructor>> Contact()
- getFromDataBase(in aContext:Context, in cId:Long):Contact
- getID():Long
- getEmail():String
- getType():int
- getContactDetails():ListContactDetails
- <<annotations>> deleteInDB(in context:Context):void
- <<annotations>> insertInDB(in context:Context):void
- <<annotations>> updateInDB(in context:Context, in old:IDatastore):void
- saveOnCloud(in context:Context):Contact
- removeInCloud(in aContext:Context):void
- getFromCloud(in context:Context, in aUserId:Long, in privacy:Privacy):Contact
- <<annotations>> fromXml(in root:Node):IXMLCapable
- <<annotations>> toXml():String
- <<annotations>> getXmlElements(in serializer:XmlSerializer):void
- getFromXml(in body:Node):Contact
- save(in context:Context):void

PName

- activity:Activity
- etName:EditText
- ivPicture:ImageView
- cdName:ContactDetails
- cdPicture:ContactDetails

- <<constructor>> PName(in aActivity:Activity, in aContact:Contact)
- <<annotations>> getView():View
- <<annotations>> updateContactDetails(in lcd:ListContactDetails):voi
- getName():String
- <<annotations>> onCreateOptionsMenu(in menu:Menu):boolean
- <<annotations>> onOptionsItemSelected(in item:MenuItem):boolean

ProfileItemLine

- parent:ProfileItem
- activity:Activity
- hint:String=""
- inputtype:int
- type:int
- cd:ContactDetails
- sLabel:Spinner
- sPrivacy:Spinner
- etInput:EditText
- aaLabel:ArrayAdapter<T1->String>

- <<constructor>> ProfileItemLine(in aParent:ProfileItem, in aCd:ContactDetails
- getView():View
- getContactDetail():ContactDetails
- <<annotations>> onCreateOptionsMenu(in menu:Menu):boolean
- <<annotations>> onOptionsItemSelected(in item:MenuItem):boolean
- <<annotations>> goBack():boolean

getView() (Operation)
<<annotations>> modifiers = Override
updateContactDetails() (Operation)
<<annotations>> modifiers = Override
onCreateOptionsMenu() (Operation)
<<annotations>> modifiers = Override
onOptionsItemSelected() (Operation)
<<annotations>> modifiers = Override

-contact

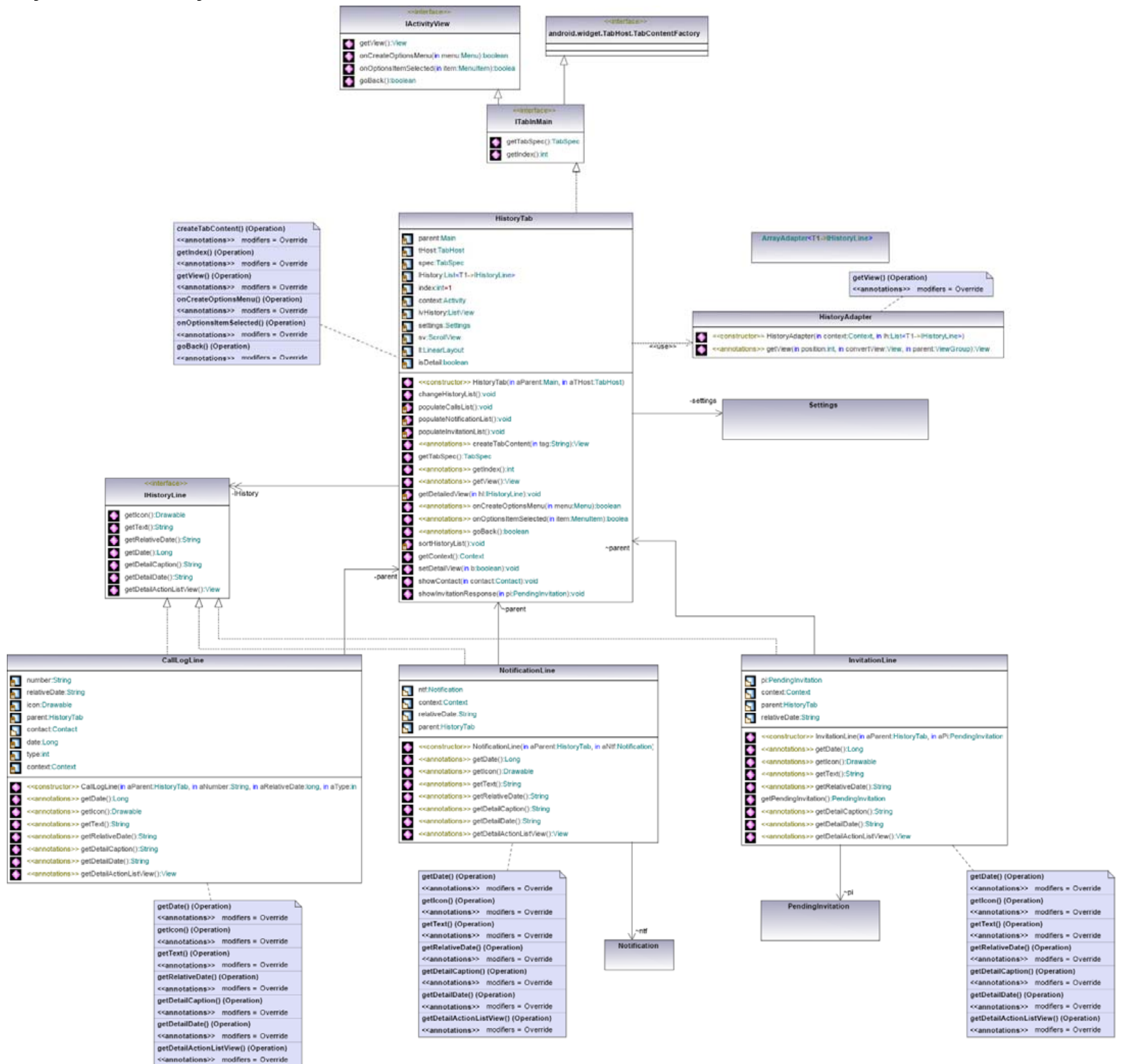
run() (Operation)
<<annotations>> modifiers = Override

SaveProfileProcess

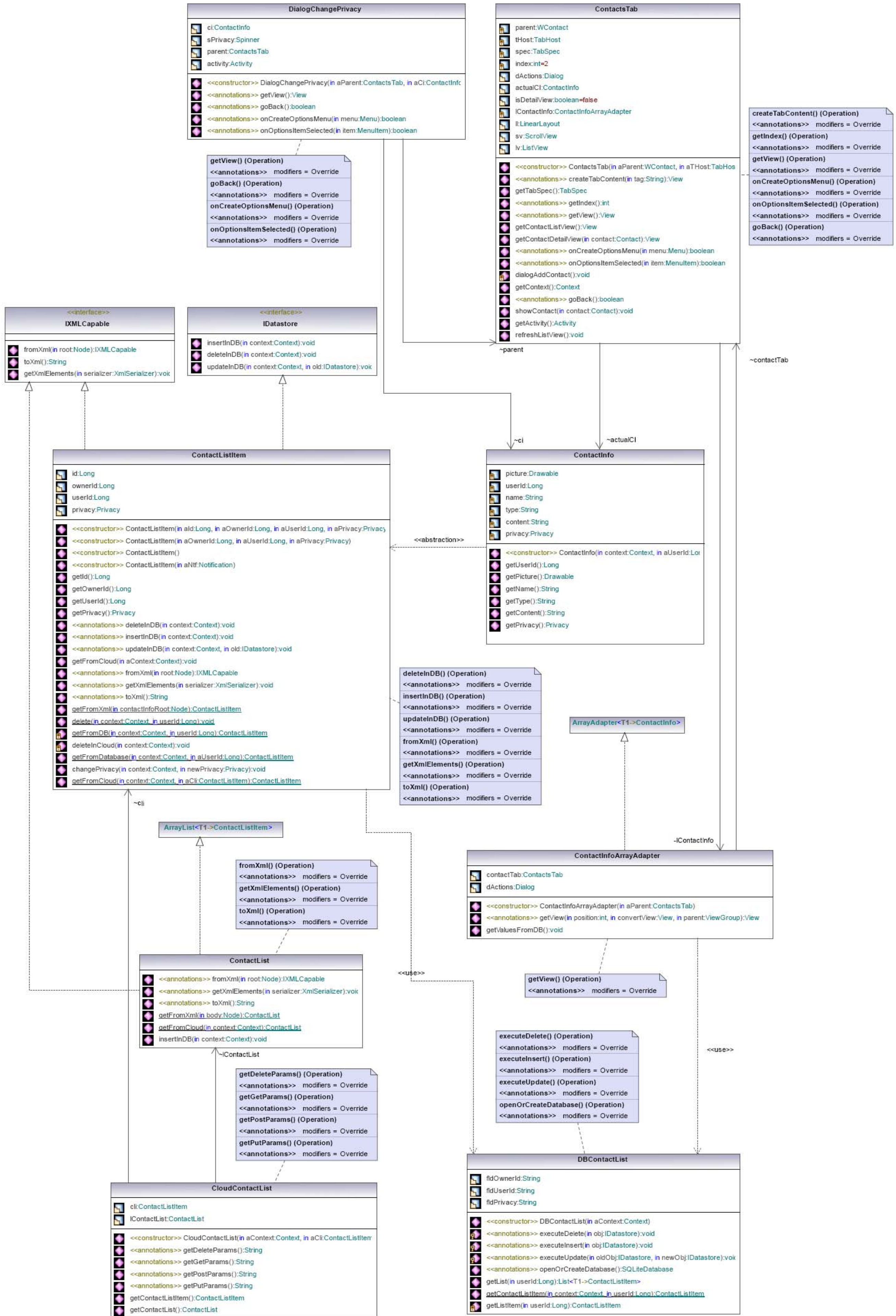
- context:Context
- lProfileItems:List<T1->ProfileItem>
- contact:Contact
- handler:Handler

- <<constructor>> SaveProfileProcess(in aContext:Context, in aListProfileItems:List<T1->ProfileItem>, in aContact:Contact, in aHandler:Handle
- <<annotations>> run():void

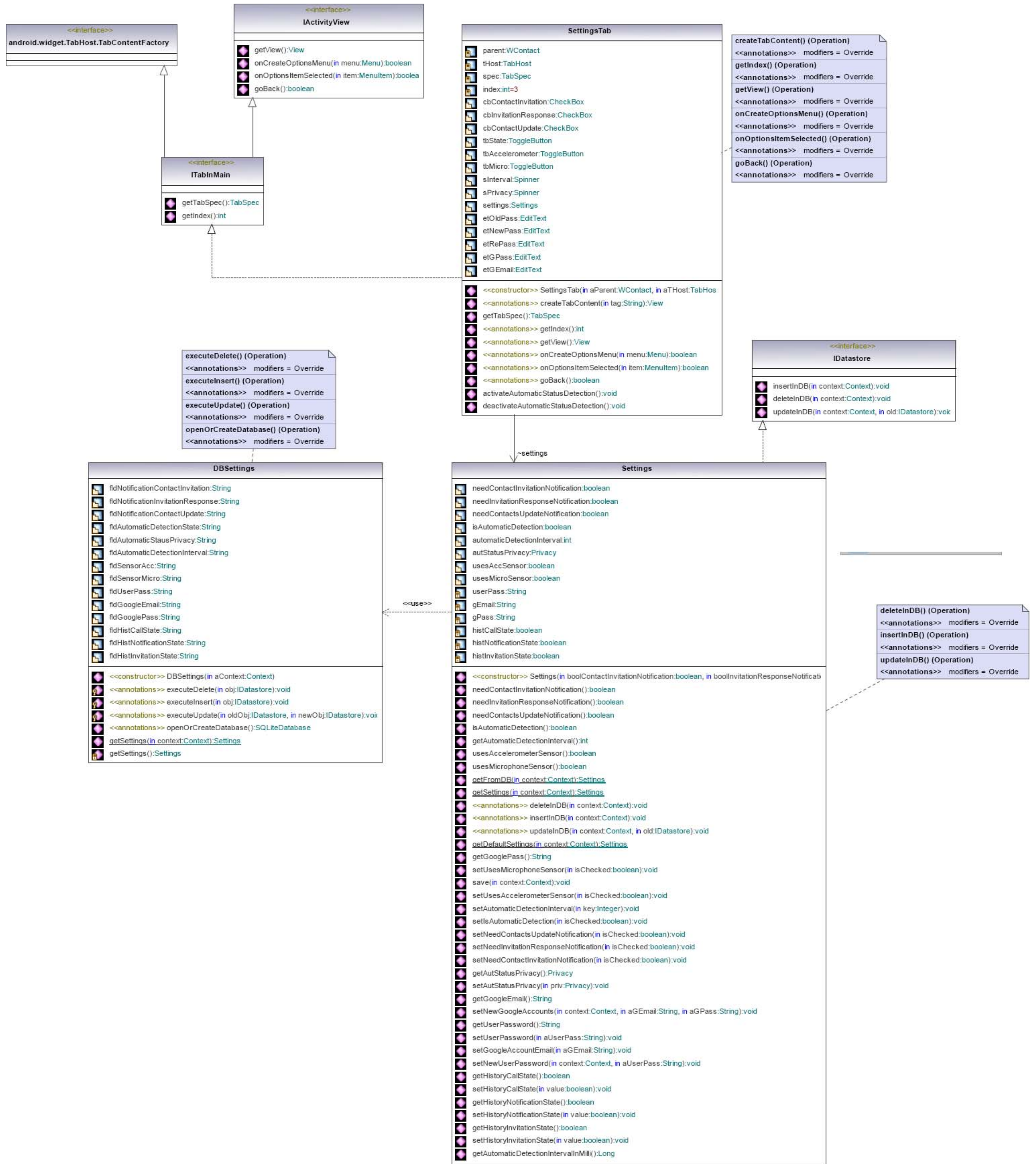
Layout – HistoryTab



Layout - ContactsTab



Layout – SettingsTab

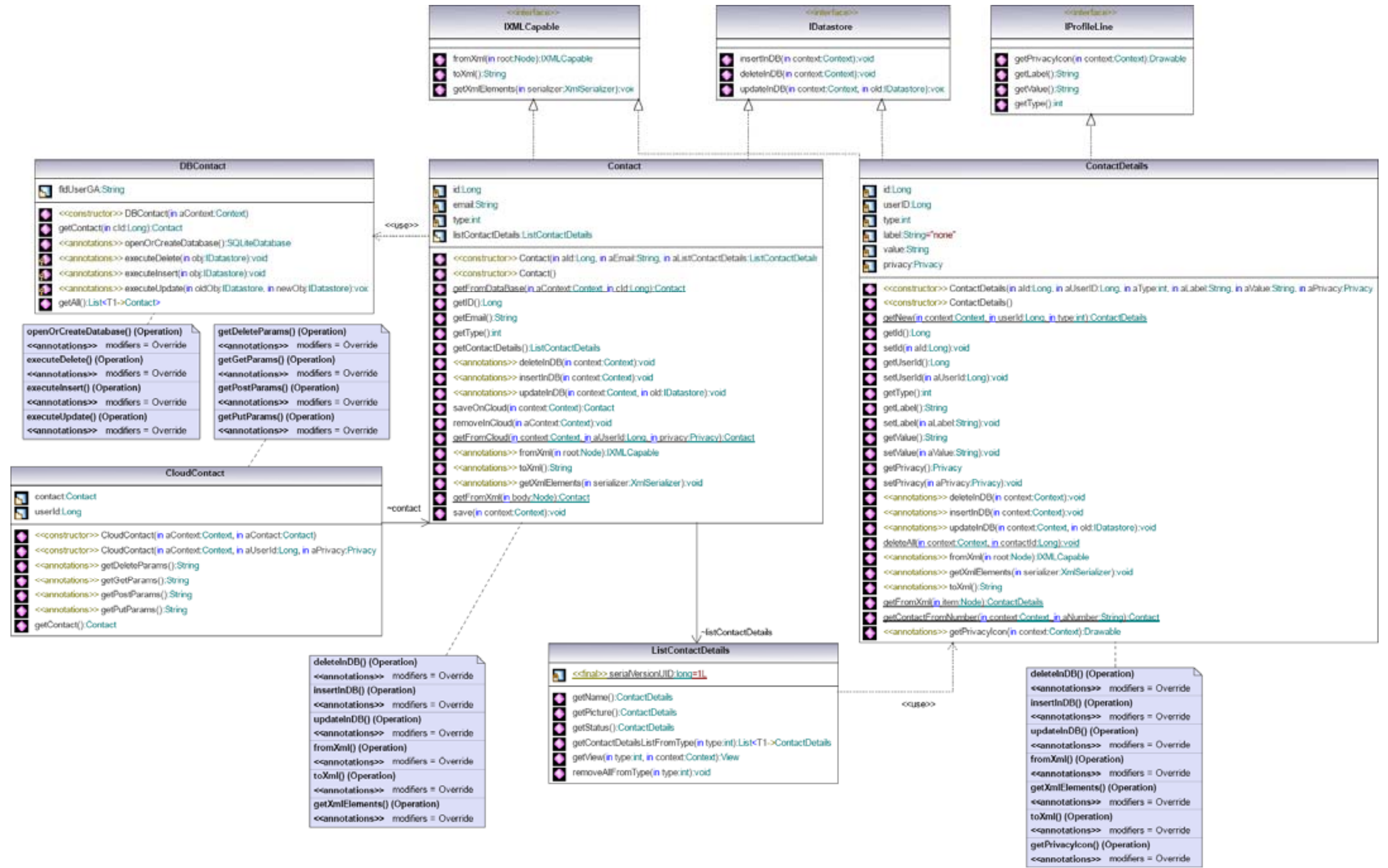


Cloud

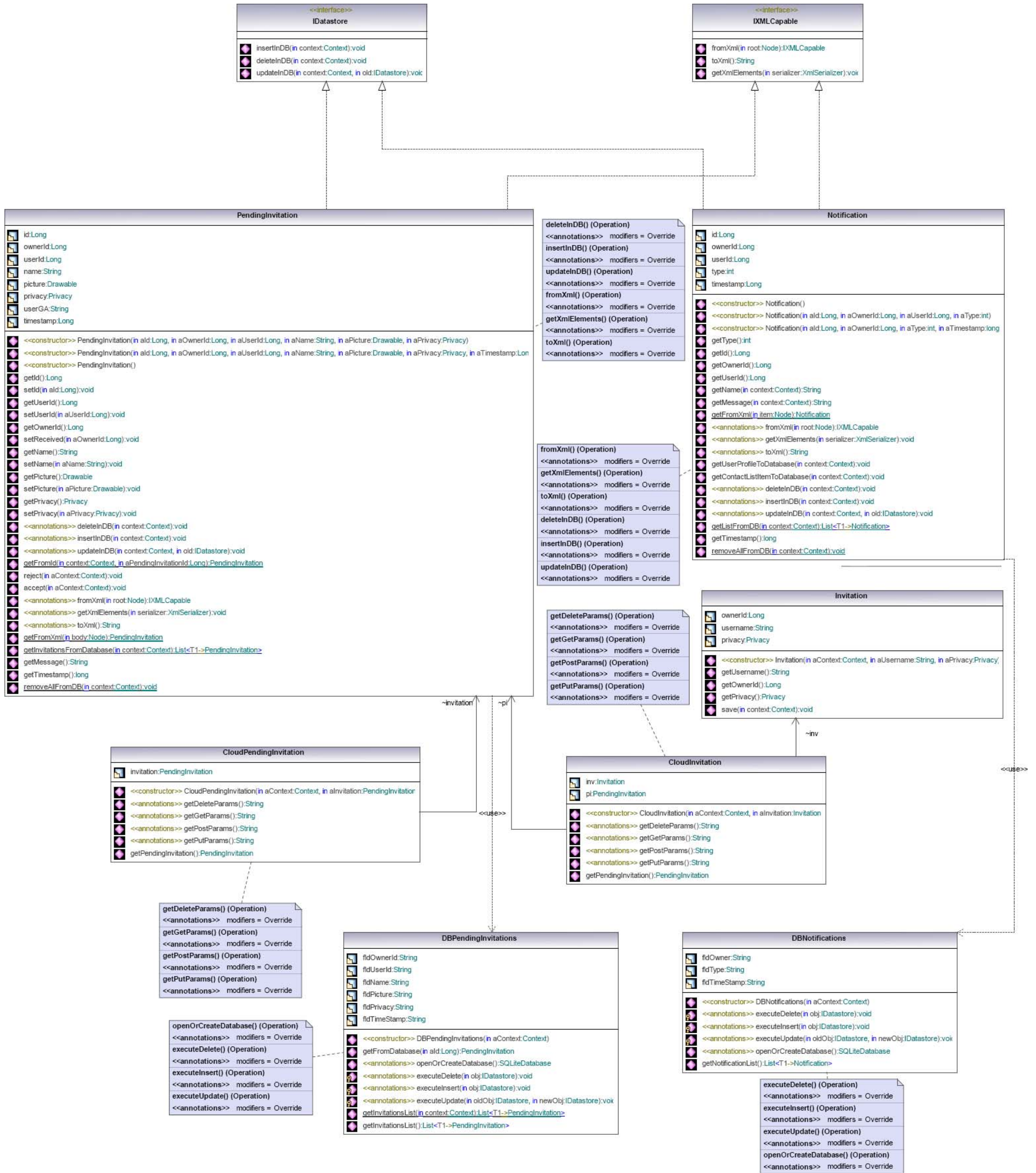
Certificate

Authentication

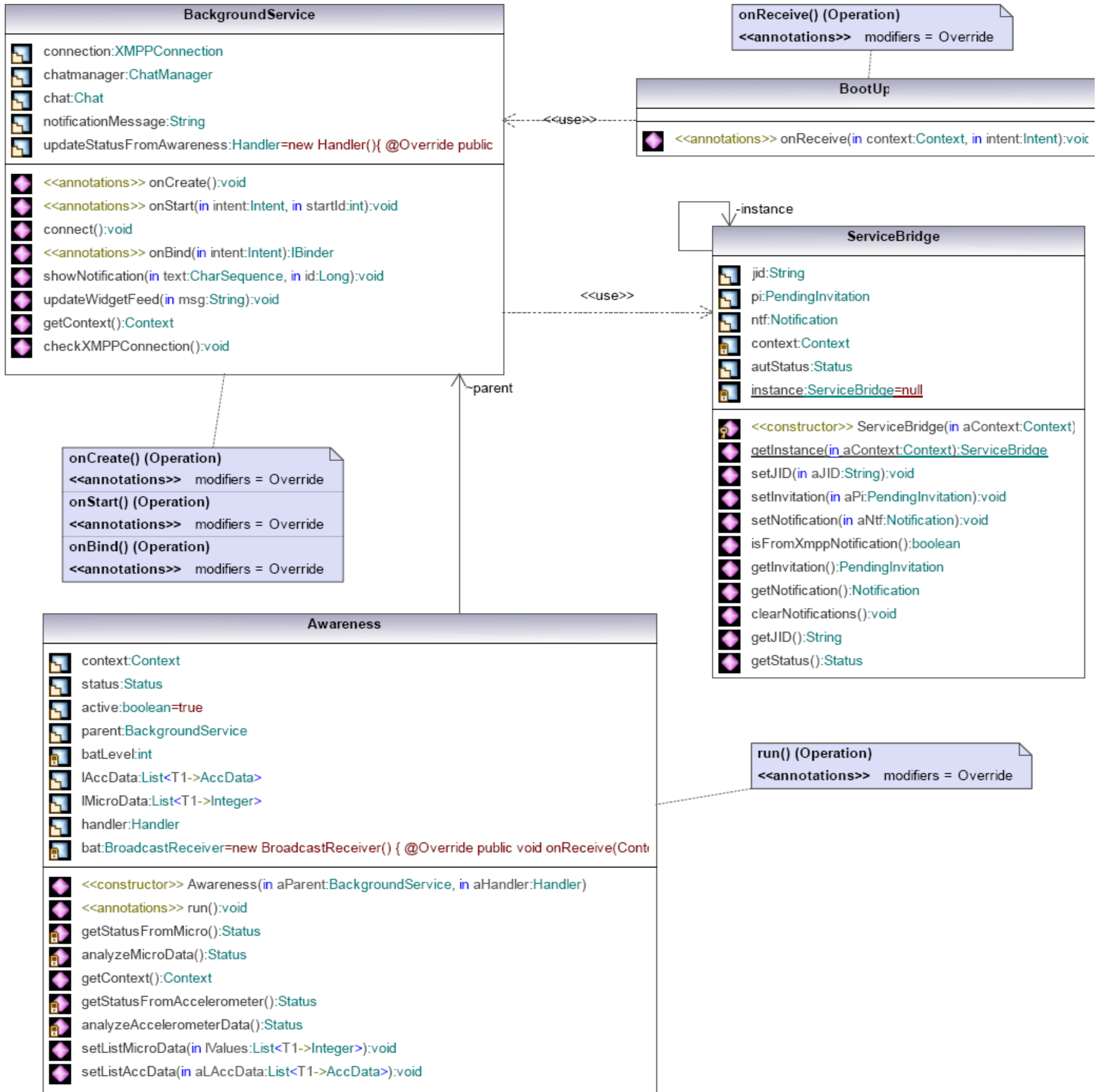
Contact



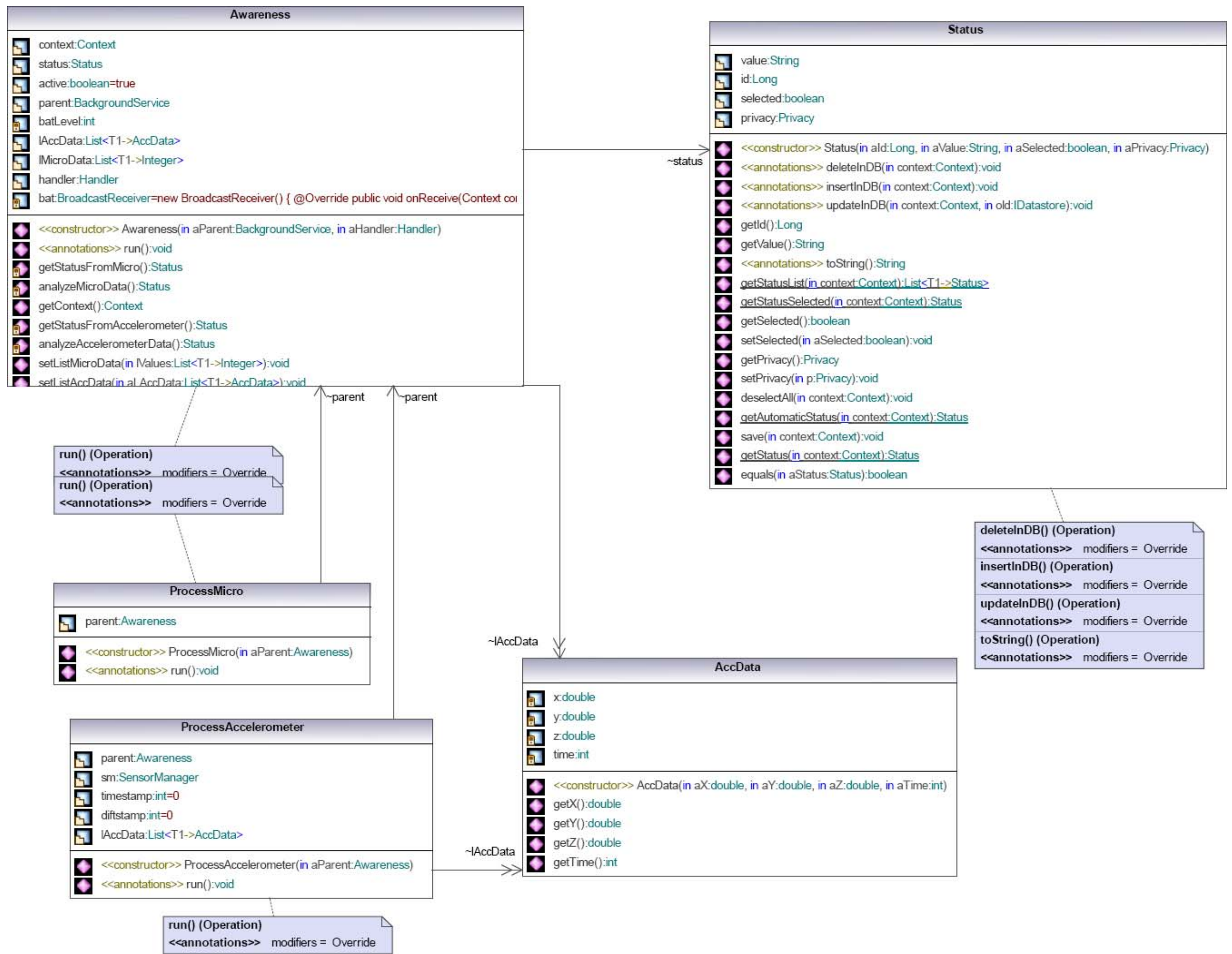
Notifications and Invitations



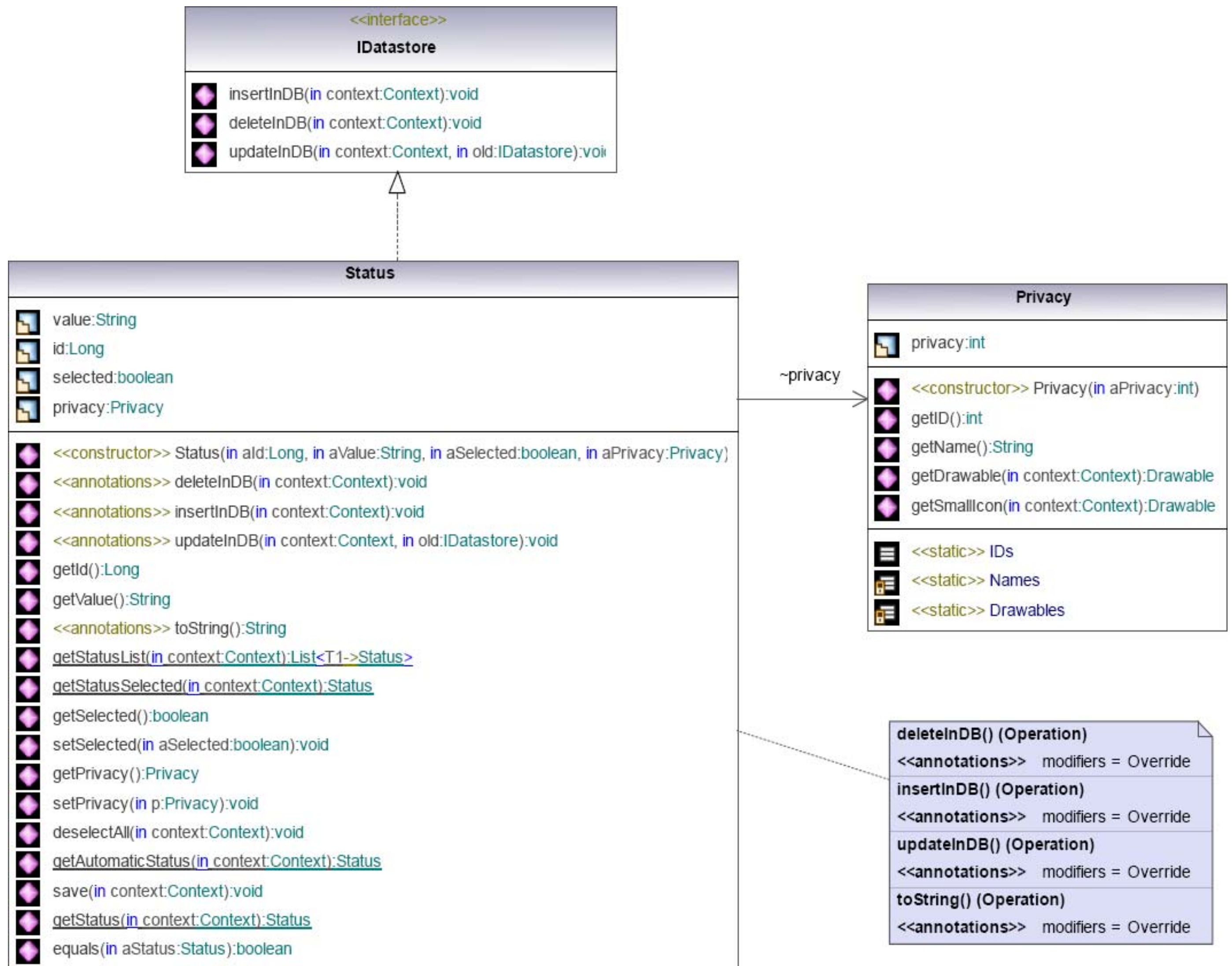
BackgroundService



Awareness



Status



APPENDIX E

Server Sequence Diagrams

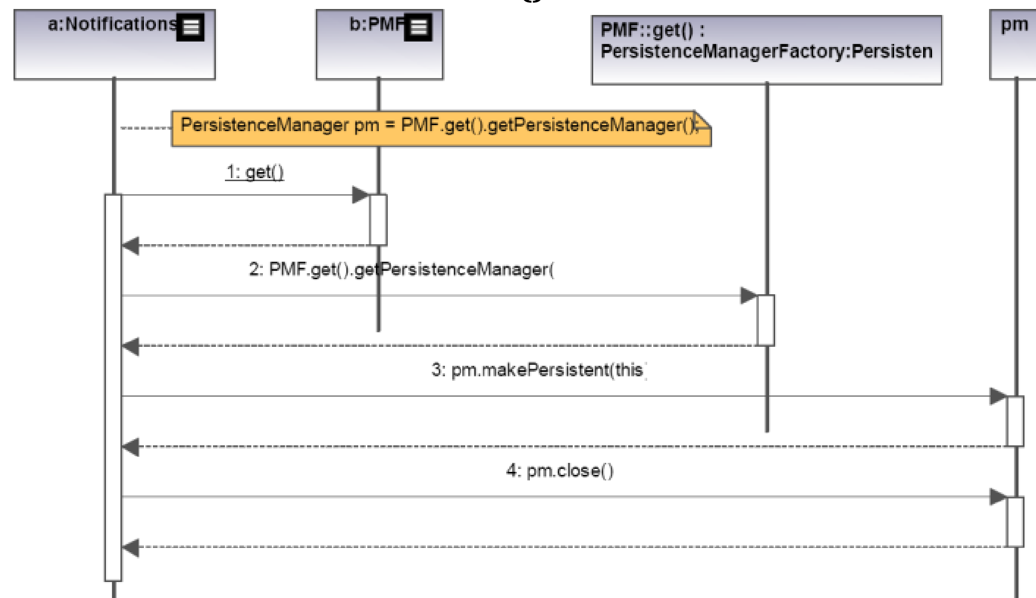
Contact – save()

Contact – delete()

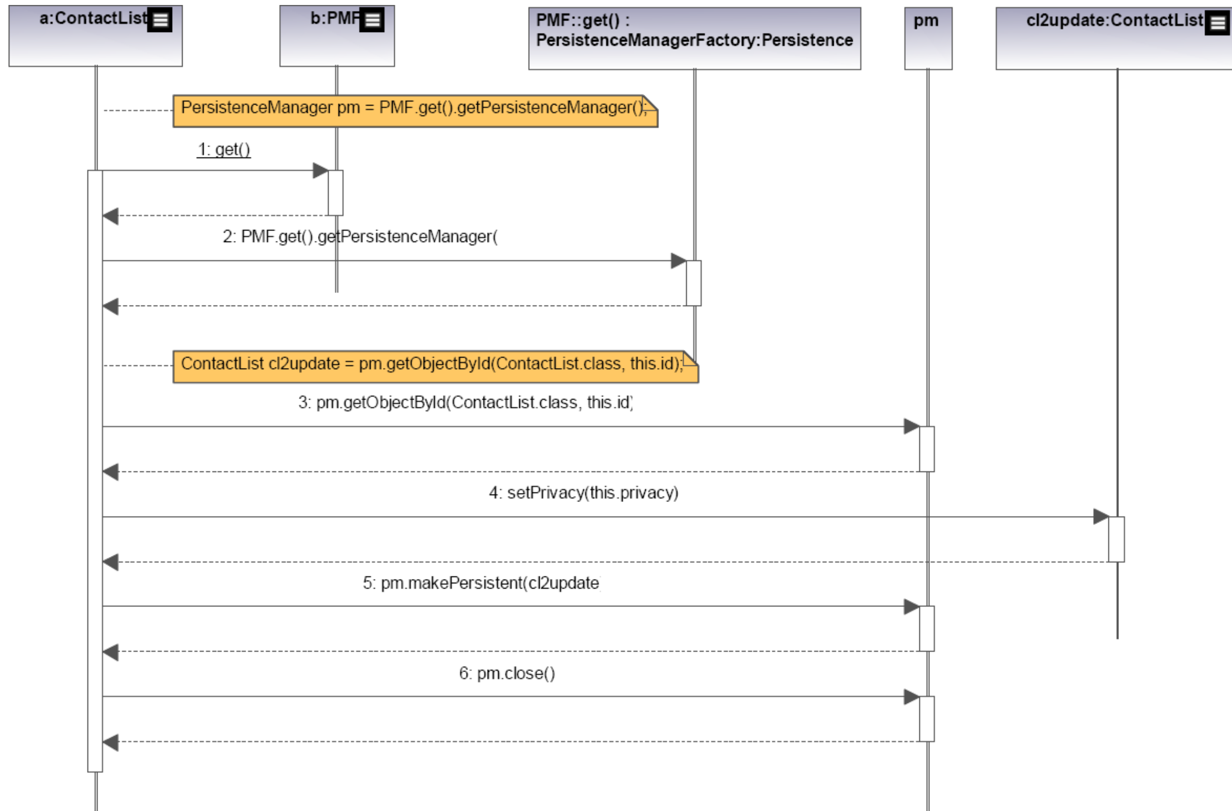
a:Contact
b:PMF
PMF::get() : PersistenceManagerFactory:Persist
pm
cd:ContactDetails
c:Certificate
c|0:Contact
d:Authentication
e:ContactList
f:PendingInvitations

PersistenceManager pm = PMF.get().getPersistenceManager();
1: get()
2: PMF.get().getPersistenceManager(
Contact c = pm.getObjectById(Contact.class, this.id);
3: pm.getObjectById(Contact.class, this.id)
4: pm.deletePersistent(c)
loop [ContactDetails cd : this.listContactDetails]
5: delete()
Certificate cert = Certificate.getCertificate(this.id);
6: getCertificate(this.id)
7: delete()
8: getID()
9: delete(c.getID())
10: getID()
11: delete(c.getID())
12: getID()
13: delete(c.getID())
14: pm.close()

Notification – createNotification()



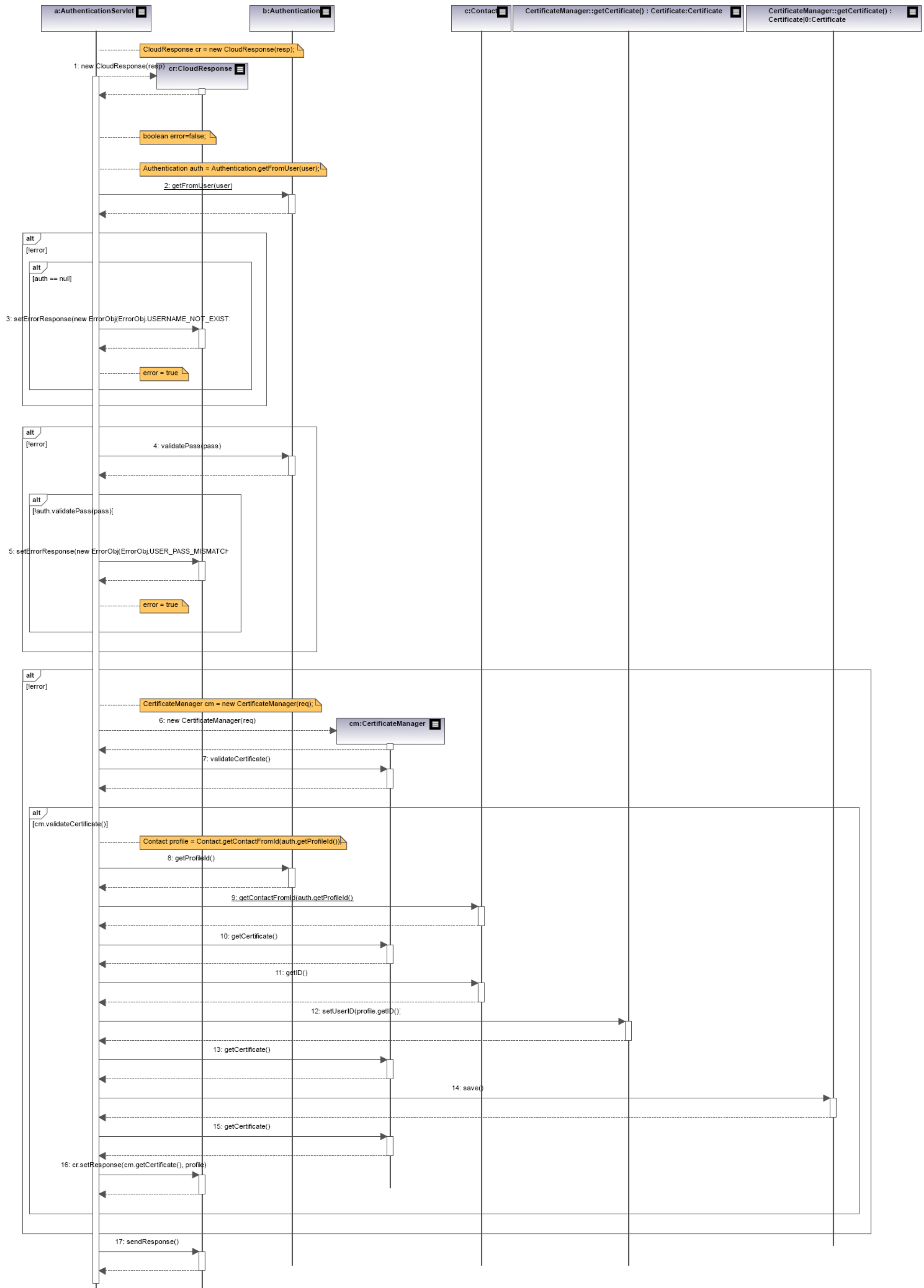
ContactList - updatePrivacyInContactList()



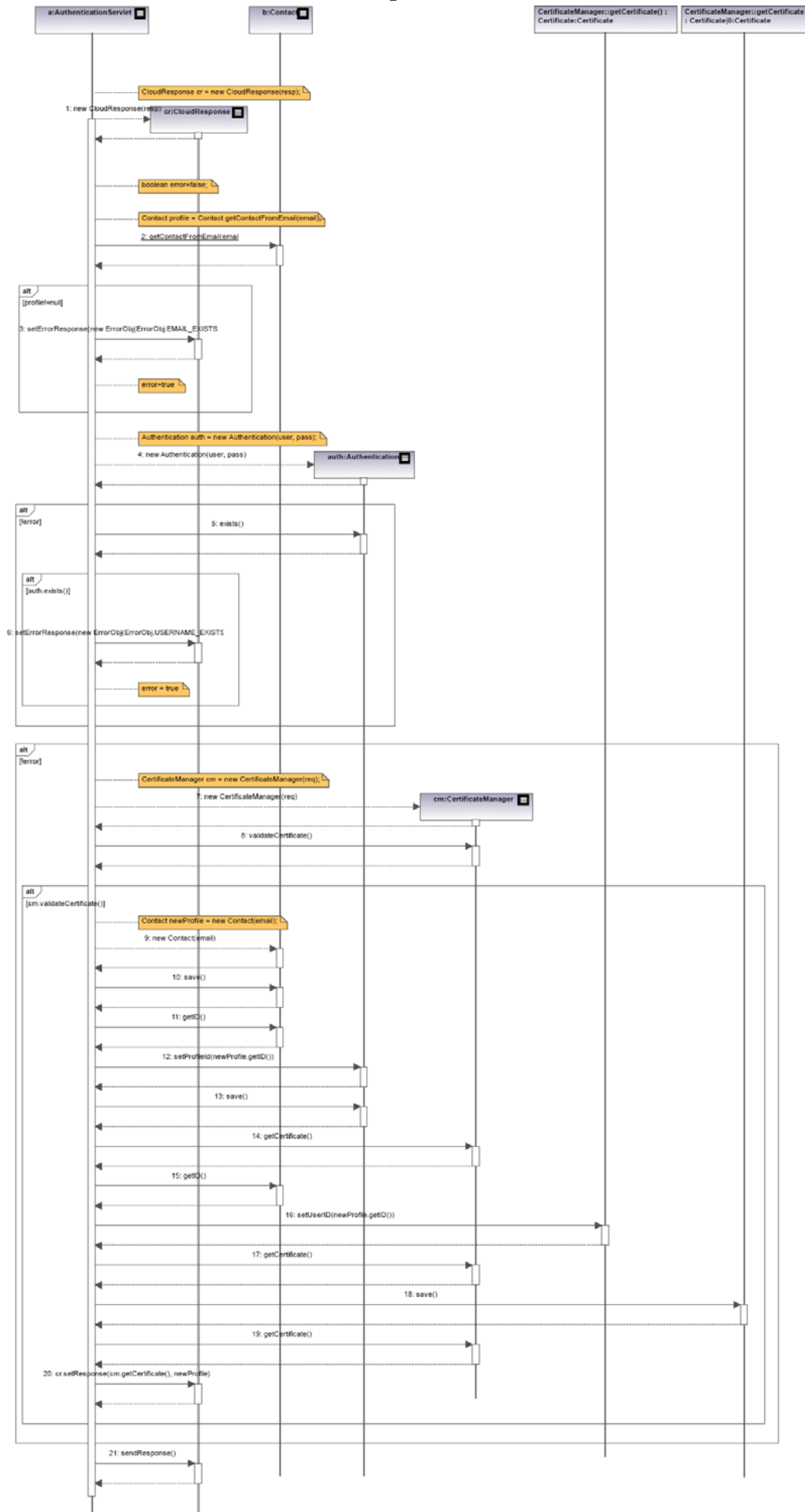
ContactList – getList()

a:ContactList
b:PMF
PMF::get() :
PersistenceManagerFactory:Persist
ContactList.class
pm
pm.newQuery(query)
PersistenceManager pm = PMF.get().getPersistenceManager();
1: get()
2: PMF.get().getPersistenceManager(
String query = "select from " + ContactList.class.getName() + " where ownerID == " + ownerID;
3: ContactList.class.getName()
List<ContactList> lcl = (List<ContactList>) pm.newQuery(query).execute();
4: pm.newQuery(query)
5: pm.newQuery(query).execute()
ListContacts tlcl = new ListContacts();
6: new ListContacts()
tlcl:ListContacts
loop [ContactList cl : lcl]
7: tlcl.add(cl)
8: pm.close()
return tlcl

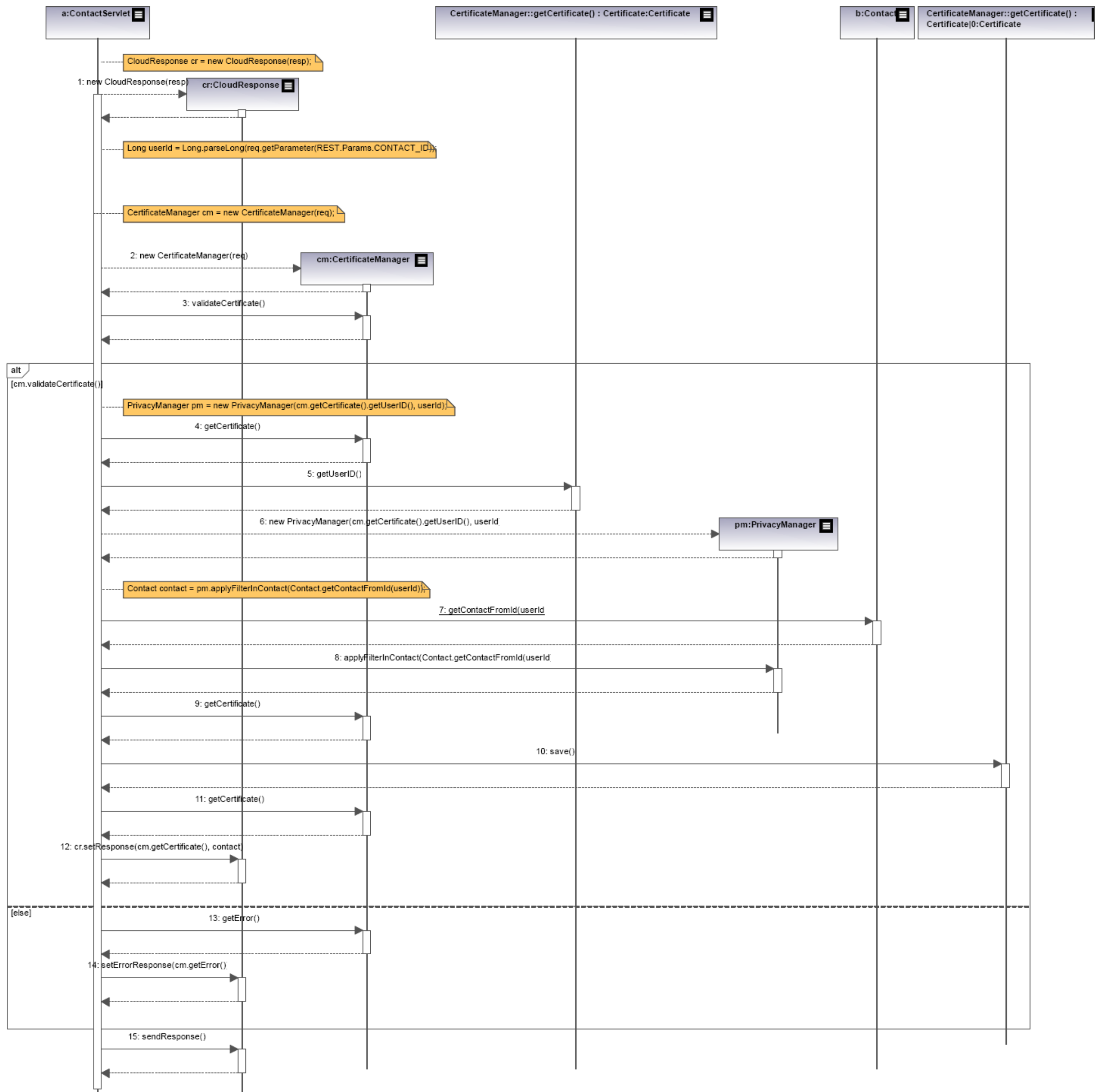
AuthenticationServlet GET Request



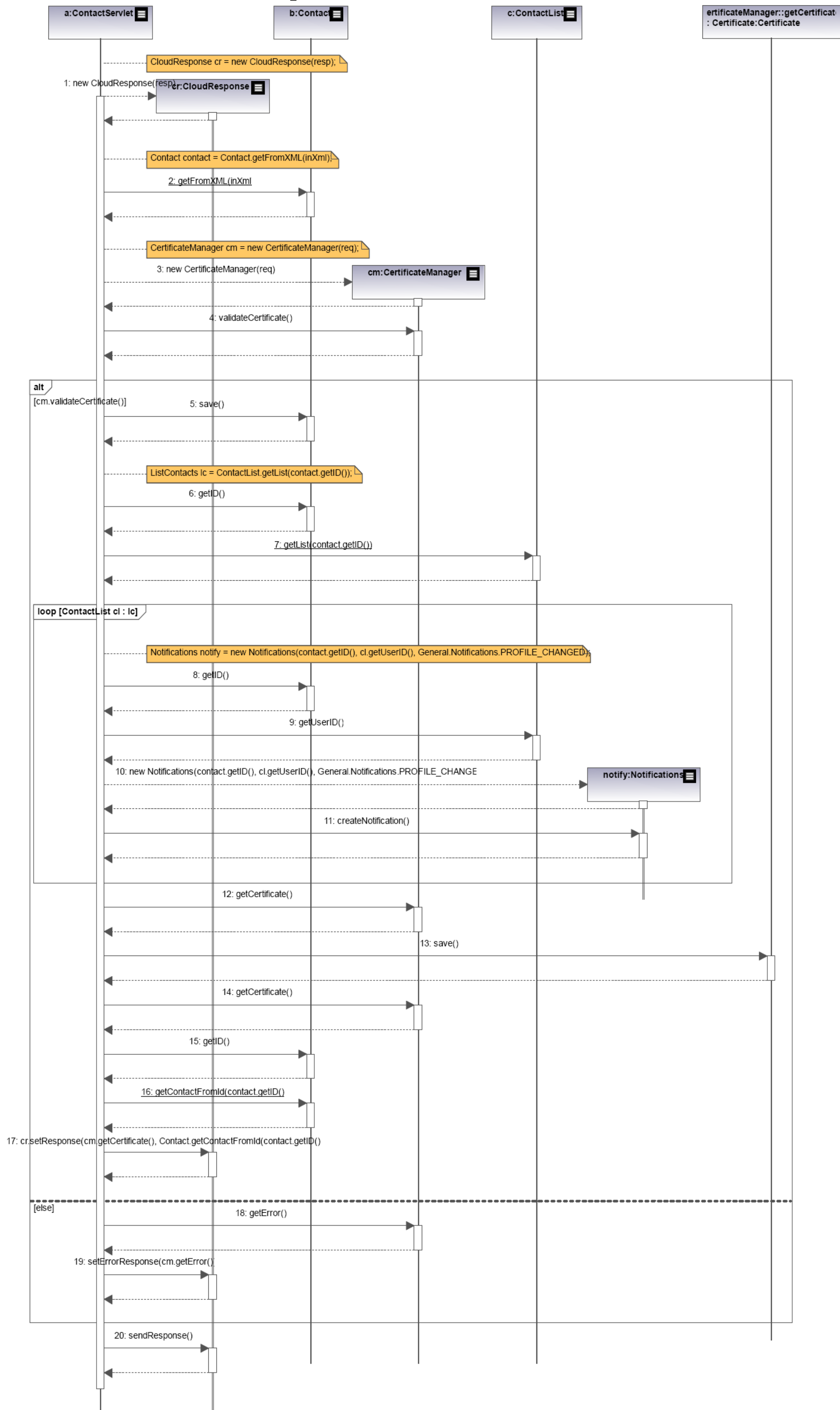
AuthenticationServlet PUT Request



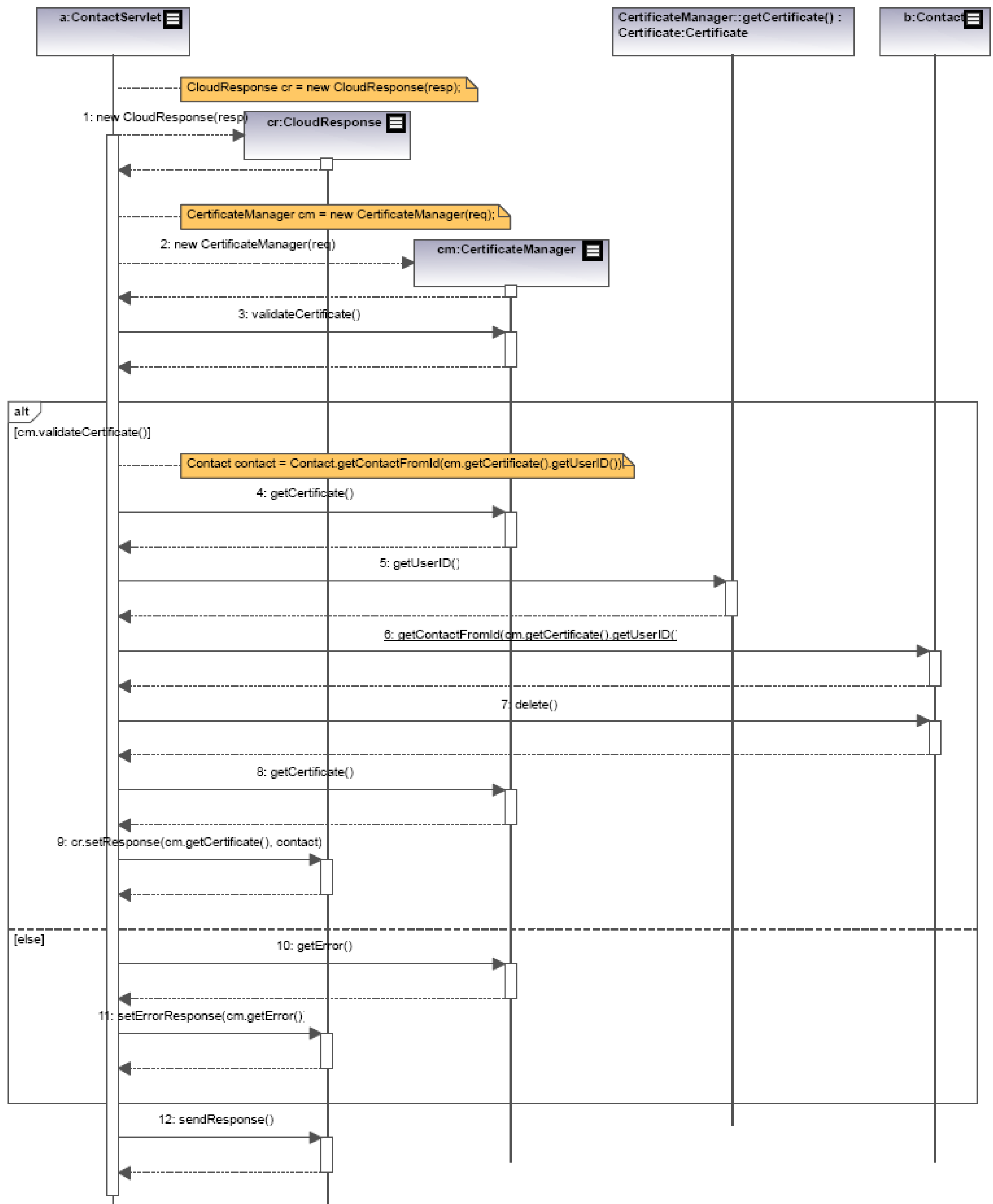
ContactServlet GET Request



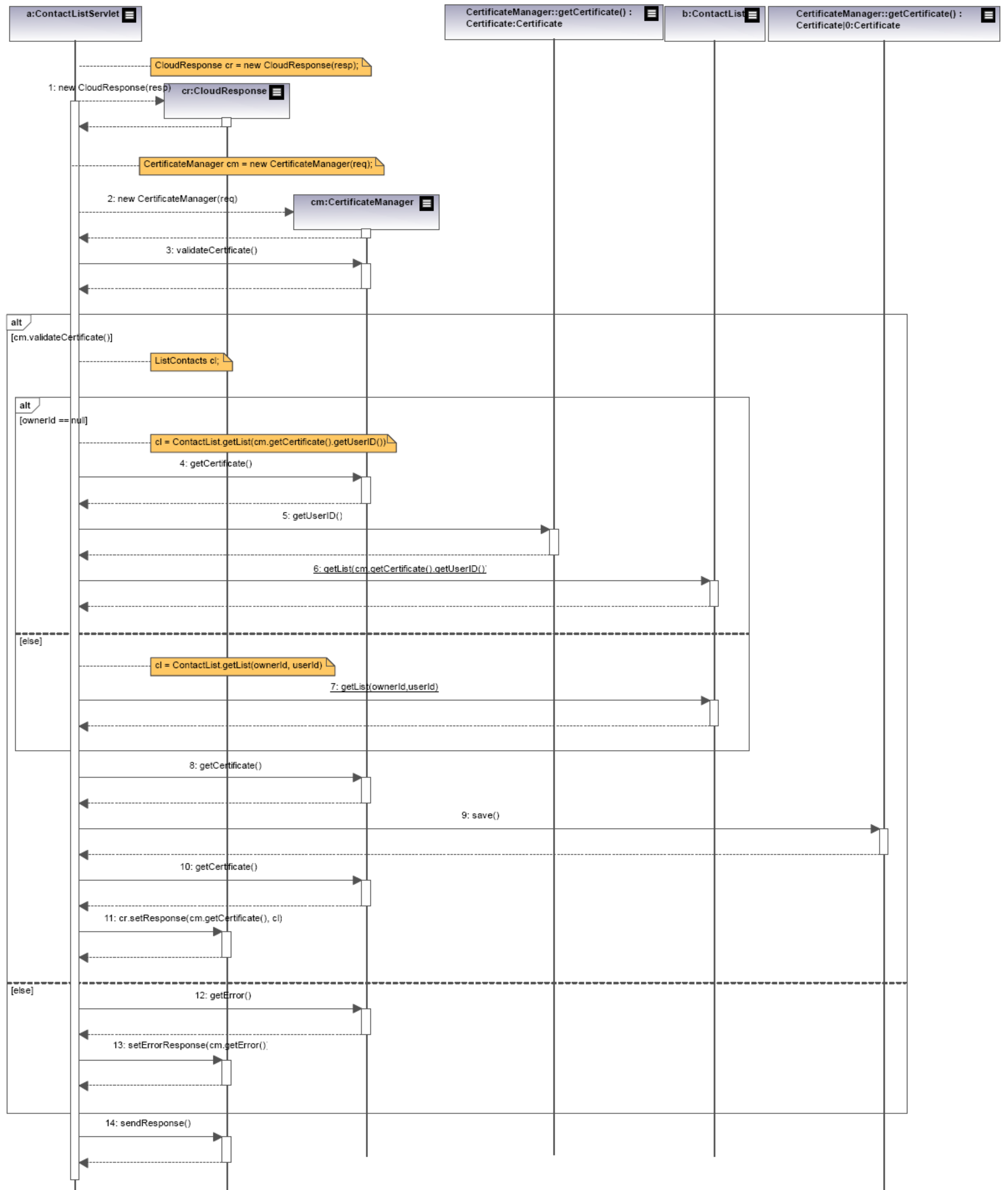
ContactServlet POST Request



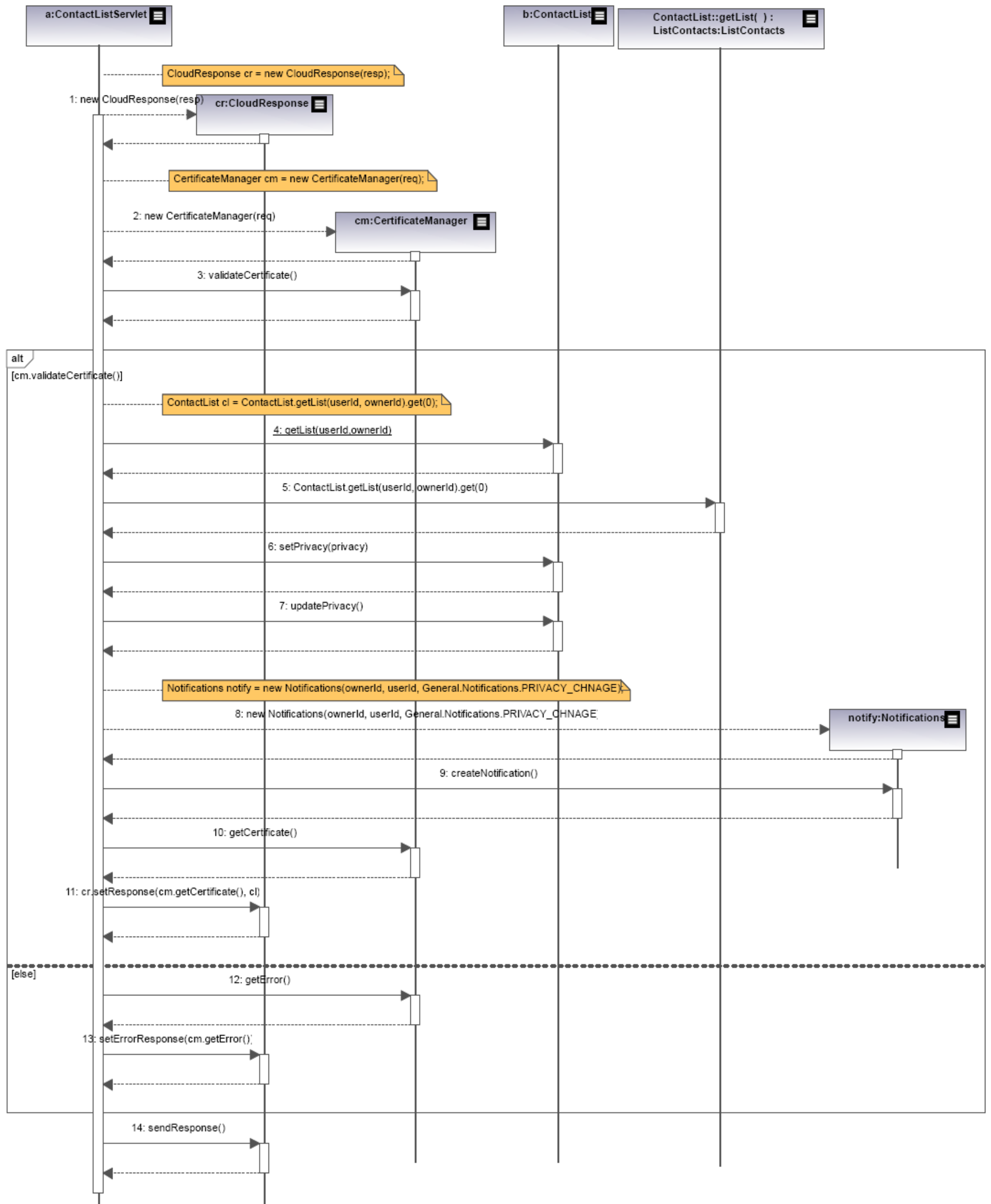
ContactServlet DELETE Request



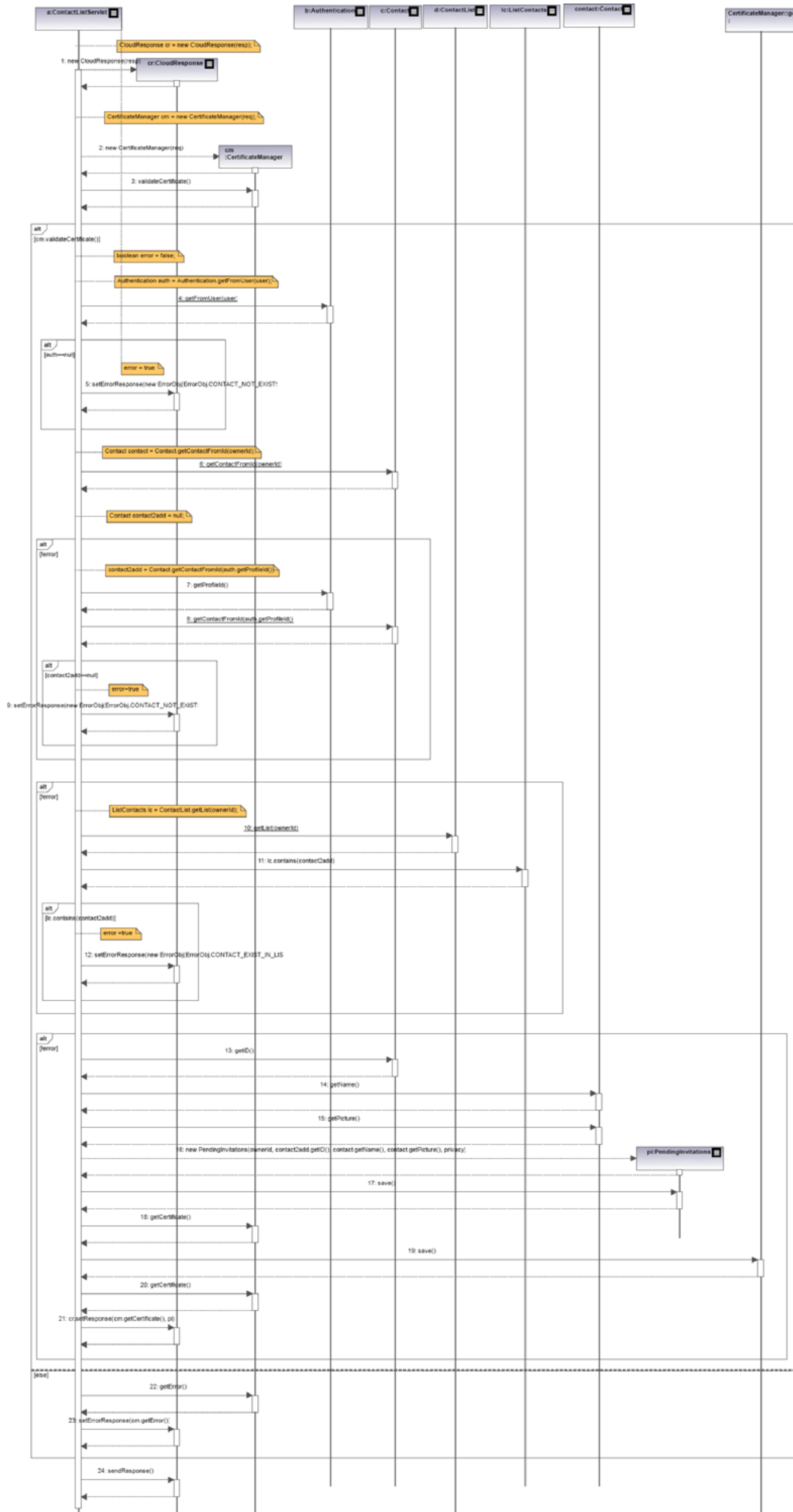
ContactListServlet GET Request



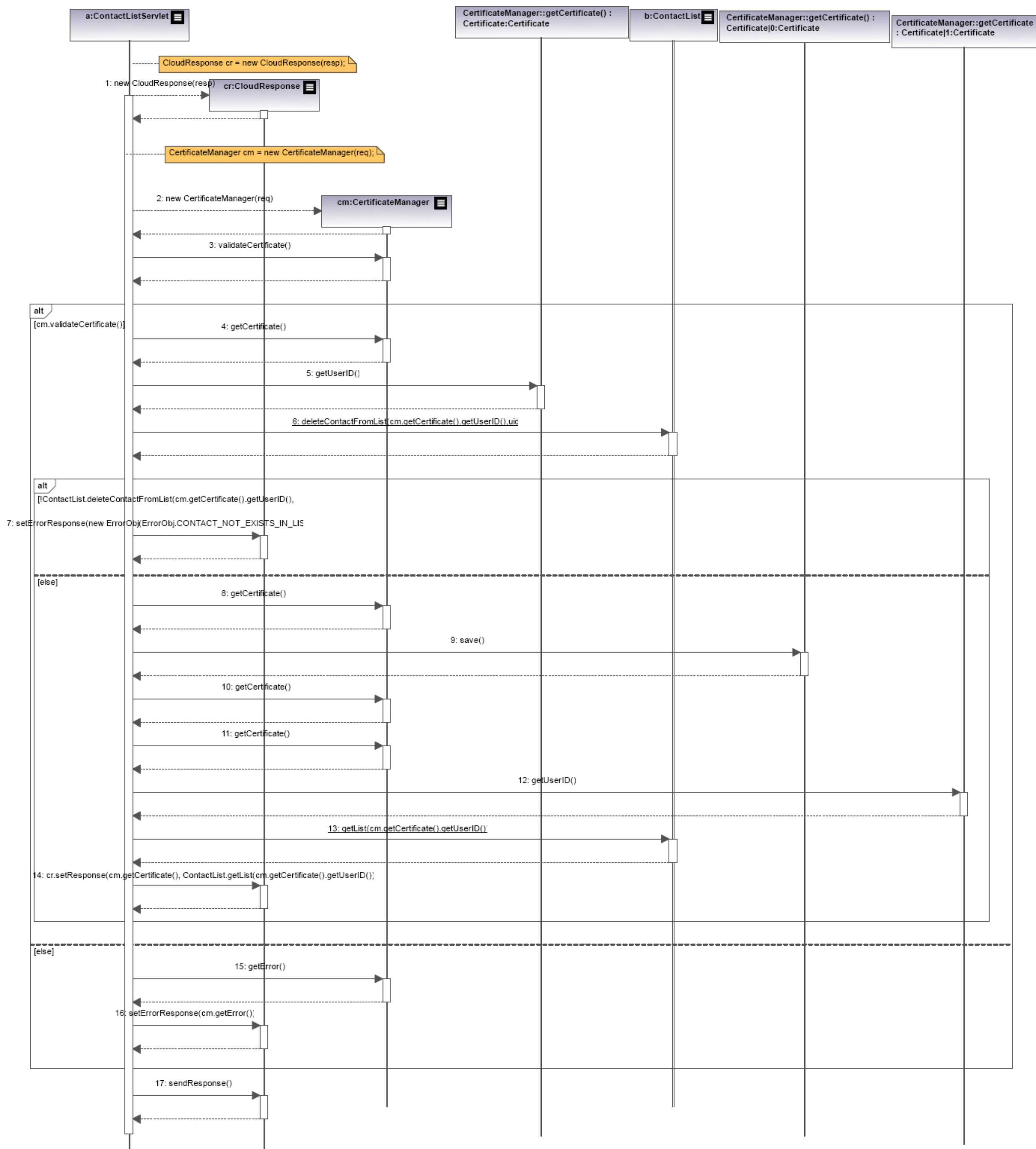
ContactListServlet POST Request



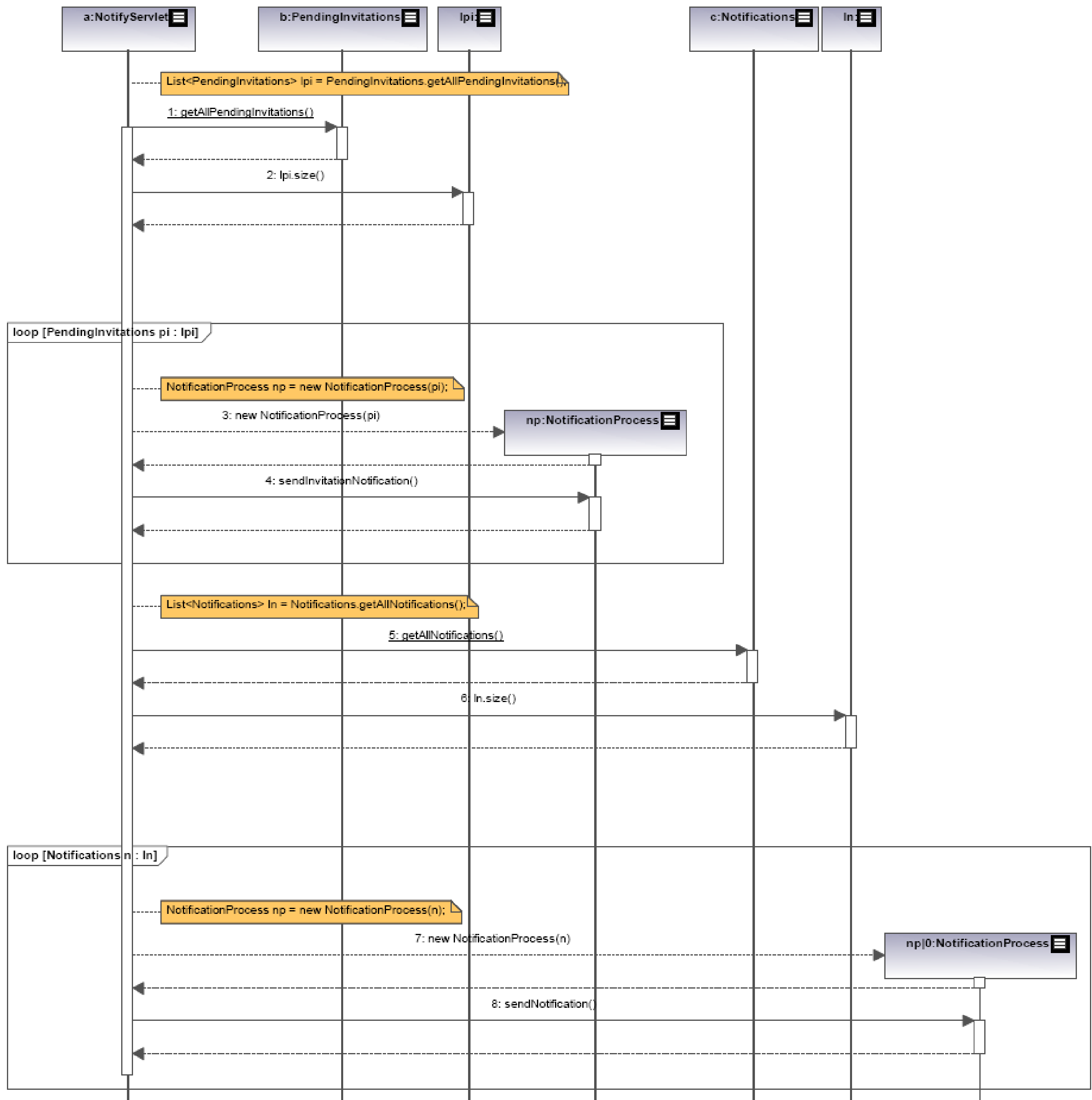
ContactListServlet PUT Request



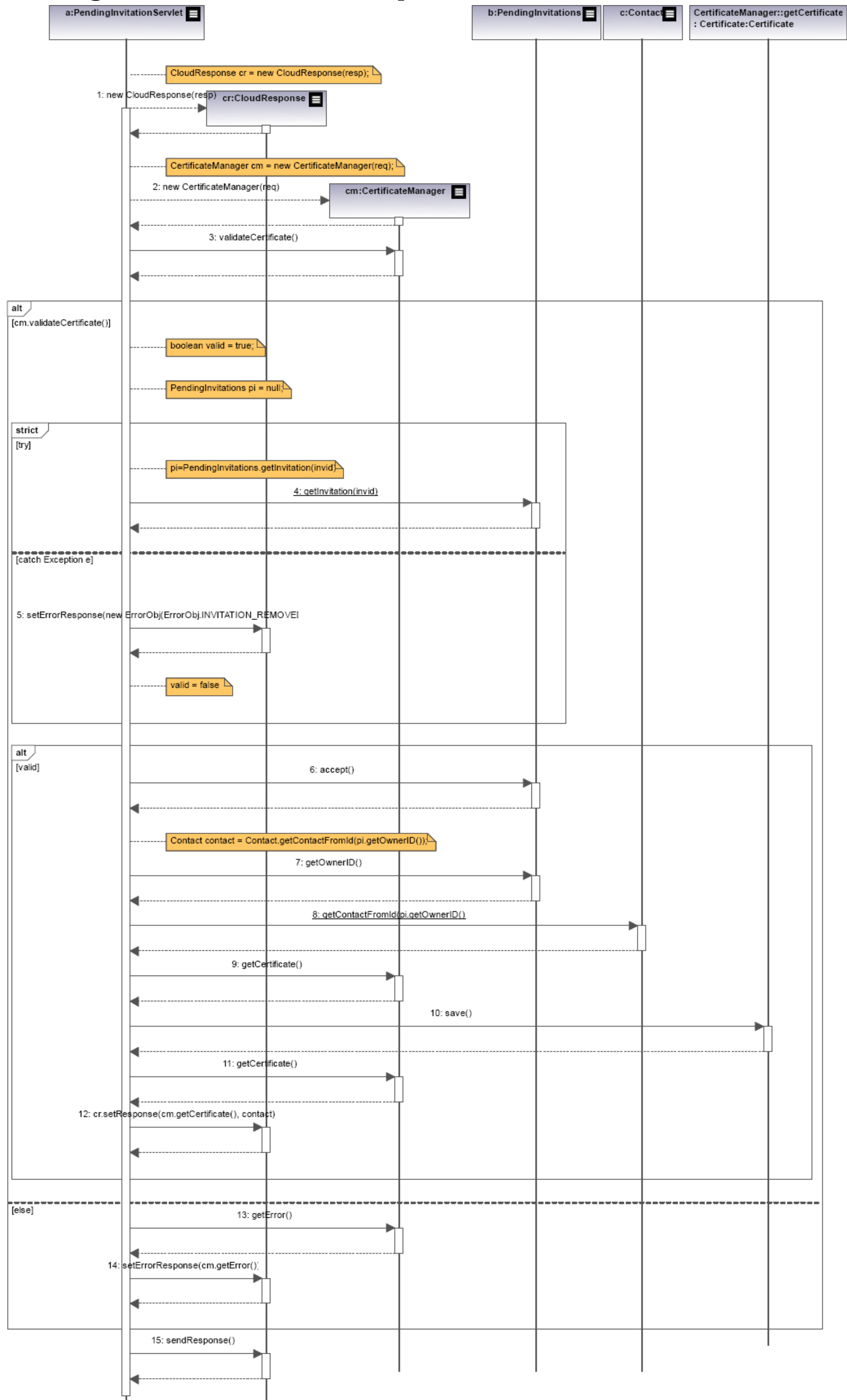
ContactListServlet DELETE Request



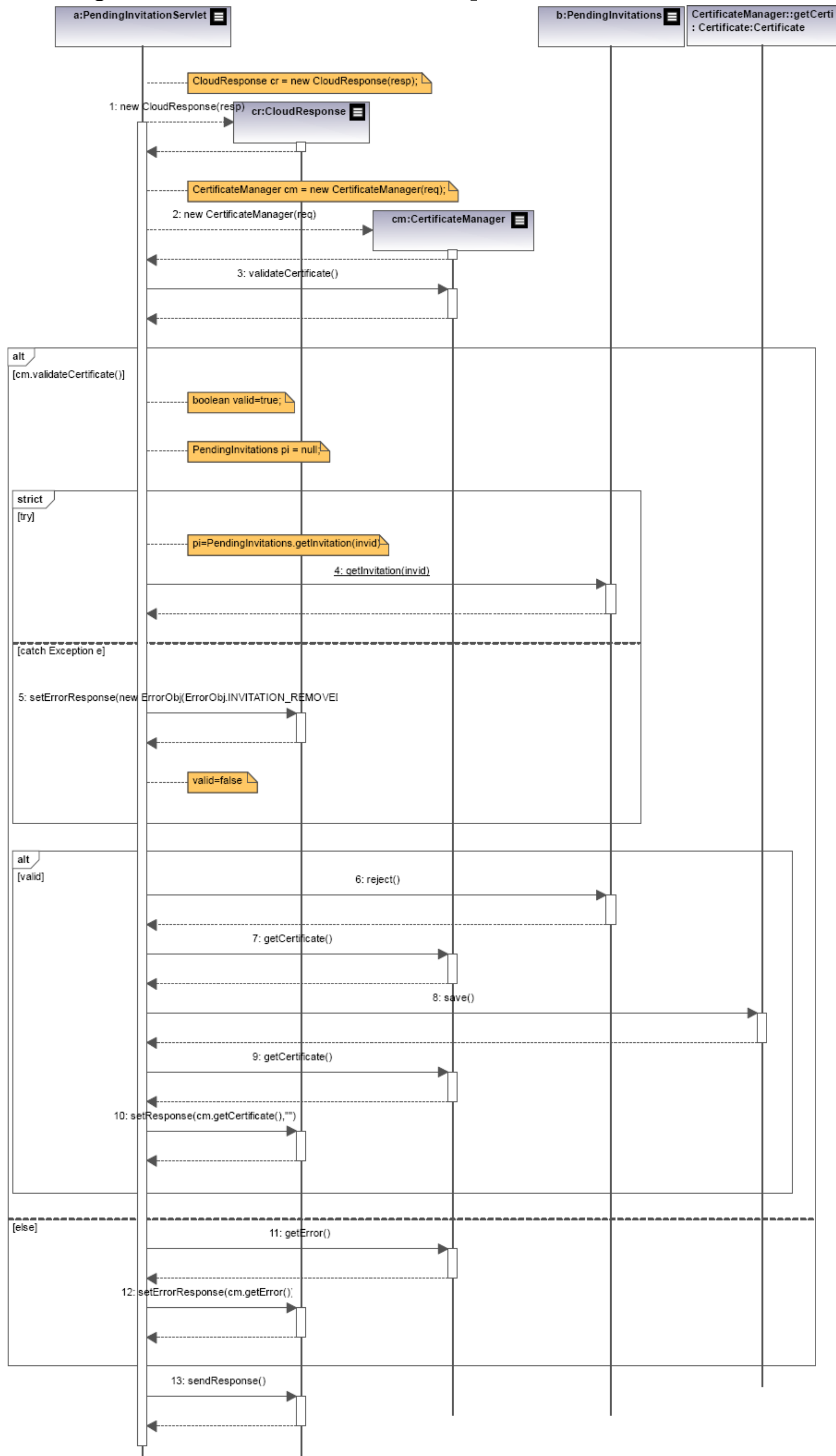
NotifyServlet GET Request



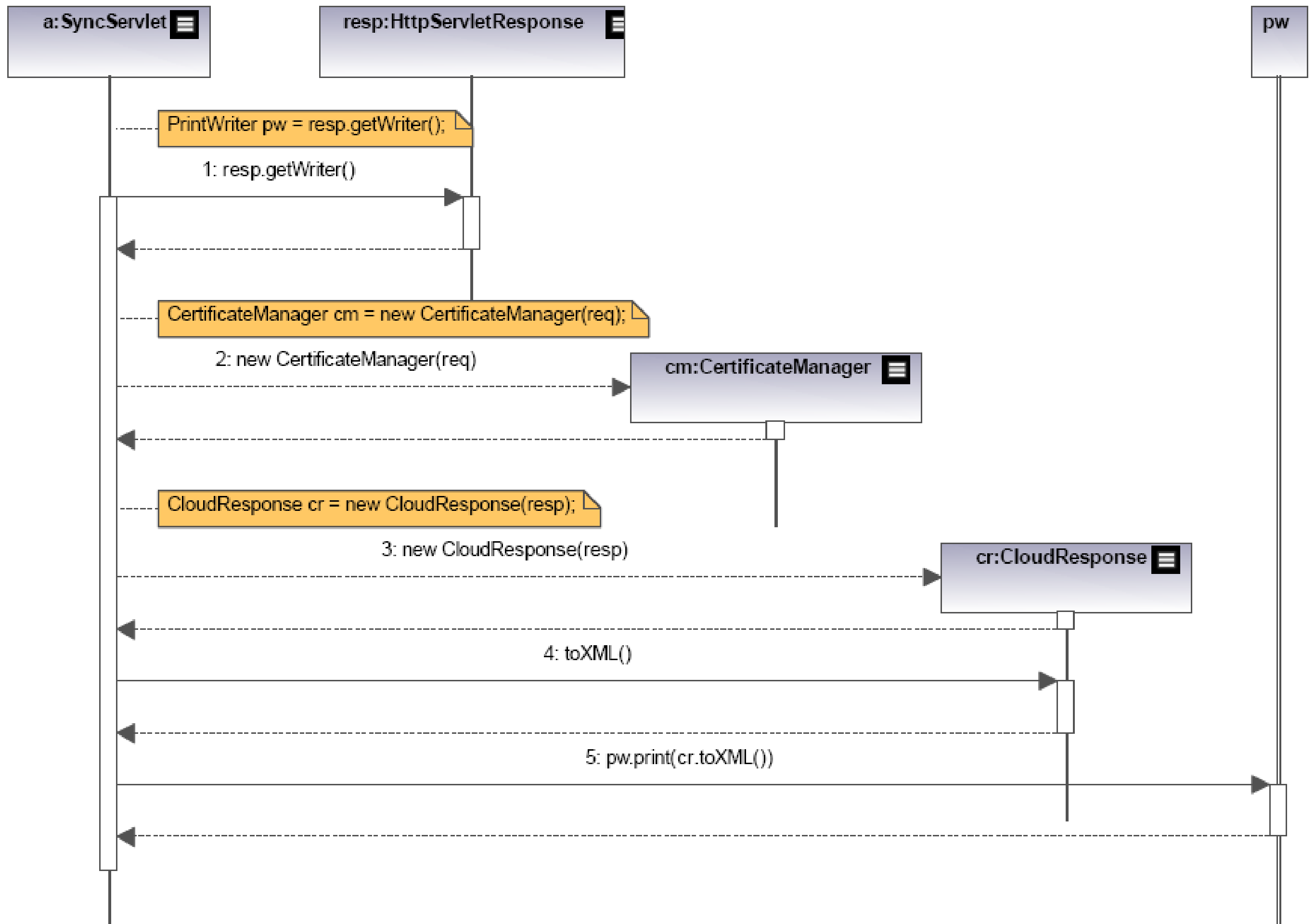
PendingInvitationServlet PUT Request



PendingInvitationsServlet DELETE Request



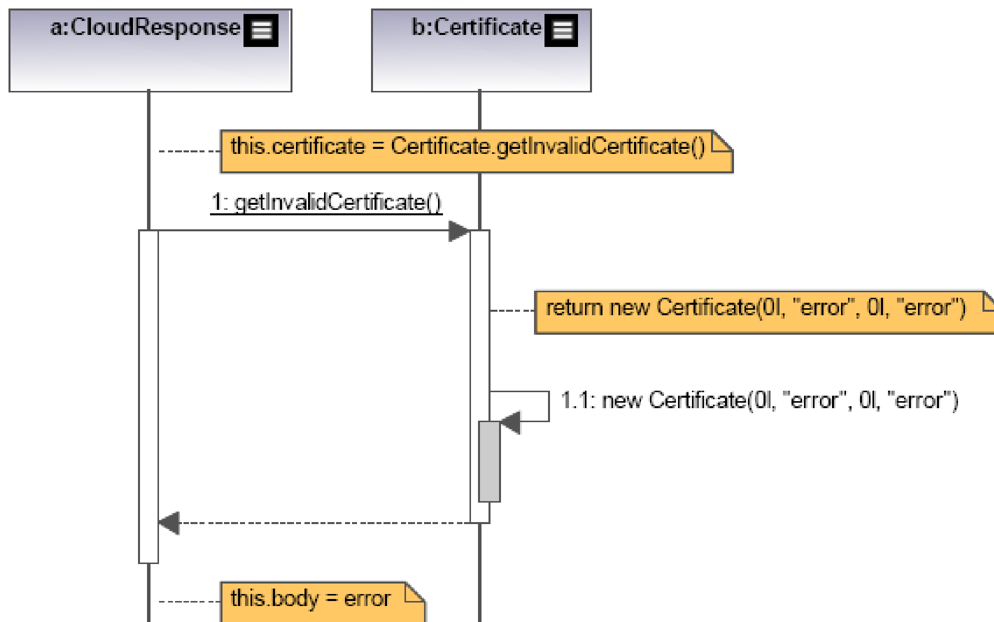
SyncServlet POST Request



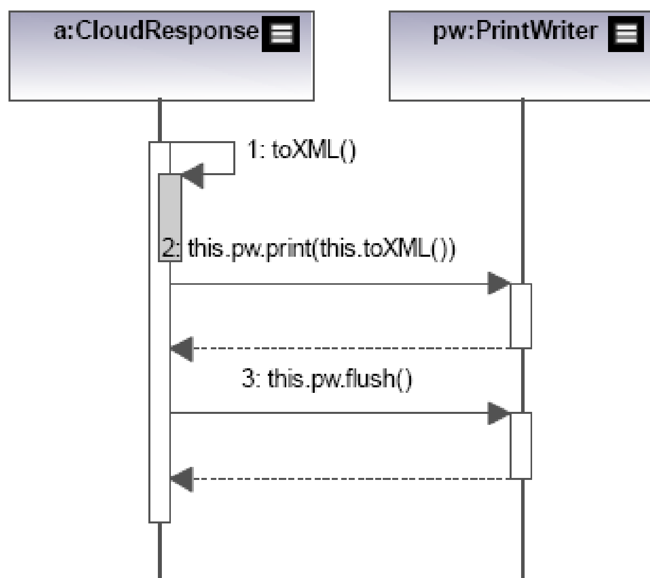
XMPPServlet POST Request

CloudResponse

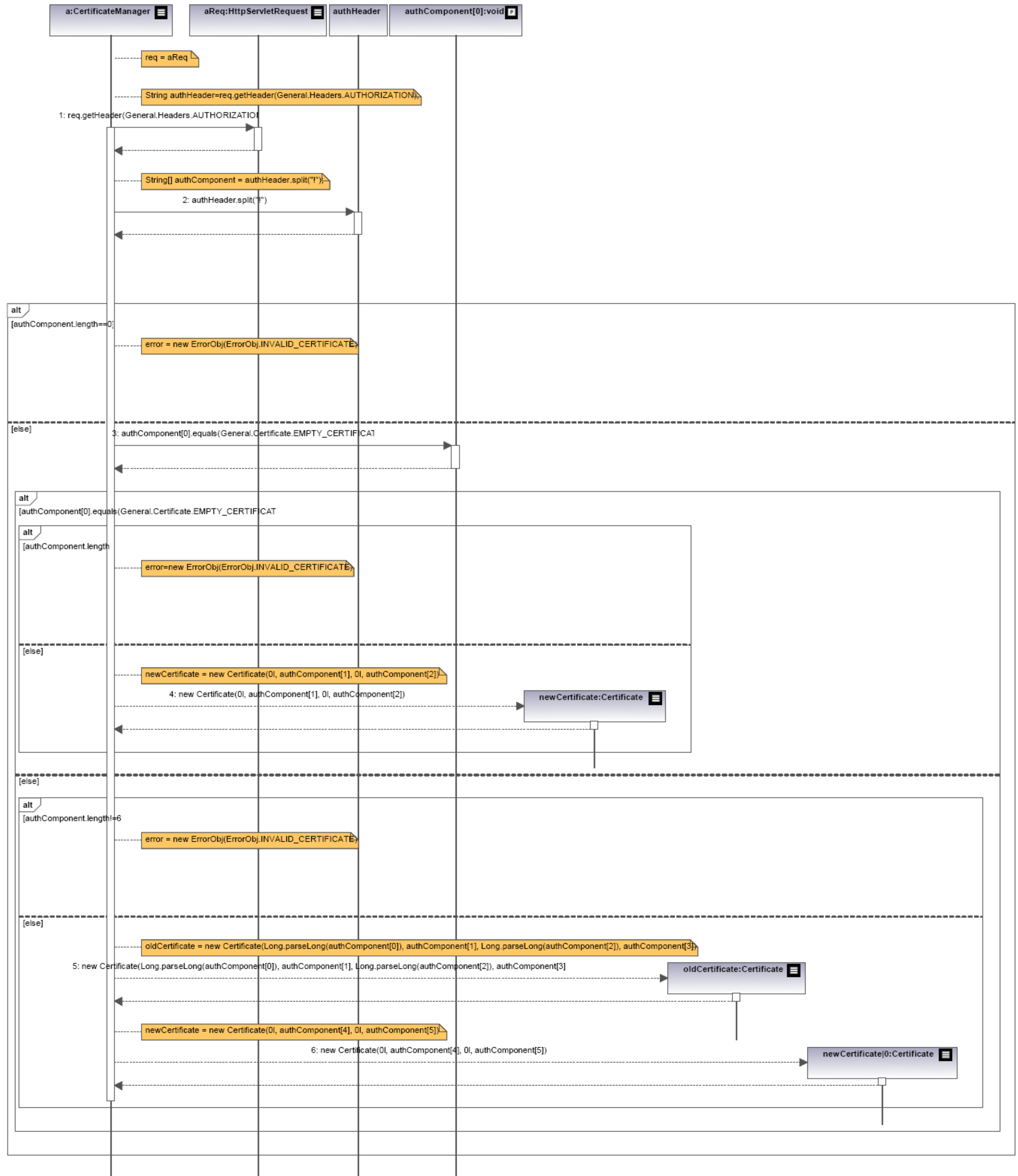
setErrorResponse()



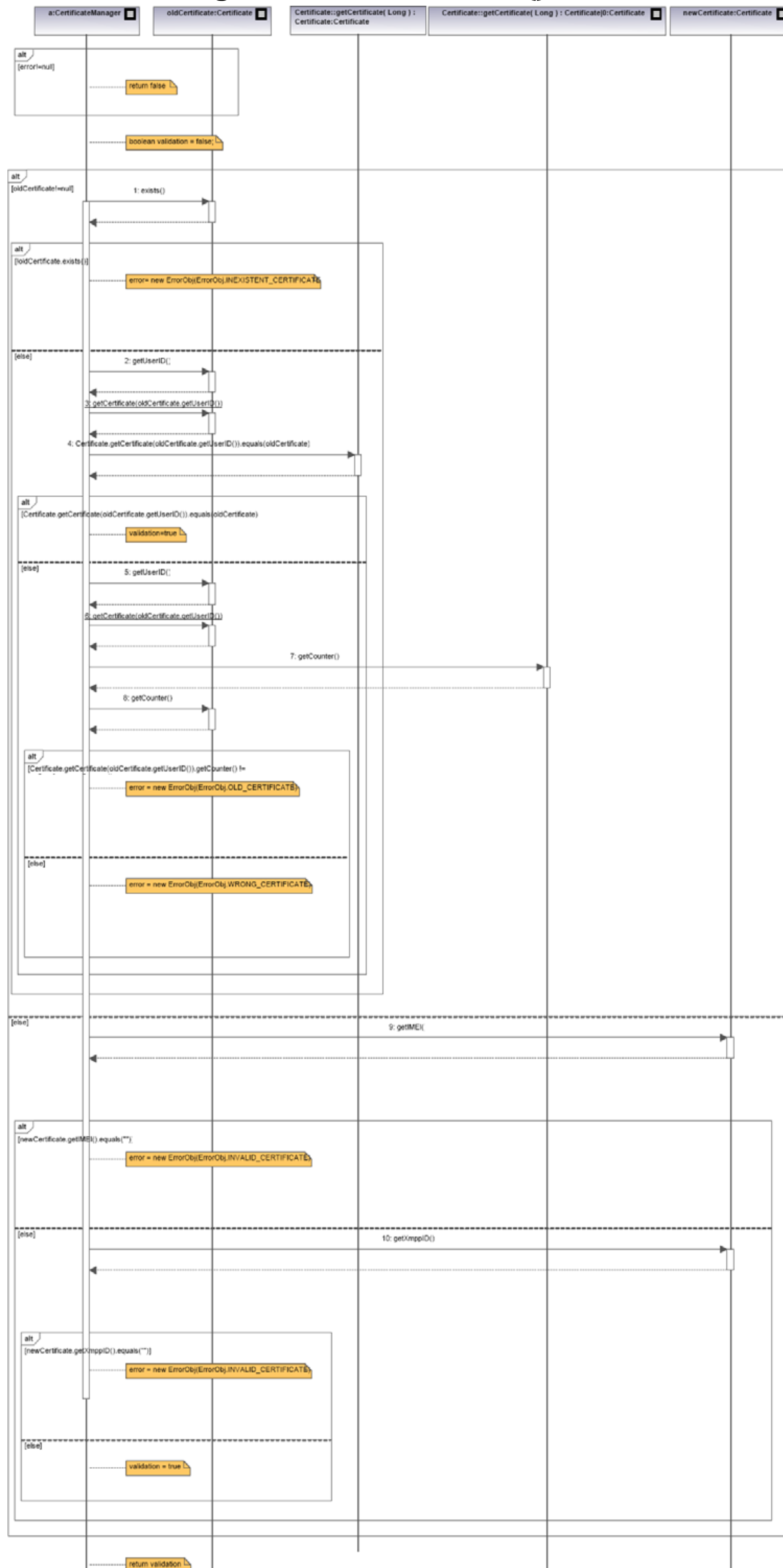
sendResponse()



CertificateManager – constructor

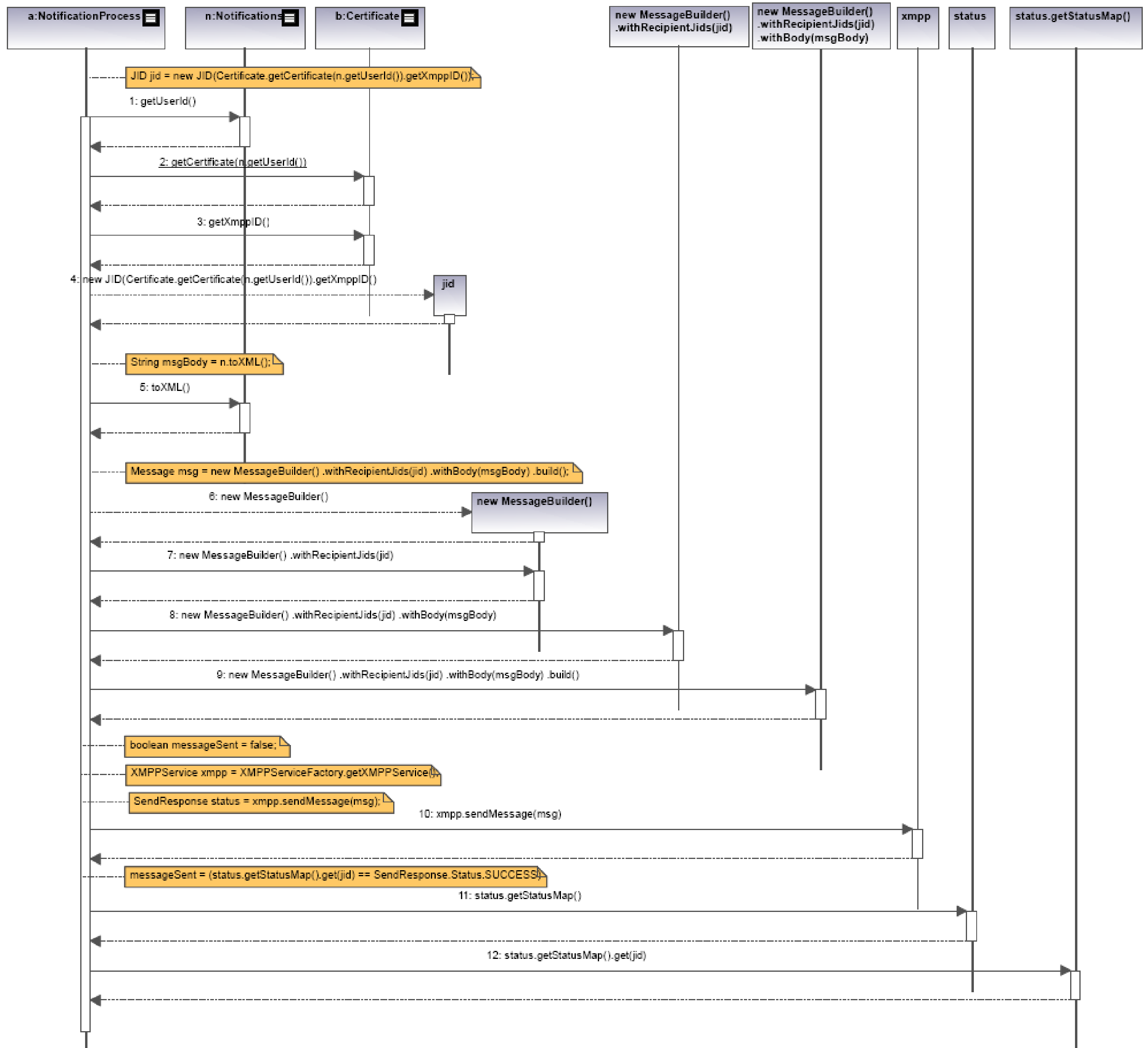


CertificateManager – validateCertificate()

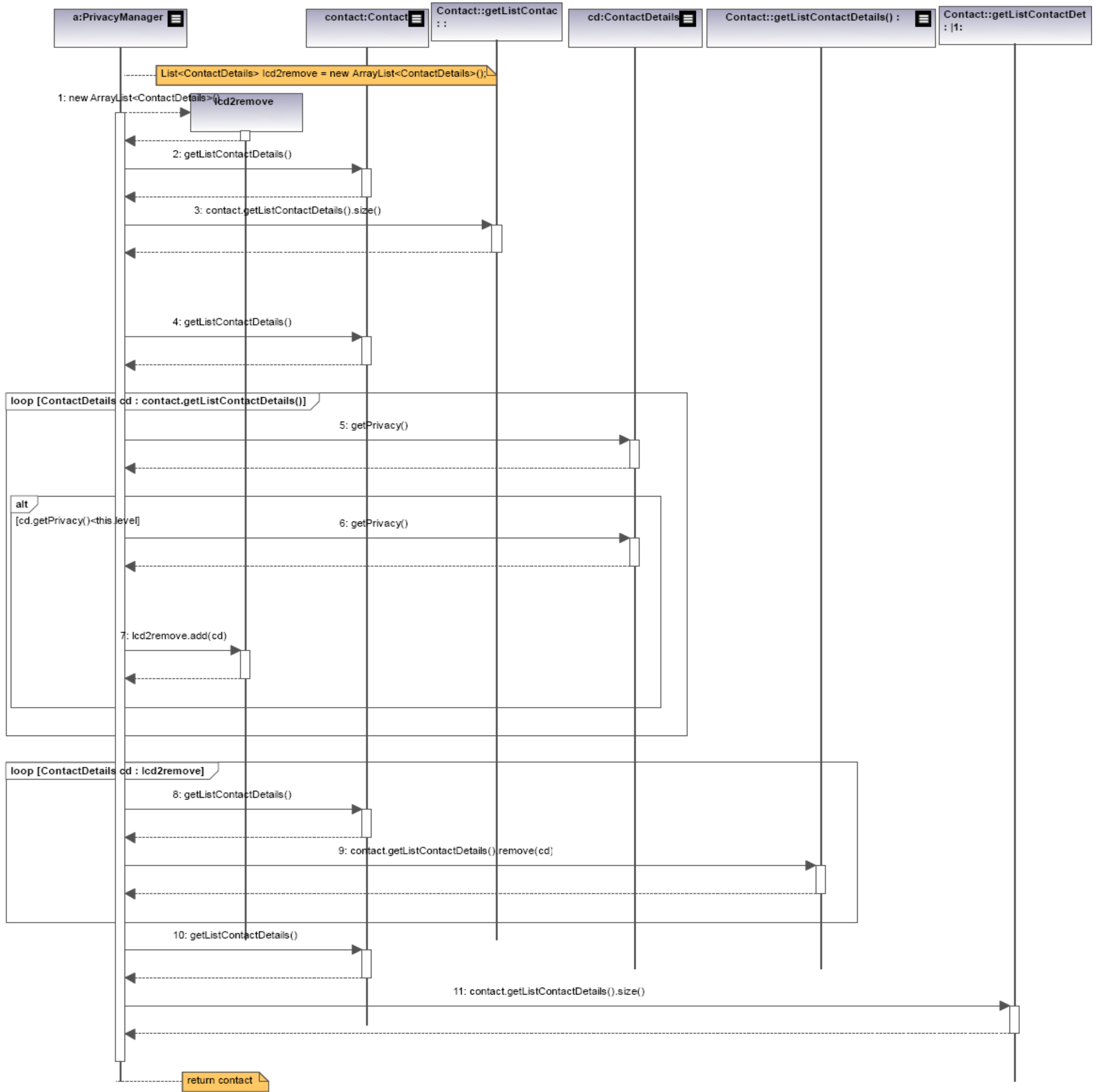


NotificationManager – sendInvitationNotification()

NotificationManager – sendNotification()



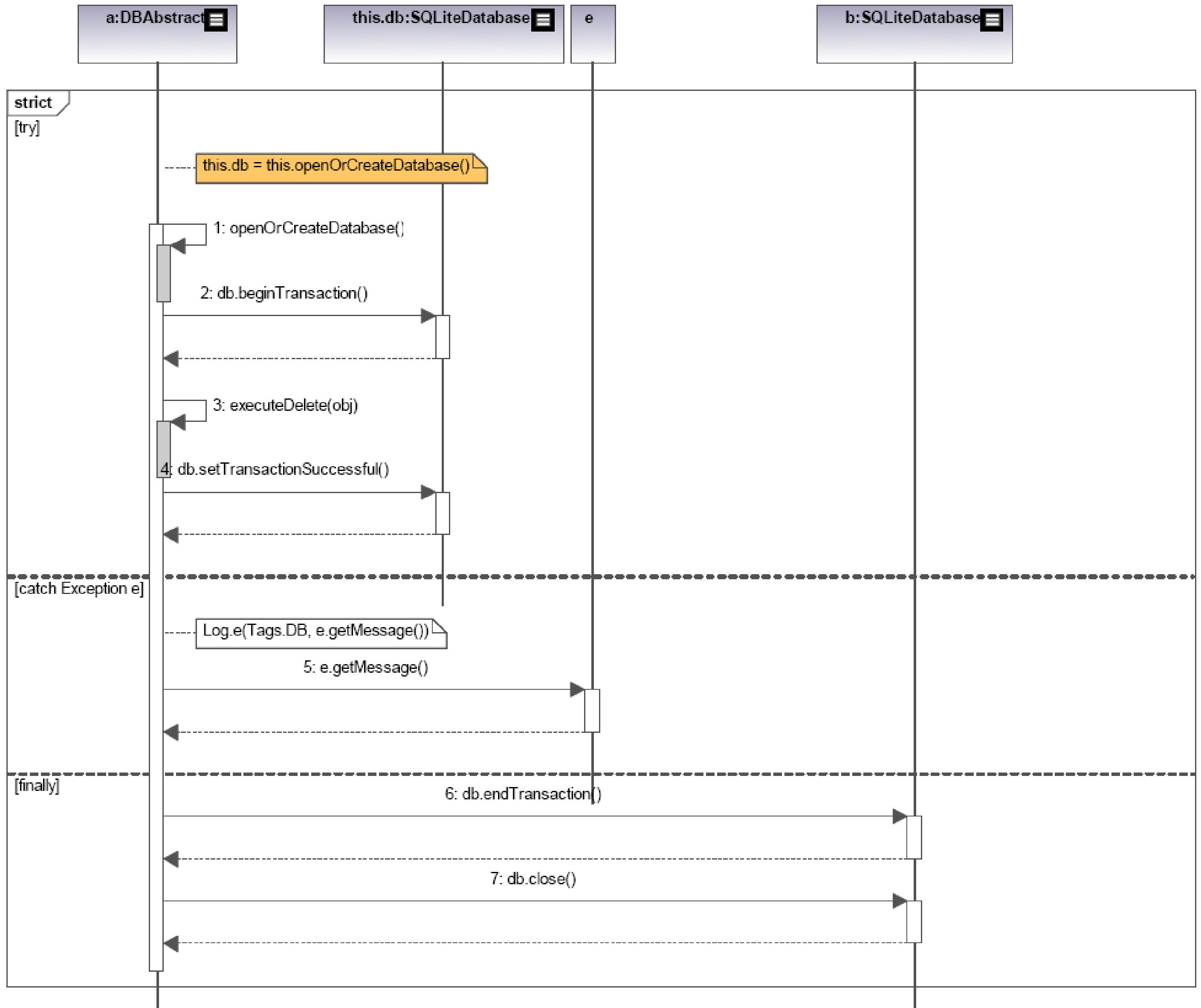
PrivacyManager – applyFilterInContact



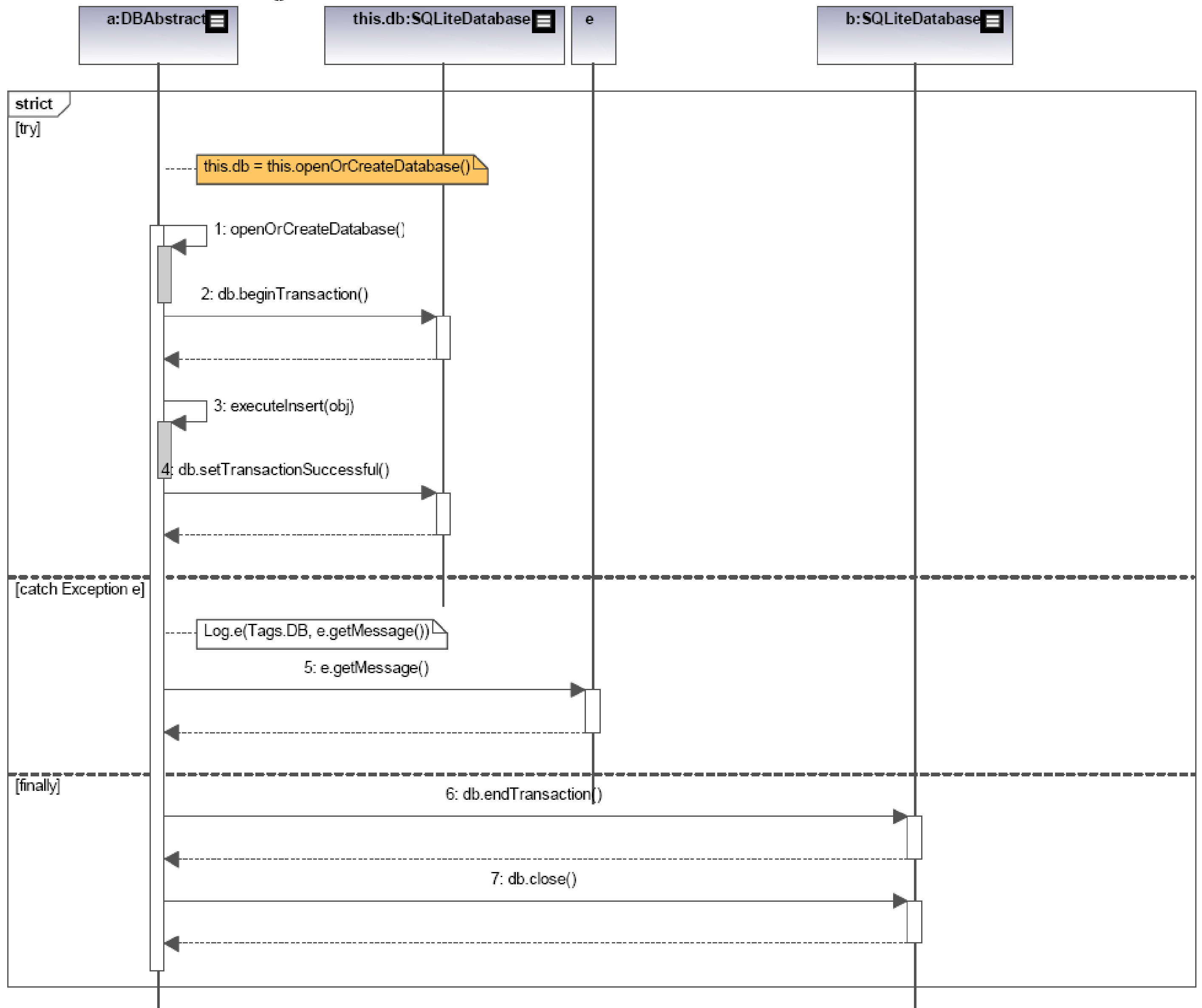
APPENDIX F

Client Sequence Diagrams

DBAbstract – delete()



DBAbstract – insert()

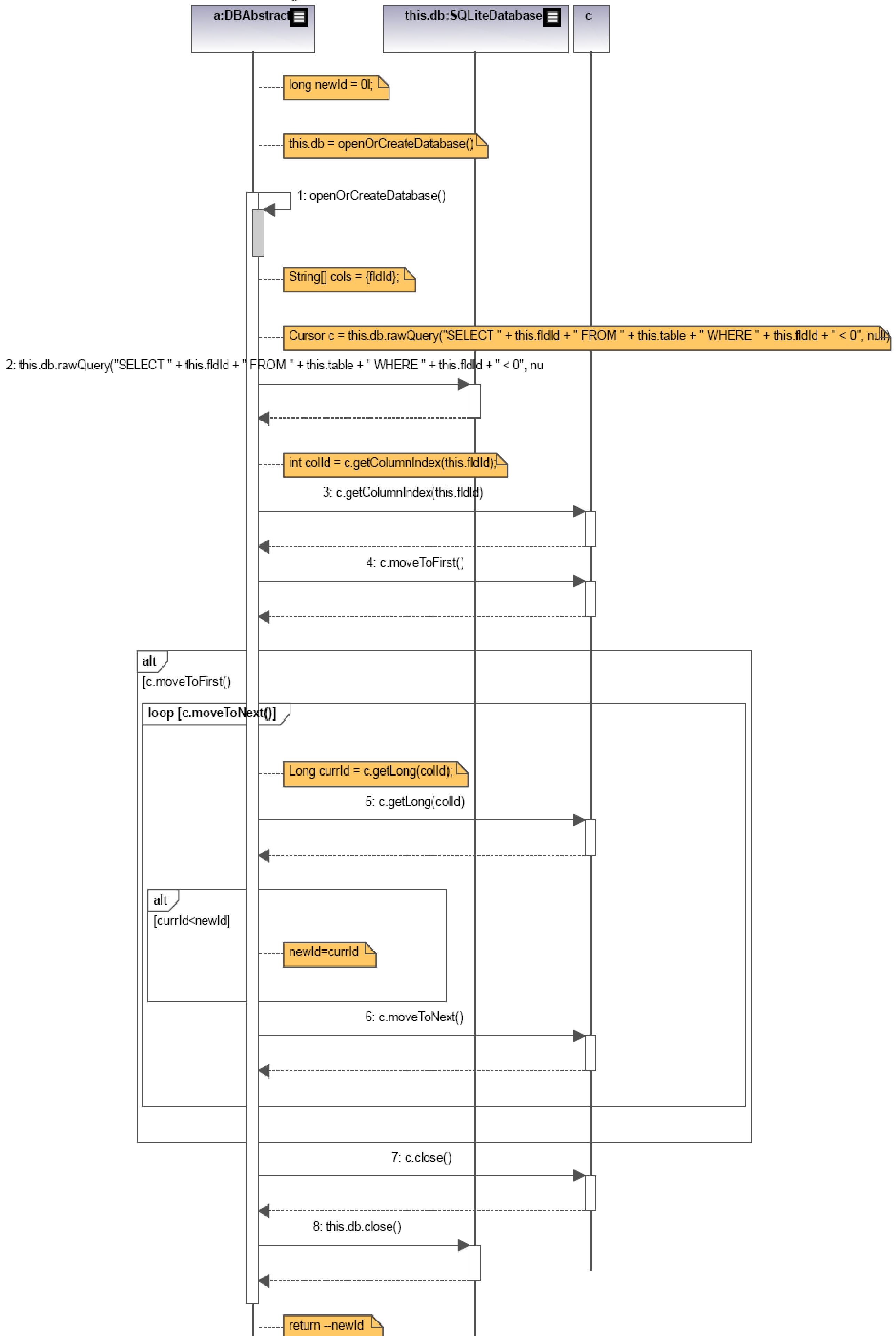


DBAbstract – update()

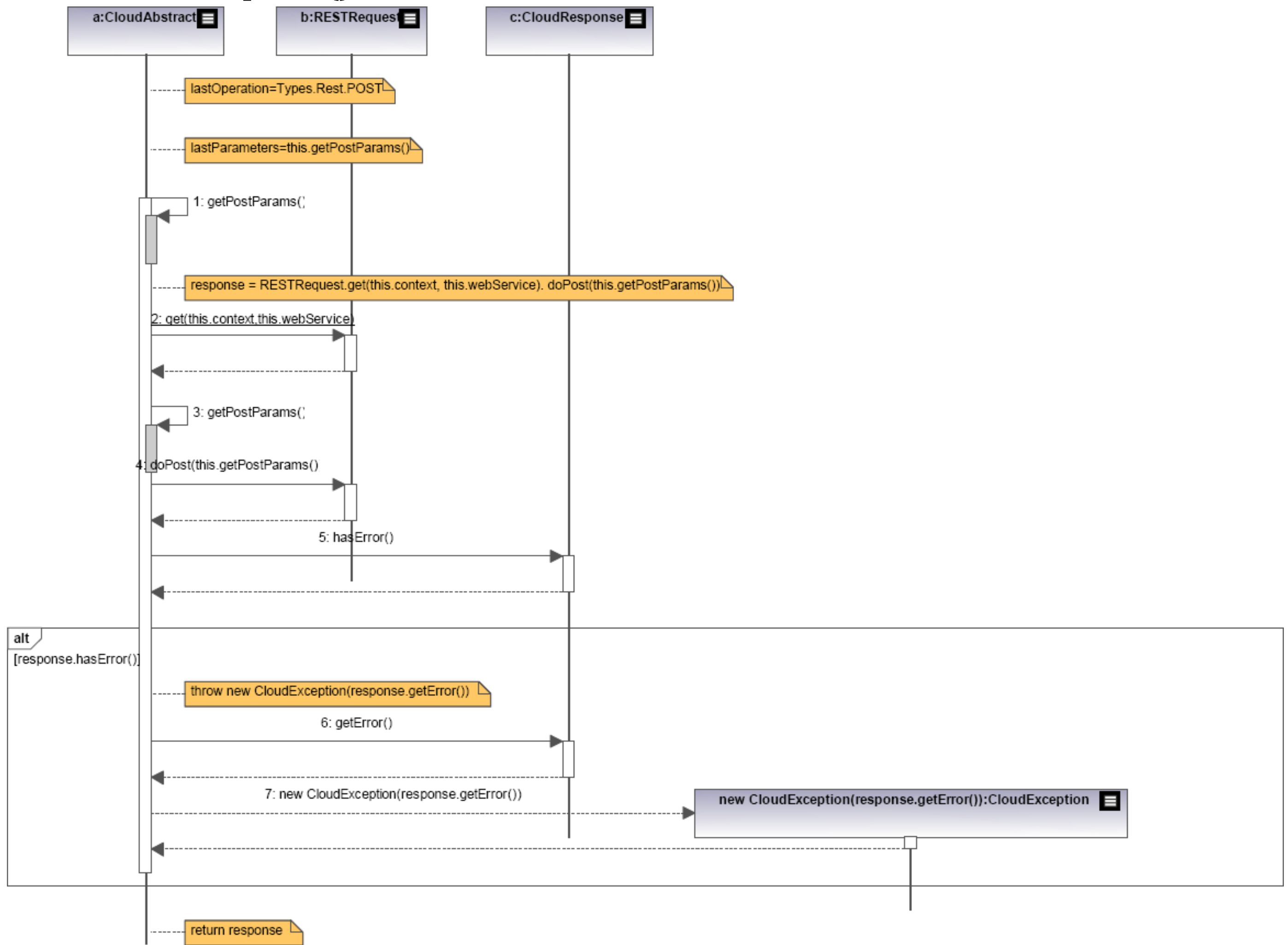
a:DBAbstract
this.db:SQLiteDatabase
e
b:SQLiteDatabase

strict
[try]
this.db = this.openOrCreateDatabase()
1: openOrCreateDatabase()
2: db.beginTransaction()
3: executeUpdate(oldObj,newObj)
4: db.setTransactionSuccessful()
[catch Exception e]
Log.e(Tags.DB, e.getMessage())
5: e.getMessage()
[finally]
6: db.endTransaction()
7: db.close()

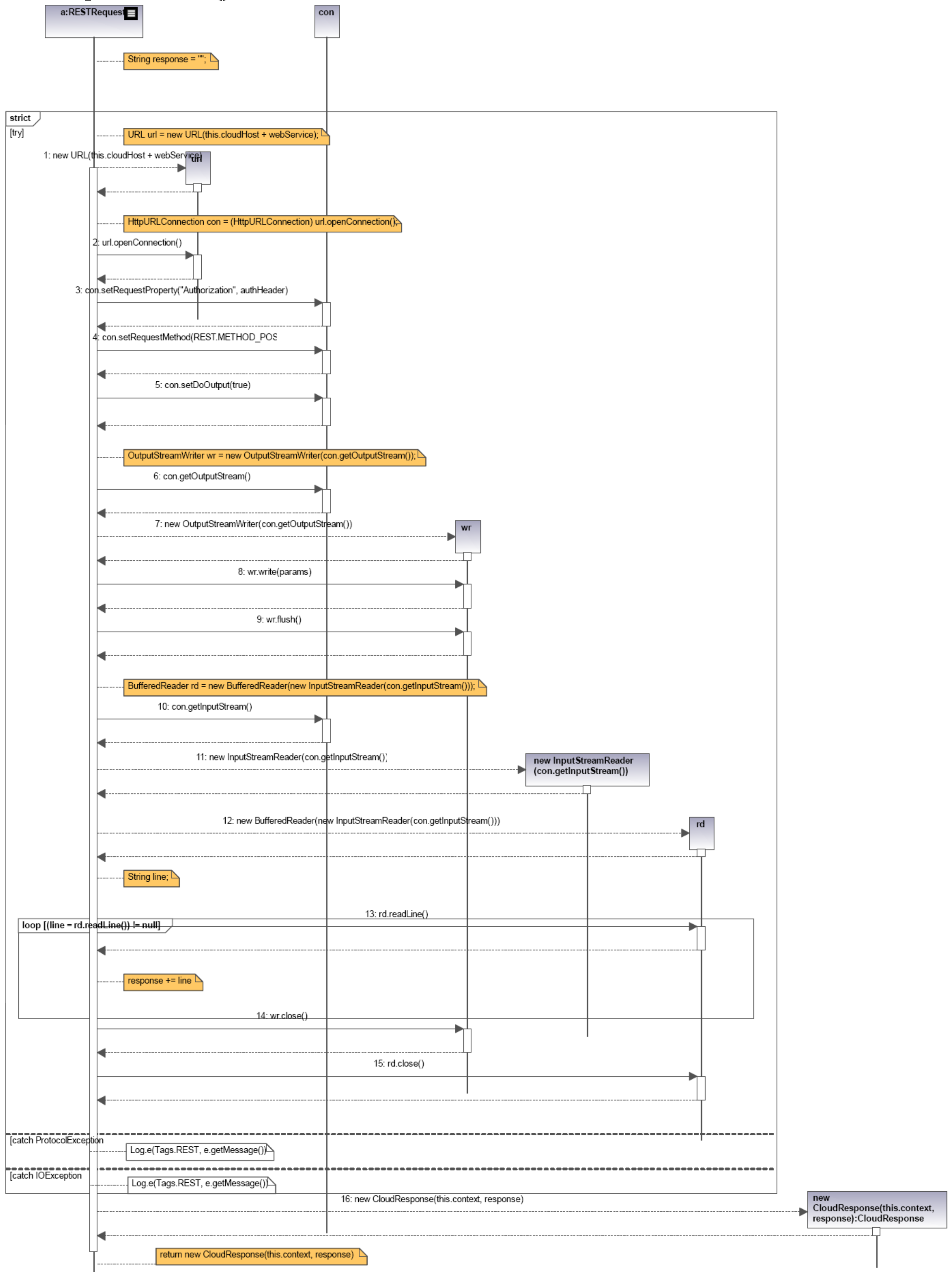
DBAbstract – newLocalId()



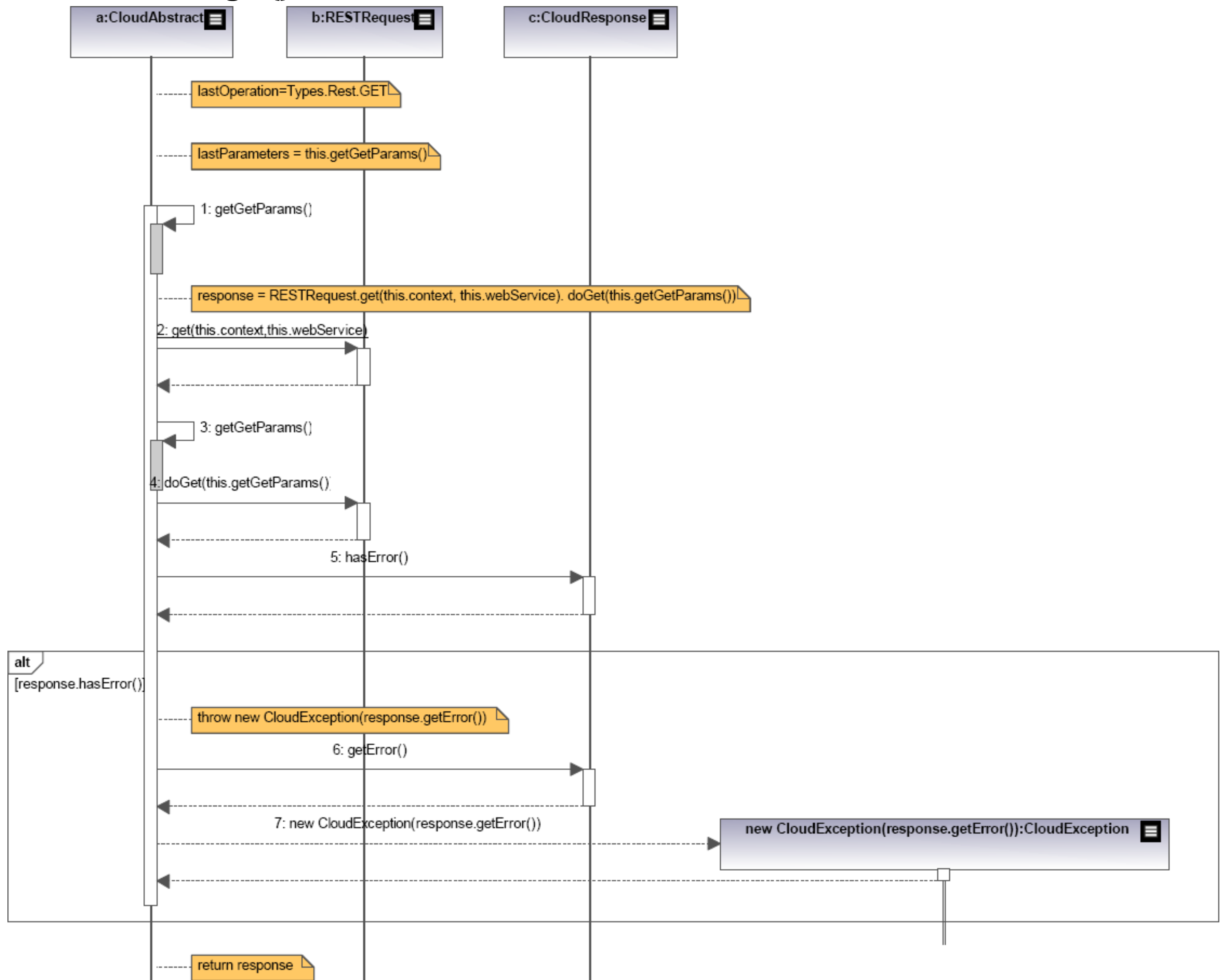
CloudAbstract-update()



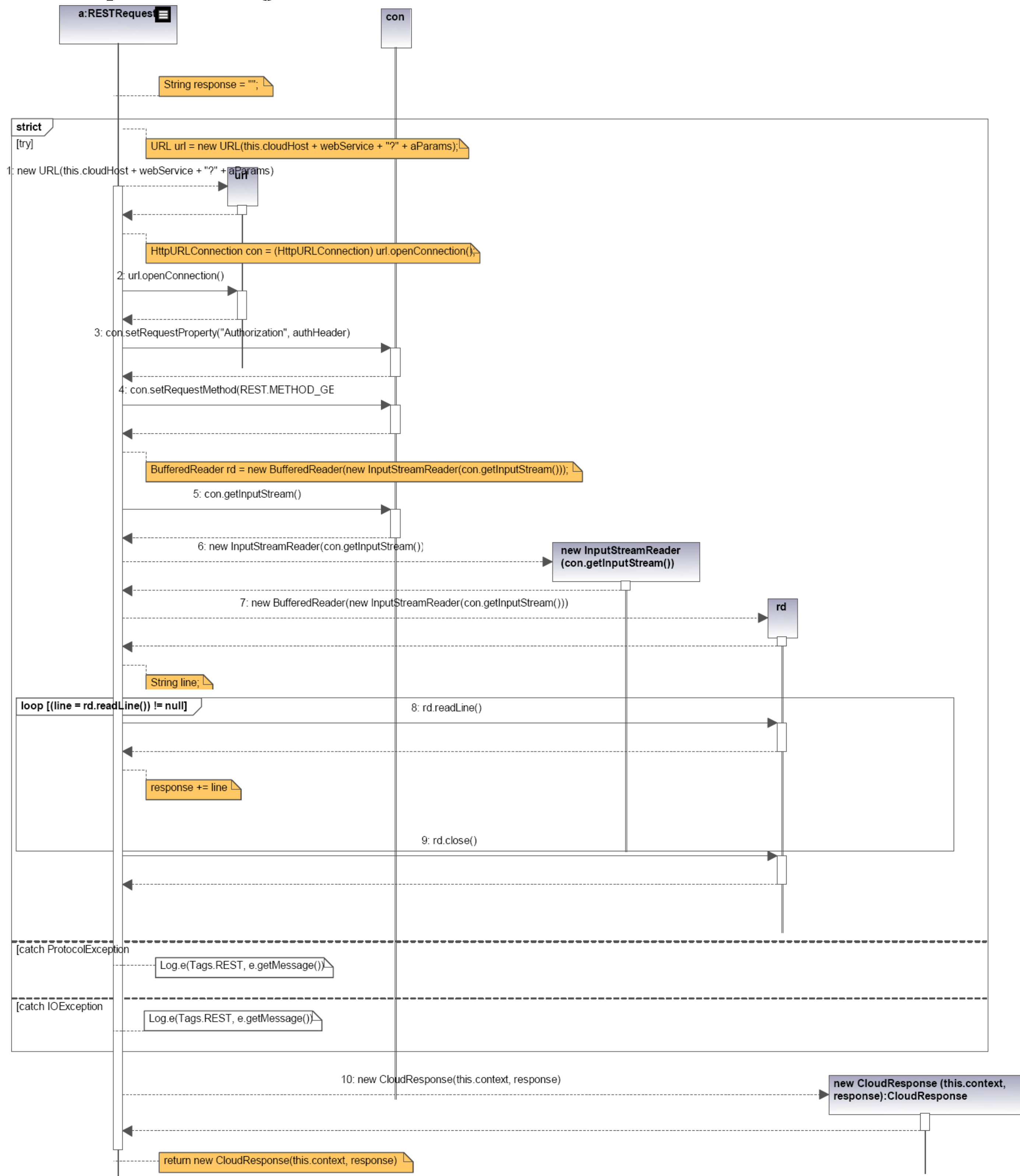
RESTRequest-doPut()



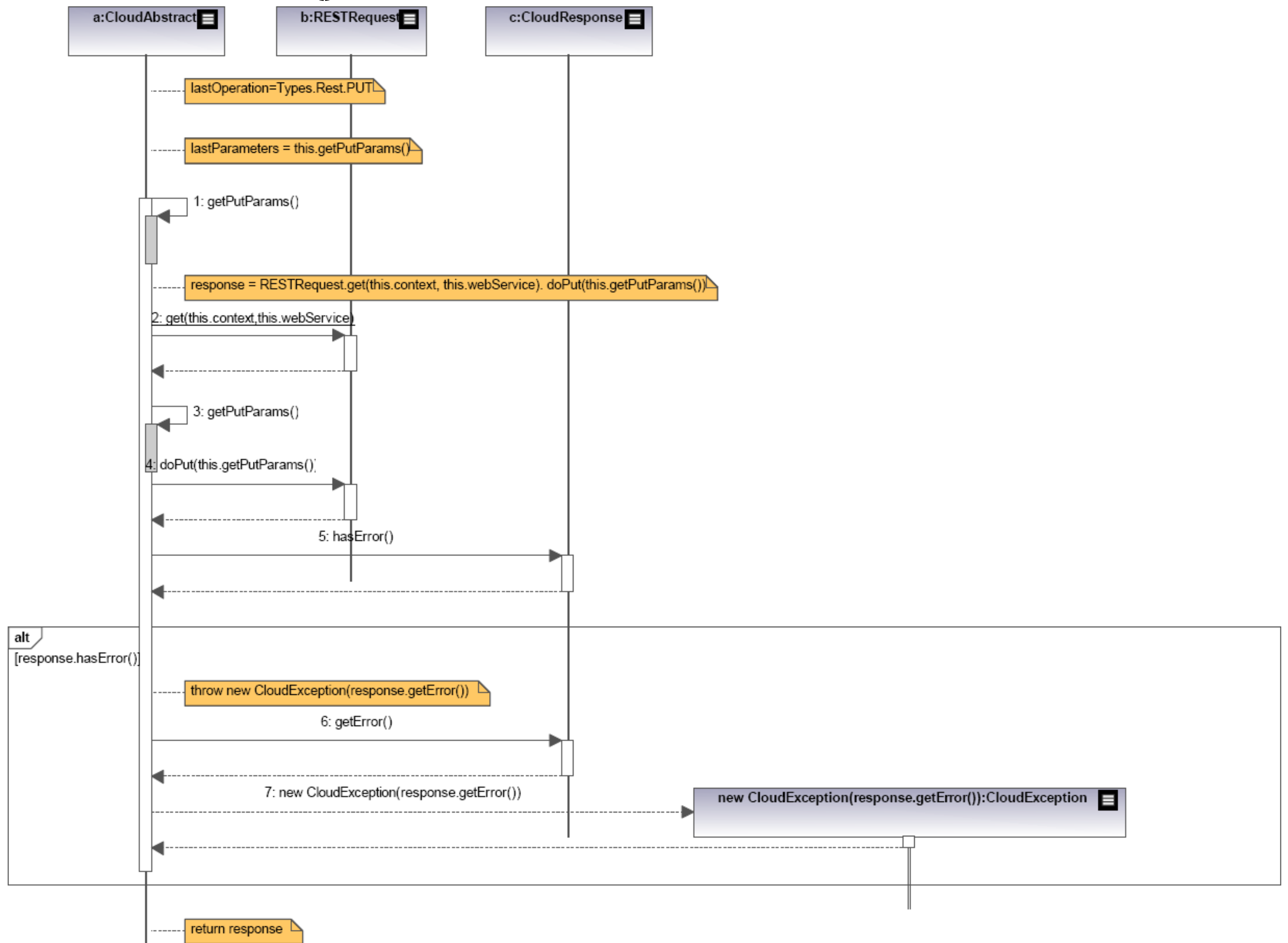
CloudAbstract - get()



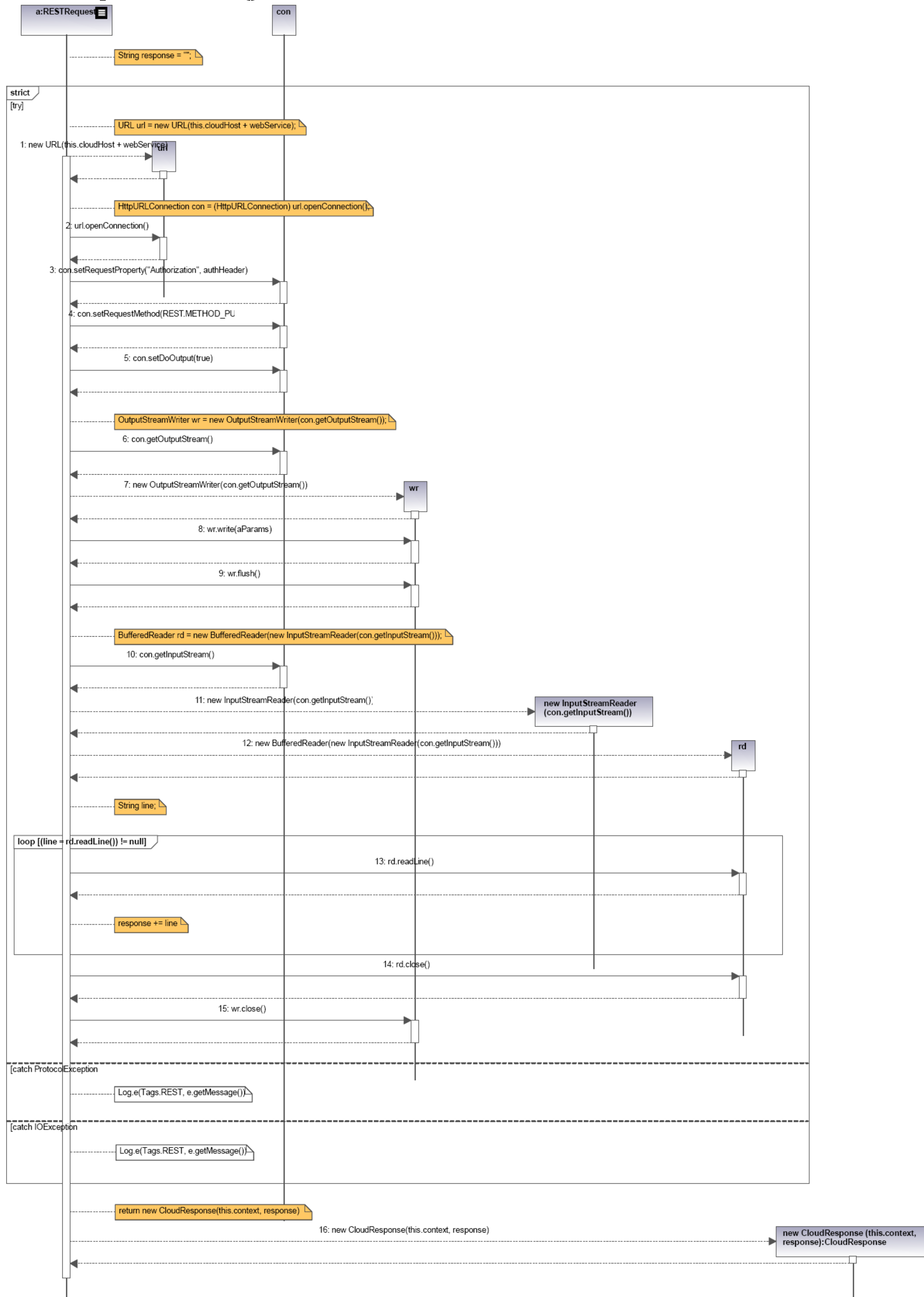
RESTRequest – doGet()



CloudAbstract- insert()

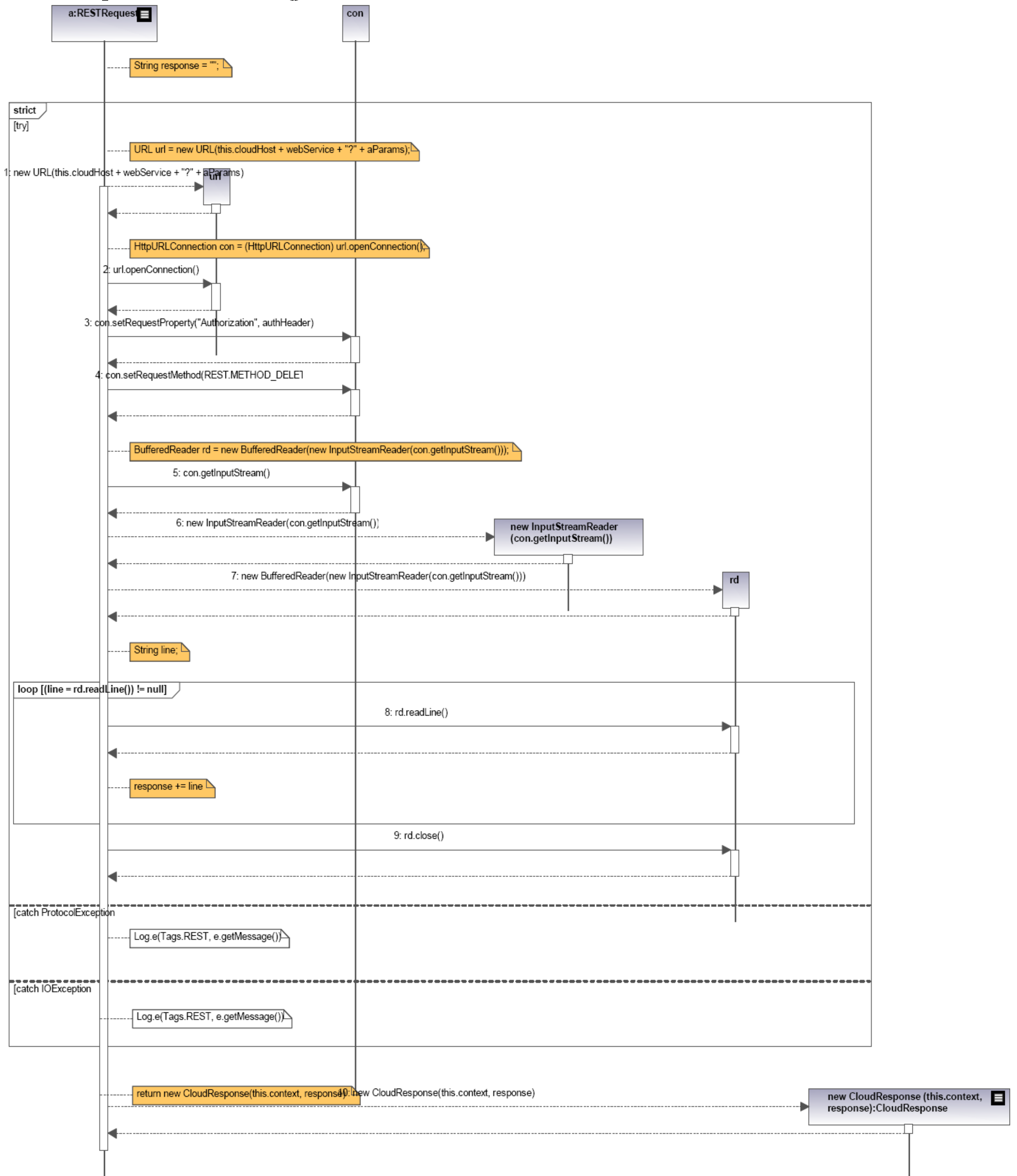


RESTRequest – doPut()

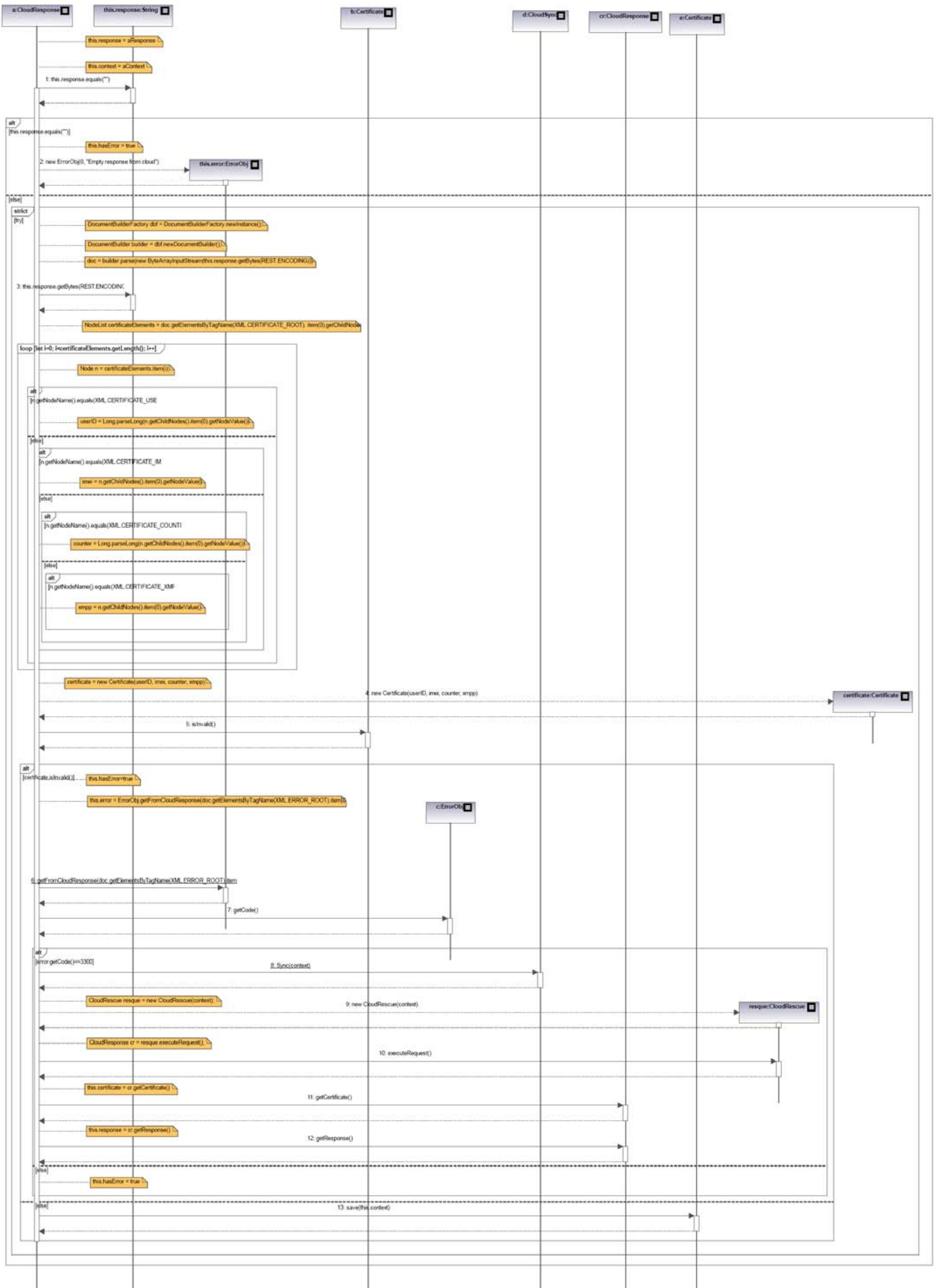


CloudAbstract – delete()

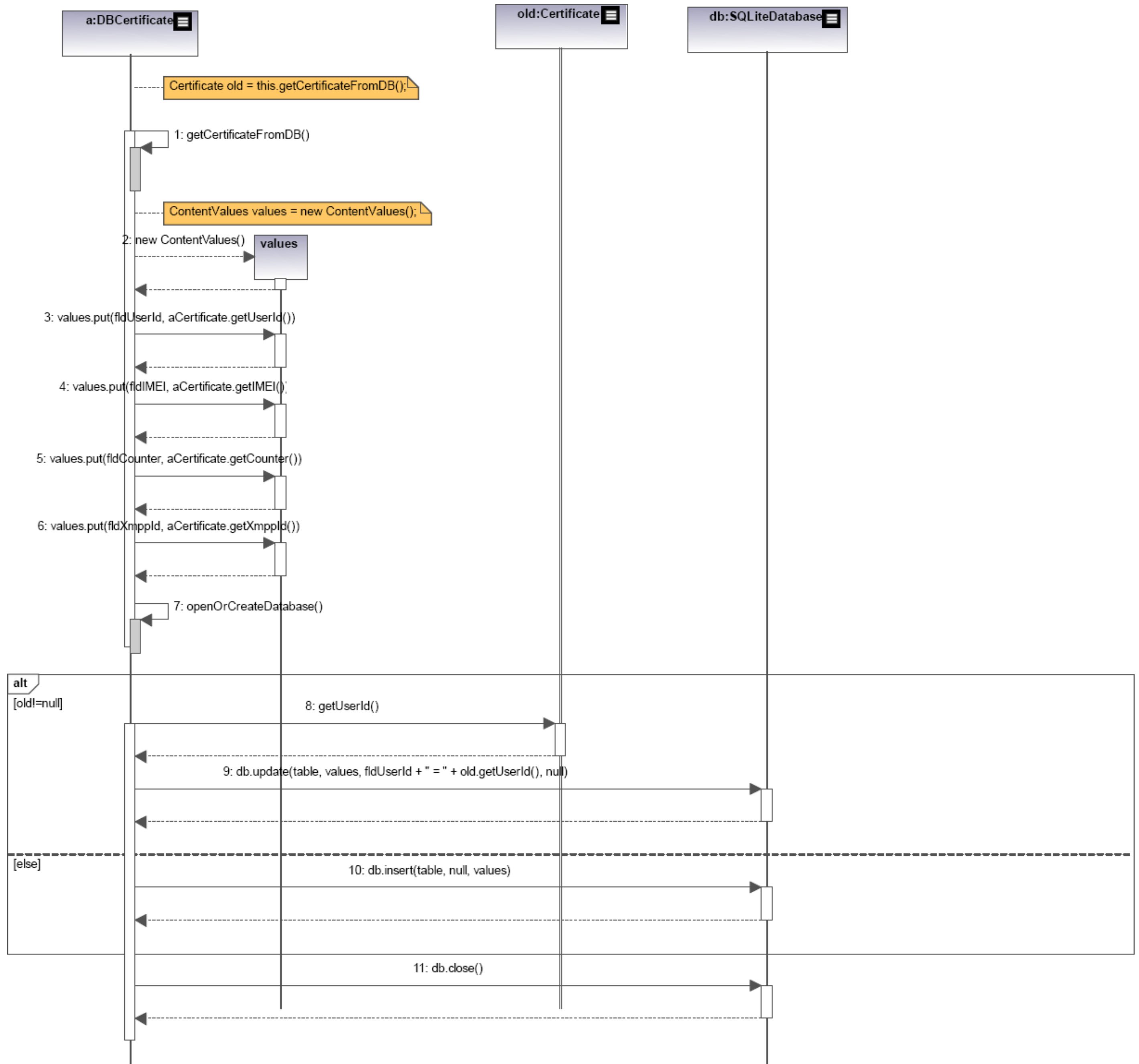
RESTRequest – doDelete()



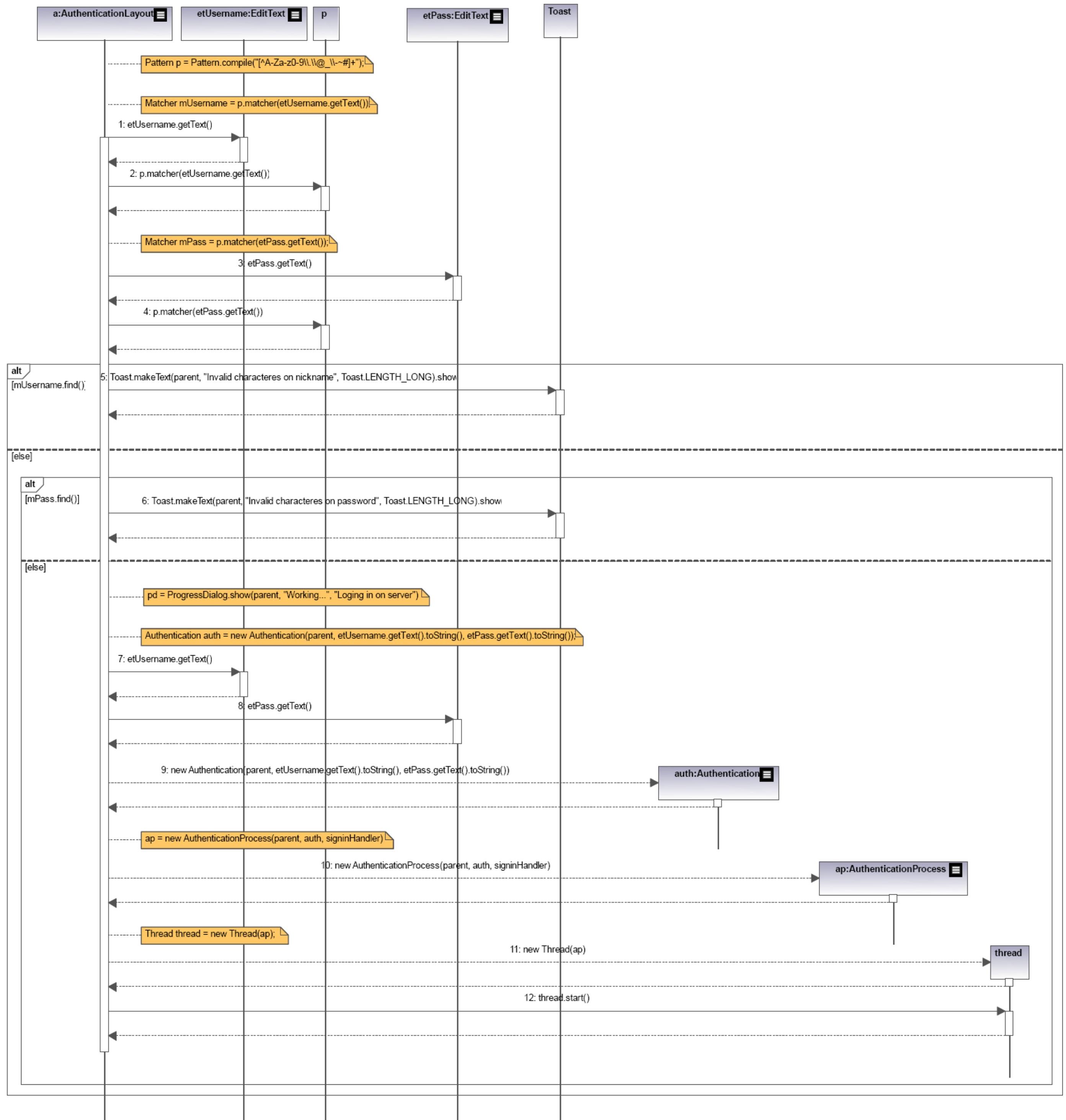
CloudResponse



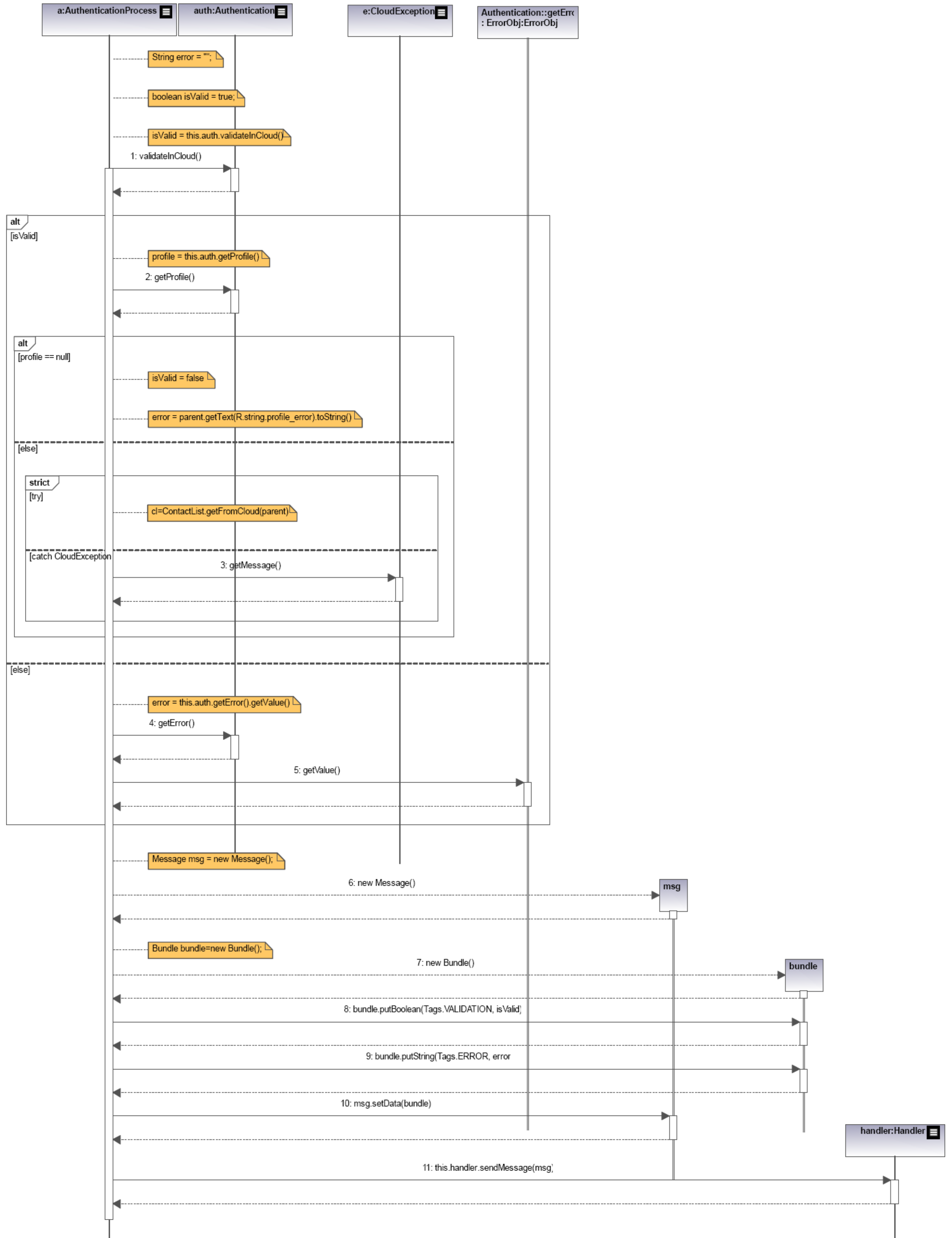
Certificate – save()



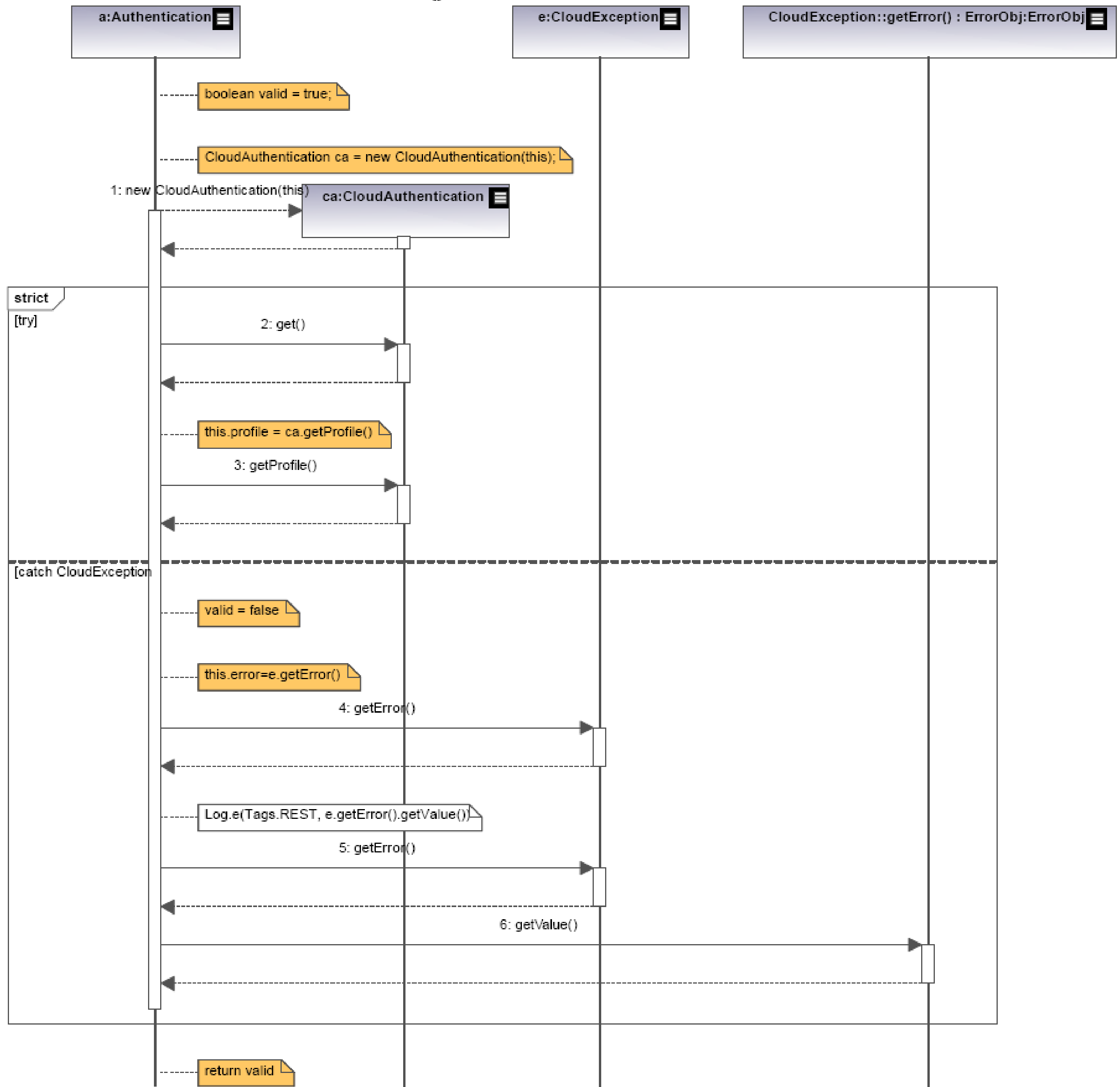
AuthenticationLayout – signin()



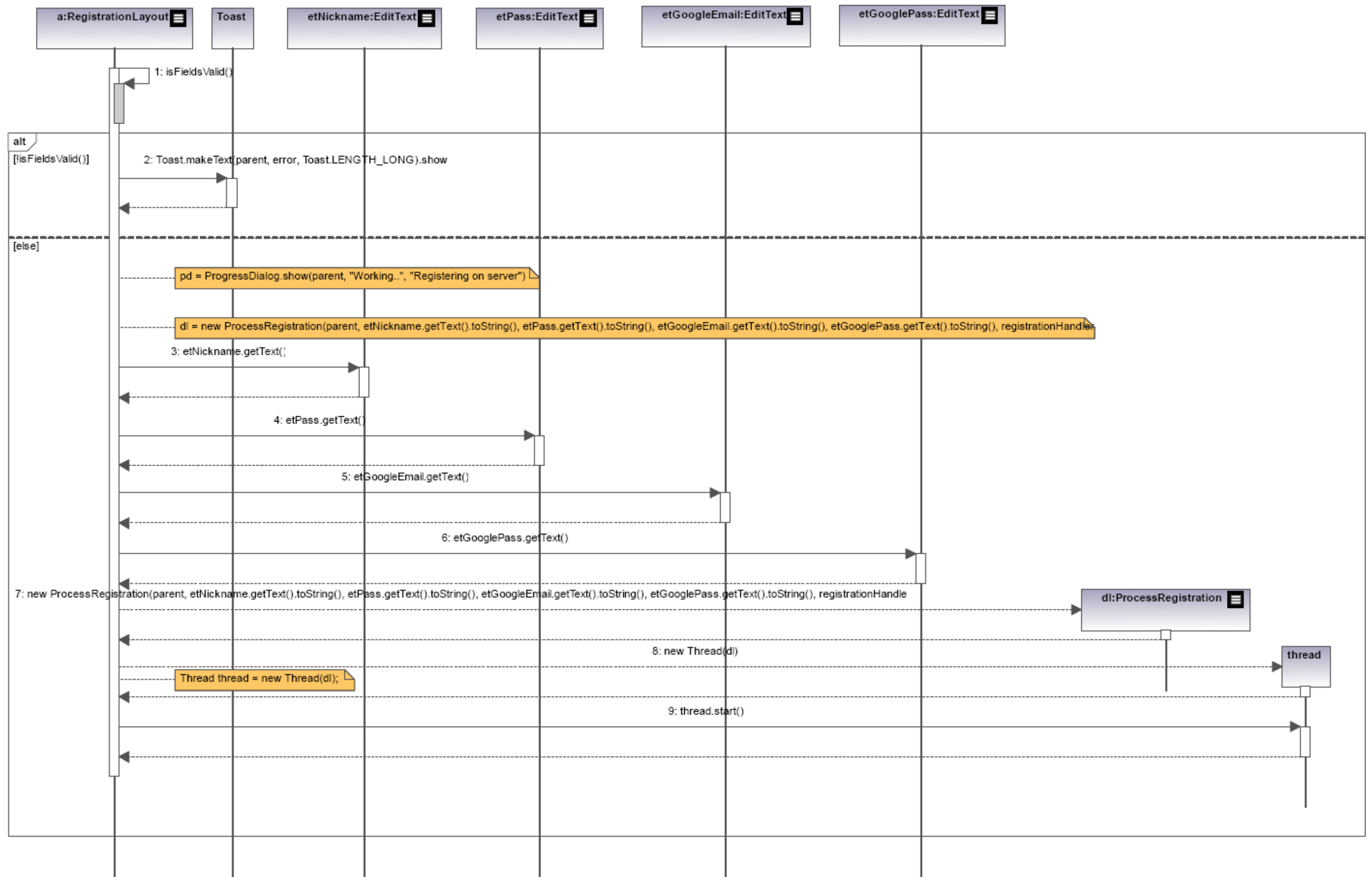
AuthenticationProcess – run()



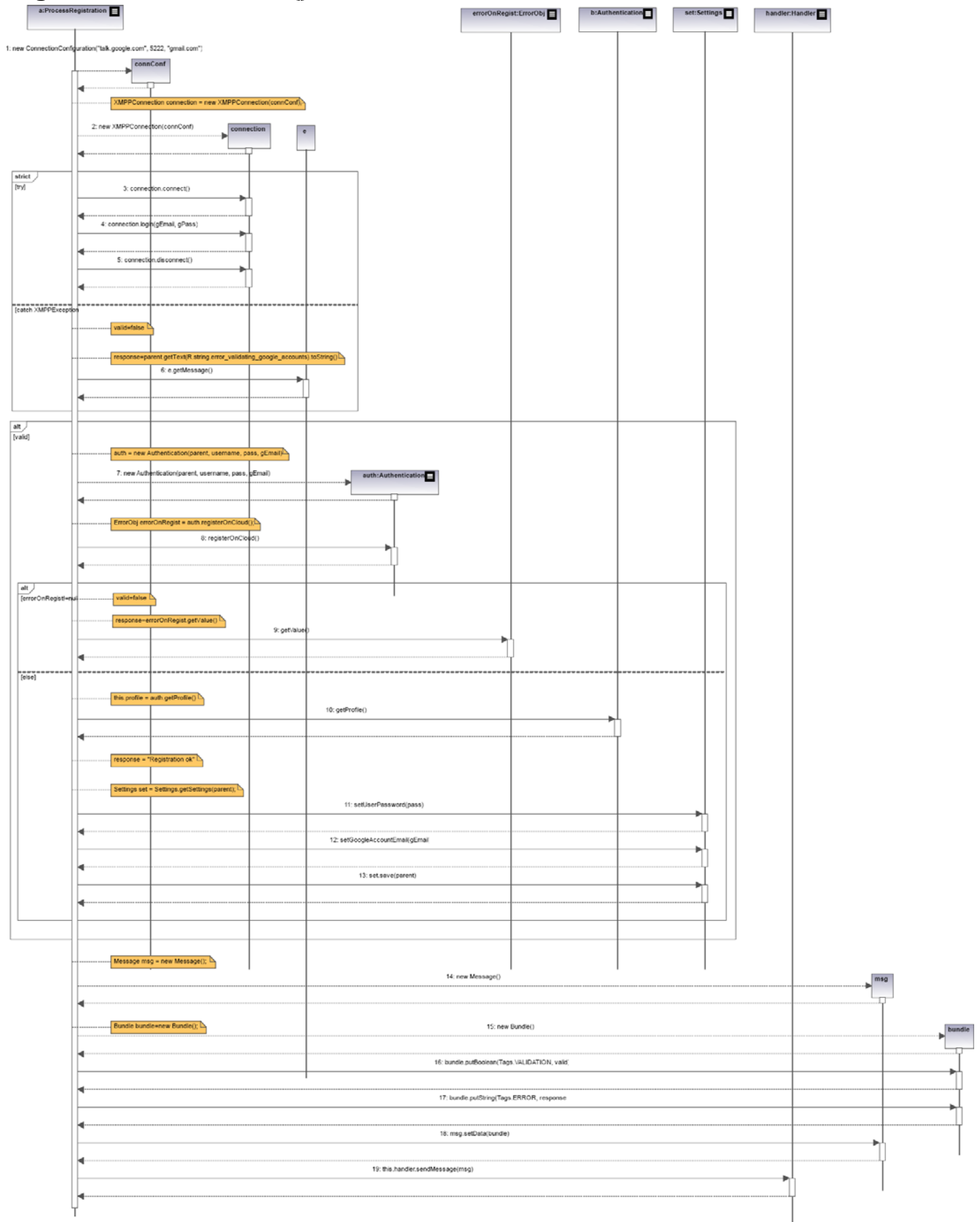
Authentication – ValidateOnCloud()



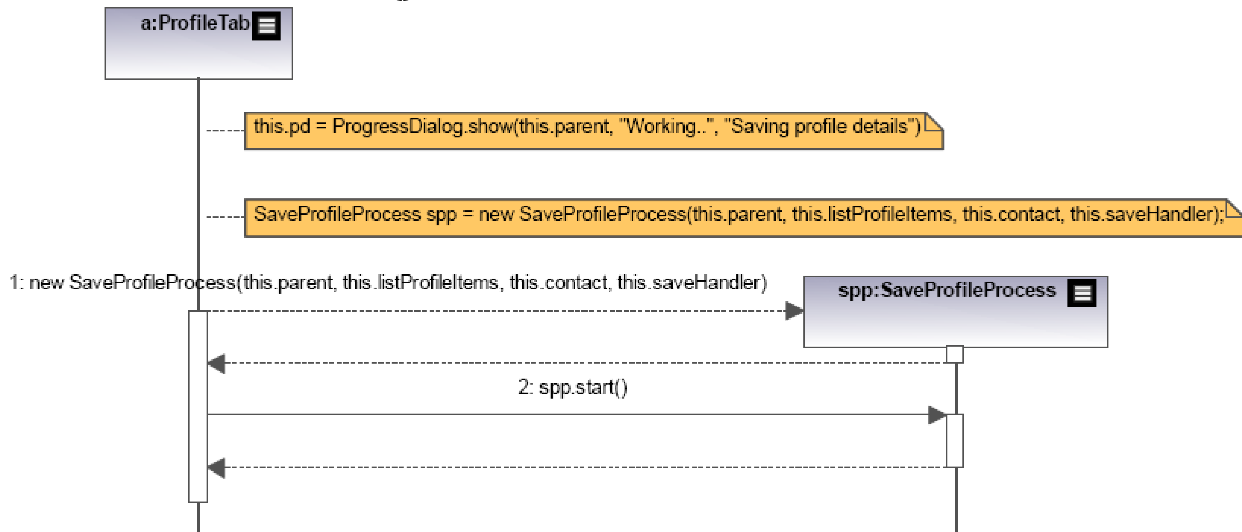
RegistrationLayout – regist()



RegistrationProcess – run()



ProfileTab – saveProfile()



ProfileItem – updateContactDetails()

a:ProfileItem

lcd:ListContactDetails

line:ProfileItemLine

cd:ContactDetails

1: removeAllFromType(this.type)

loop [ProfileItemLine line : this.lines]

ContactDetails cd = line.getContactDetail();

2: getContactDetail()

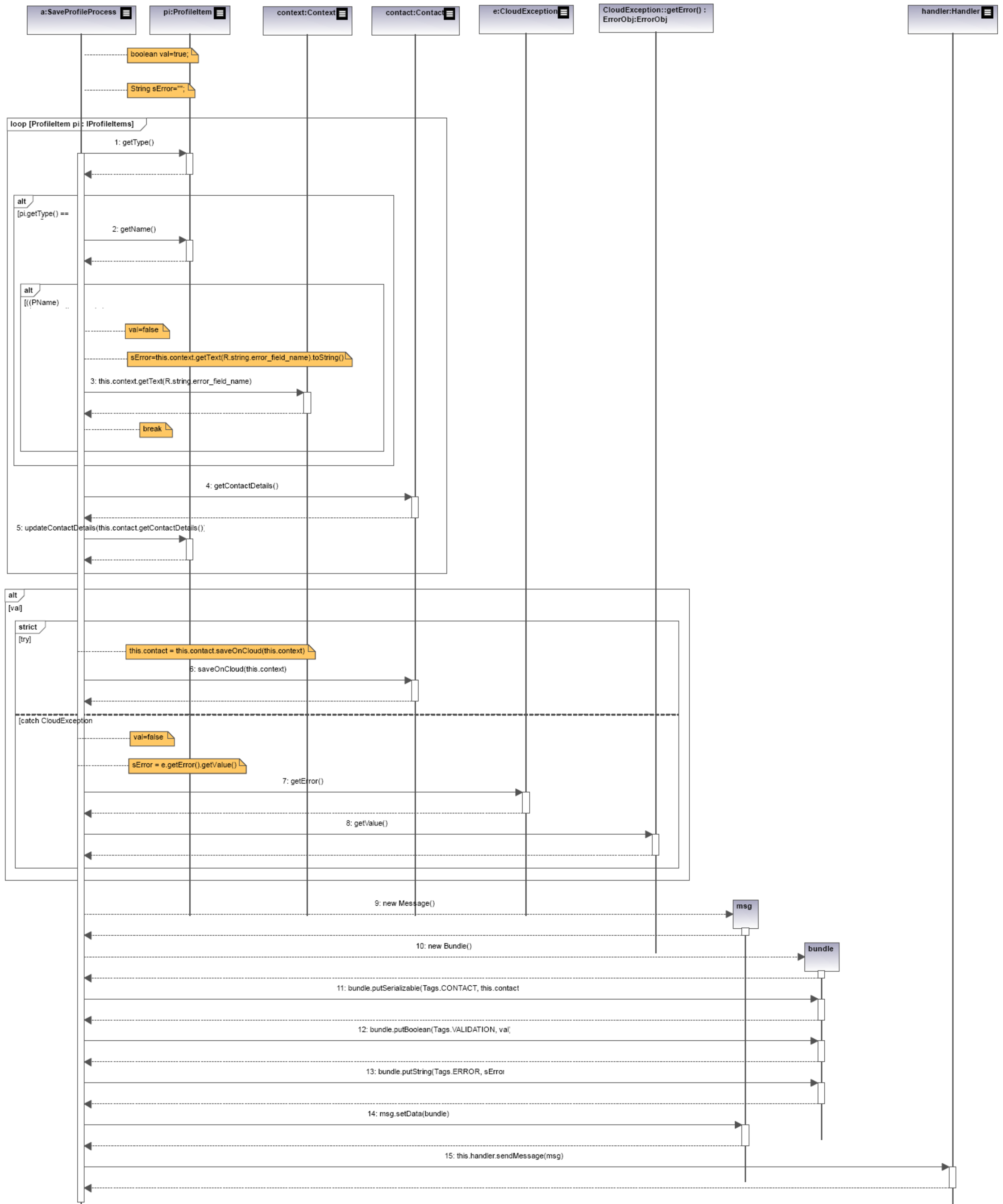
3: getValue()

alt

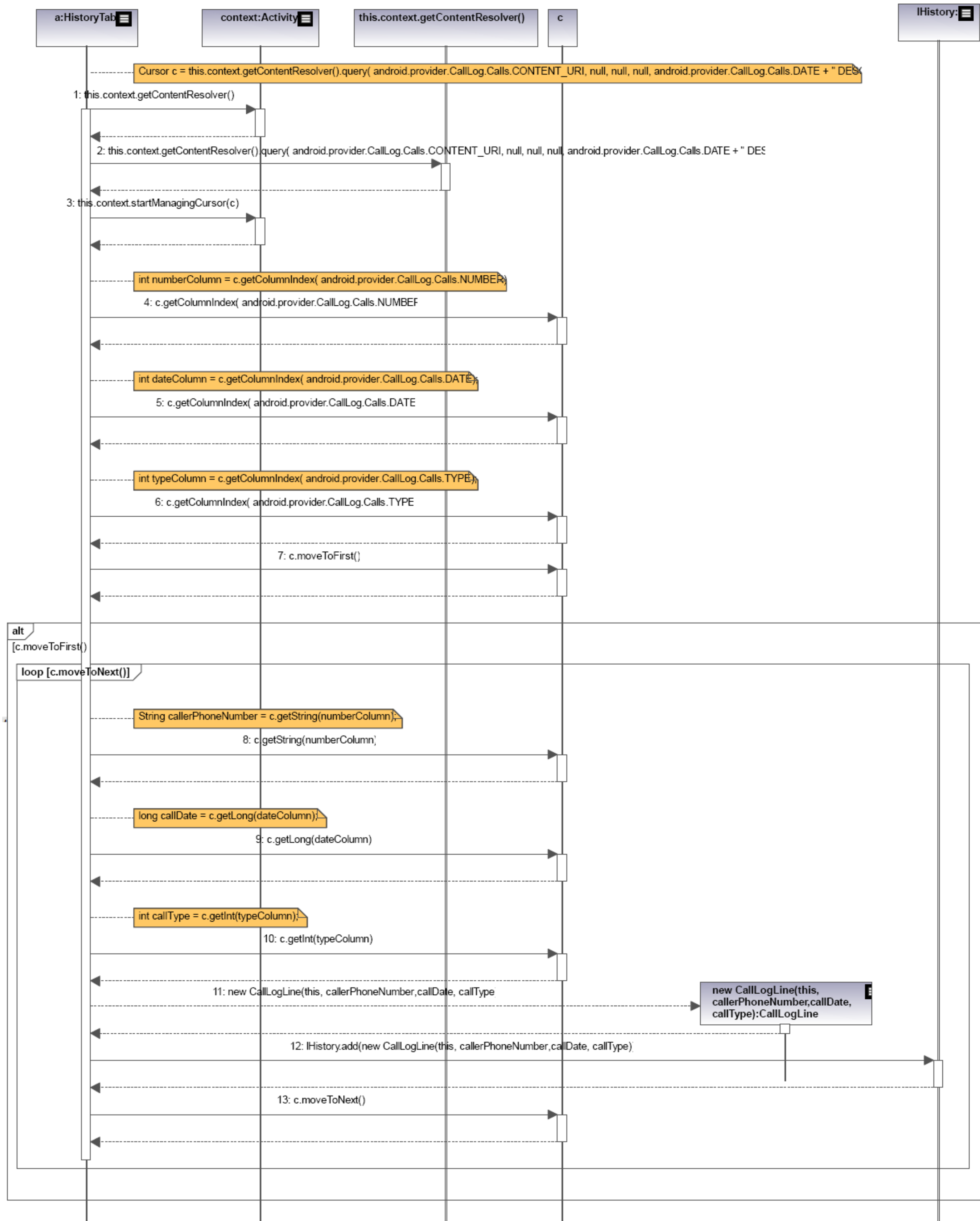
[!cd.getValue().equals("")]

4: lcd.add(cd)

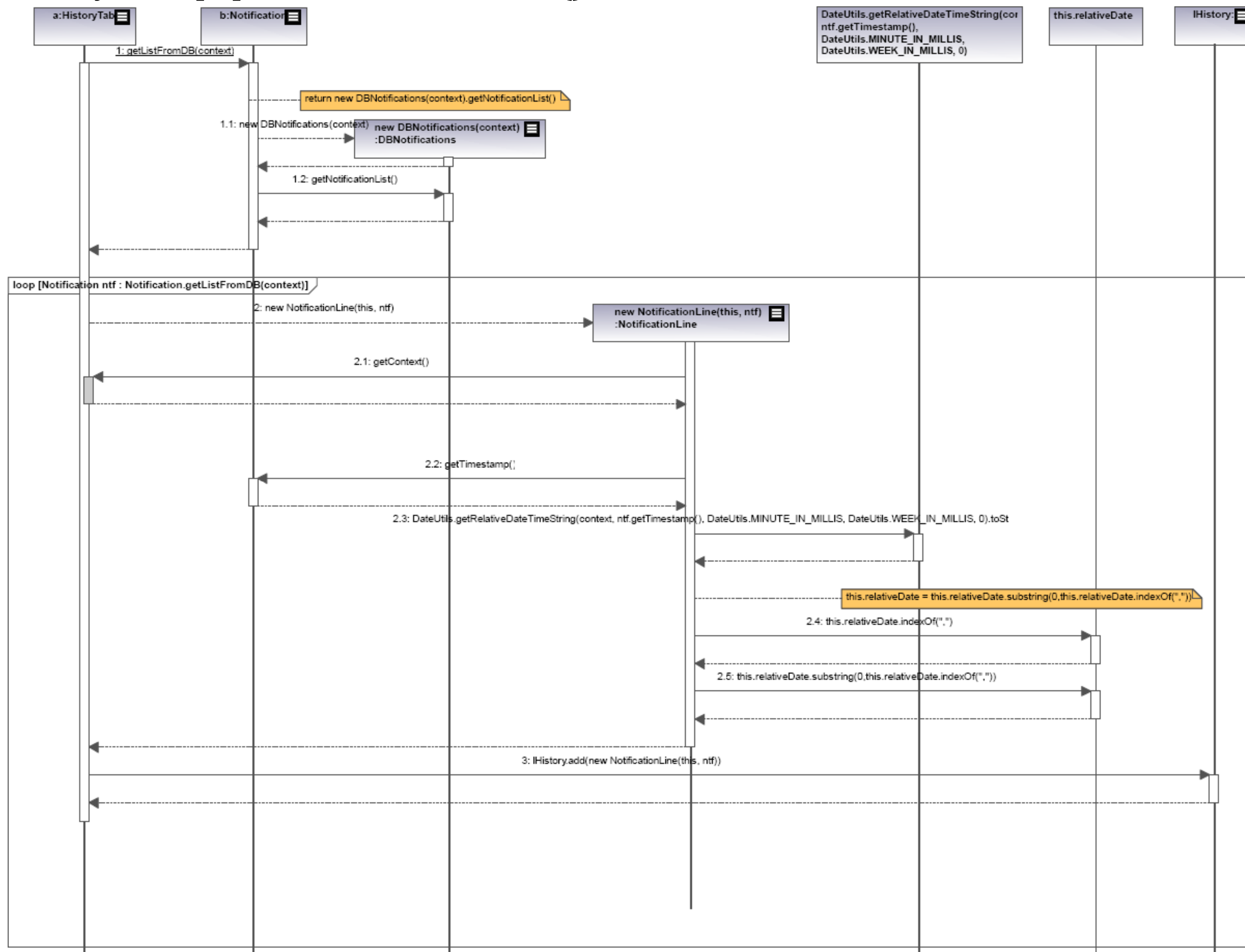
SaveProfileProcess – run()



HistoryTab – populateCallList()



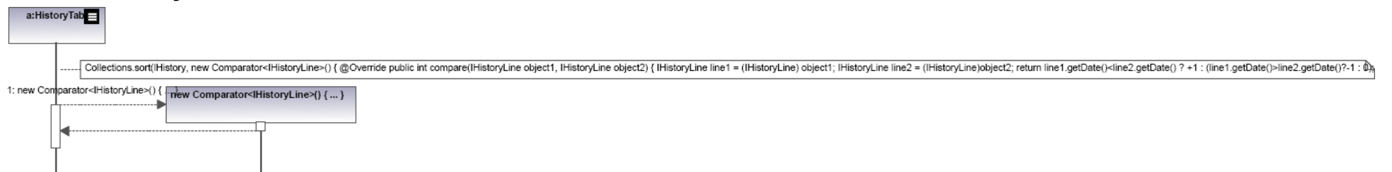
HistoryTab – populateNotificationsList()



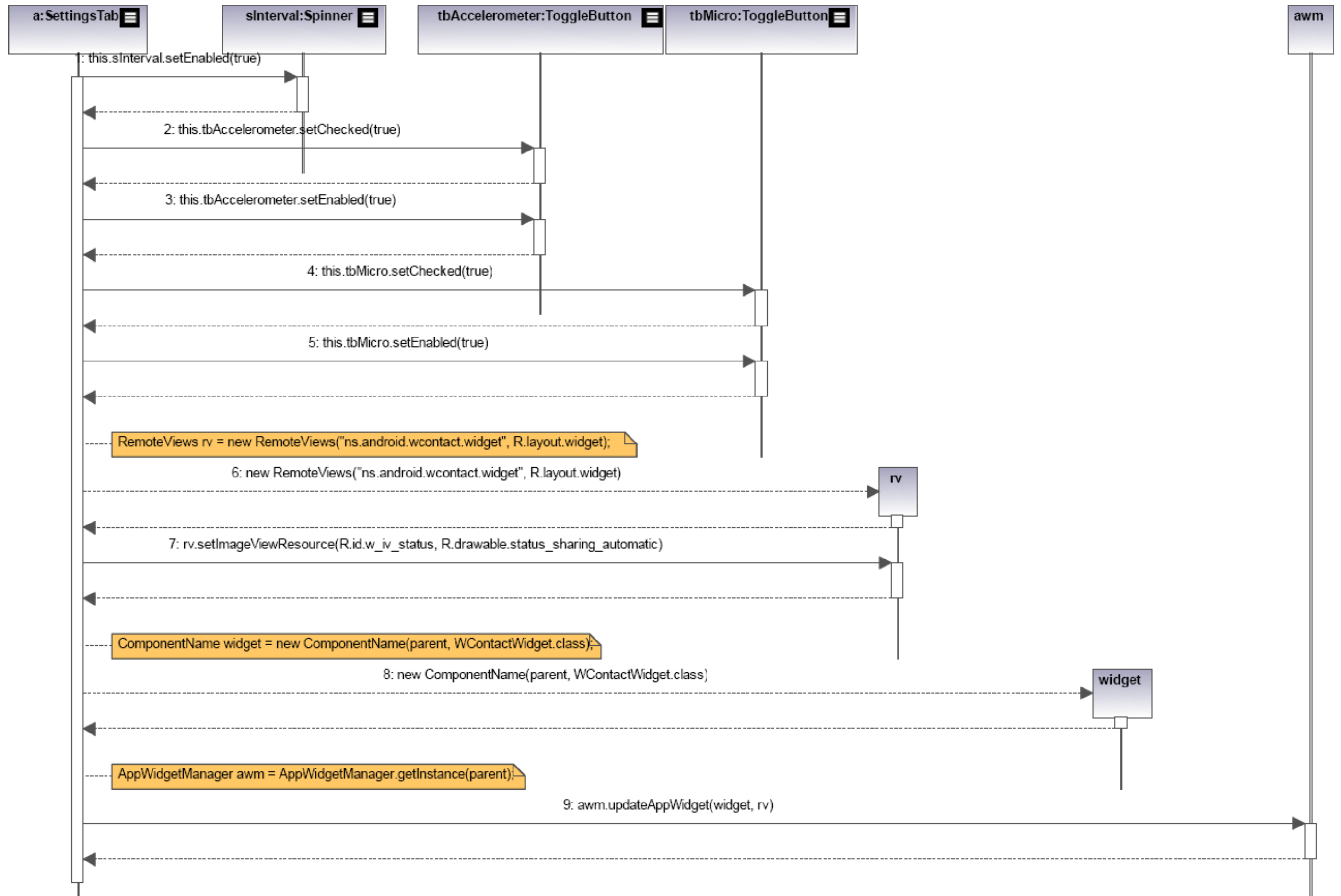
HistoryTab – populateInvitationsList()

a:HistoryTab
b:PendingInvitation
c:DBPendingInvitations
DateUtils.getRelativeDateTime pi.getTimestamp(), DateUtils.MINUTE_IN_MILLIS, DateUtils.WEEK_IN_MILLIS, 0)
this.relativeDate
lHistory:
1: getInvitationsFromDatabase(context)
return DBPendingInvitations.getInvitationsList(context)
1.1: getInvitationsList(context)
loop [PendingInvitation pi : PendingInvitation.getInvitationsFromDatabase(context)]
2: new InvitationLine(this, pi)
new InvitationLine(this, pi):InvitationLine
2.1: getContext()
2.2: getTimestamp()
2.3: DateUtils.getRelativeDateTimeString(context, pi.getTimestamp(), DateUtils.MINUTE_IN_MILLIS, DateUtils.WEEK_IN_MILLIS, 0).toS
this.relativeDate = this.relativeDate.substring(0,this.relativeDate.indexOf(","))
2.4: this.relativeDate.indexOf(",")
2.5: this.relativeDate.substring(0,this.relativeDate.indexOf(","))
3: lHistory.add(new InvitationLine(this, pi))

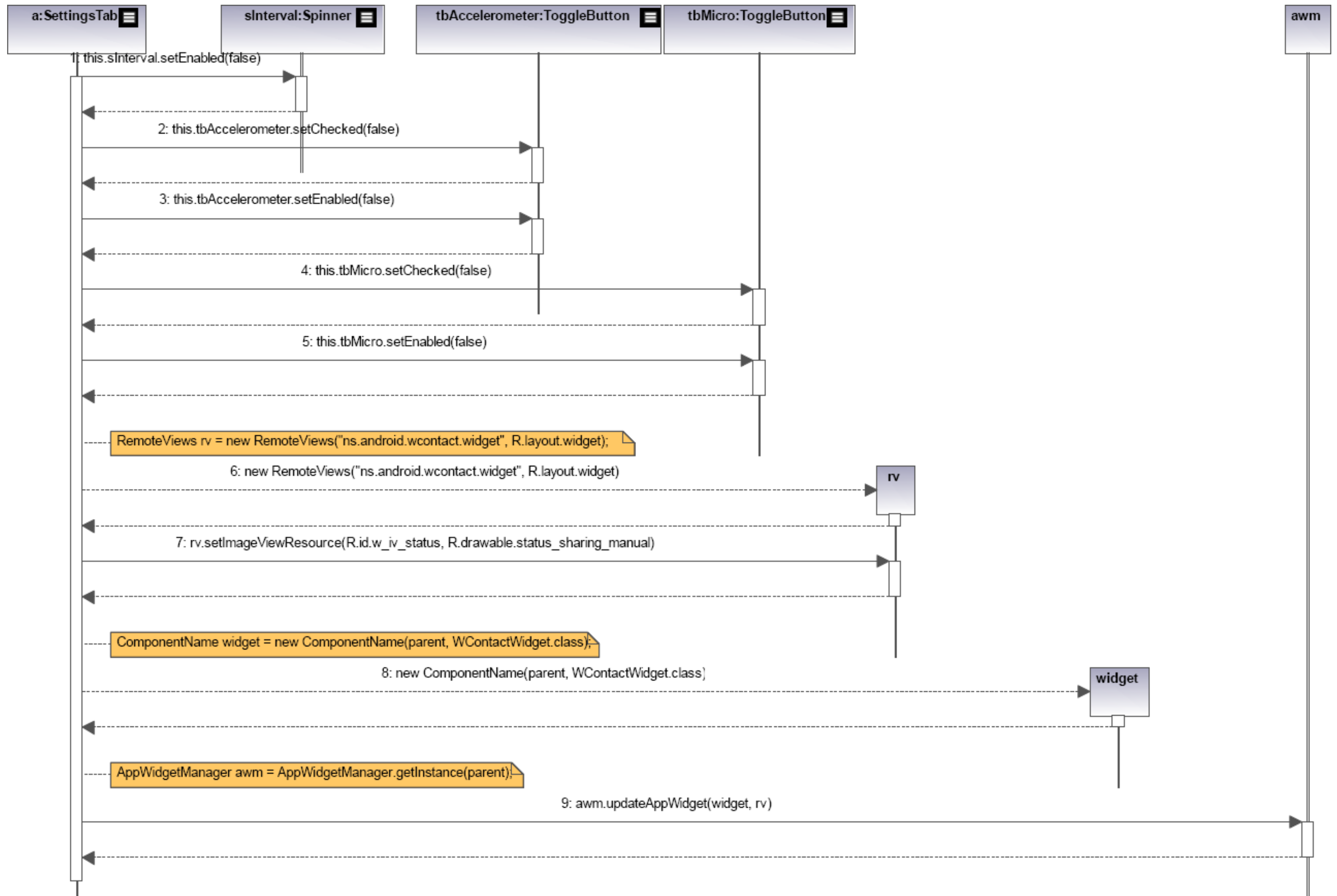
Sort history list



SettingsTab - activateAutomaticStatusDetection



SettingsTab - deactivateAutomaticStatusDetection



Tab creation

a:SettingsTab
aParent:WContact
this.parent.getText(R.string.tab_settings_name)
aTHost:TabHost
this.parent.getResources()
this.spec

this.parent = aParent

this.tHost = aTHost

this.spec = tHost.newTabSpec(this.parent.getText(R.string.tab_settings_name).toString())

1: this.parent.getText(R.string.tab_settings_name)

2: this.parent.getText(R.string.tab_settings_name).toString()

3: tHost.newTabSpec(this.parent.getText(R.string.tab_settings_name).toString())

4: this.parent.getText(R.string.tab_settings_name)

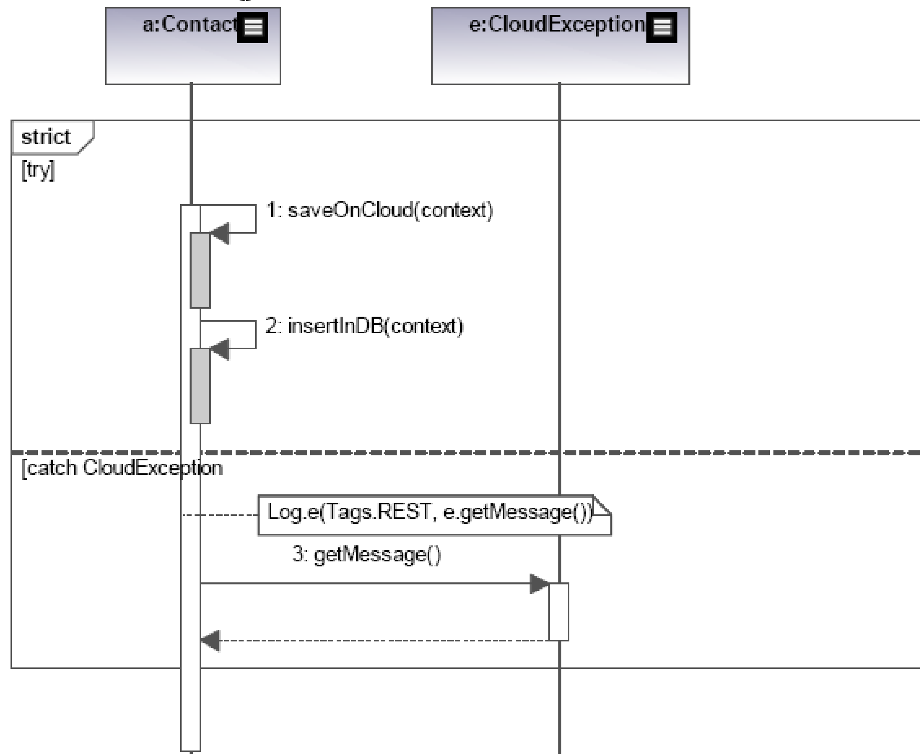
5: this.parent.getResources()

6: this.parent.getResources().getDrawable(R.drawable.ic_settings)

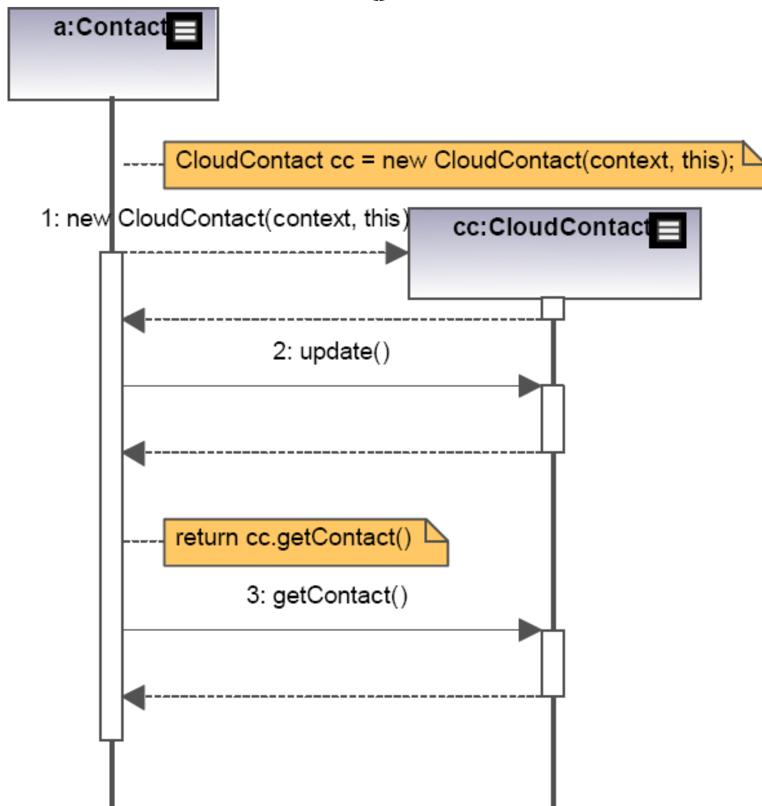
7: this.spec.setIndicator(this.parent.getText(R.string.tab_settings_name), this.parent.getResources().getDrawable(R.drawable.ic_settings))

8: this.spec.setContent(this)

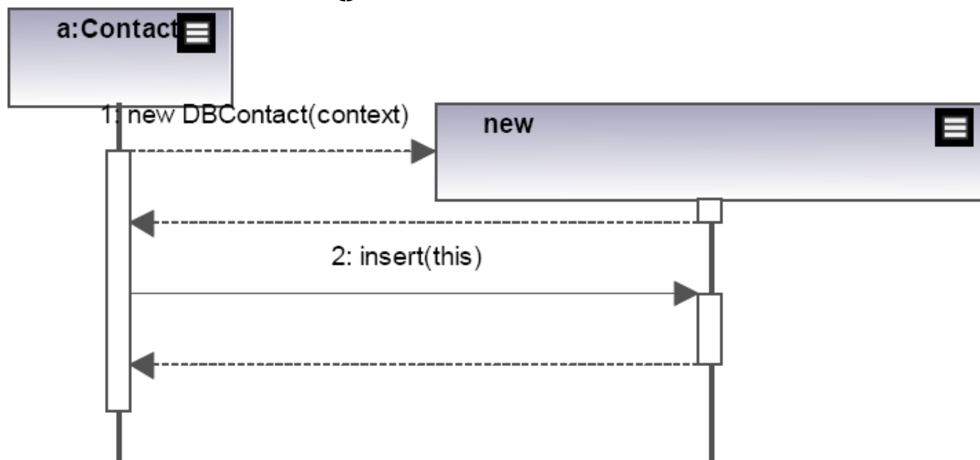
Contact – save()



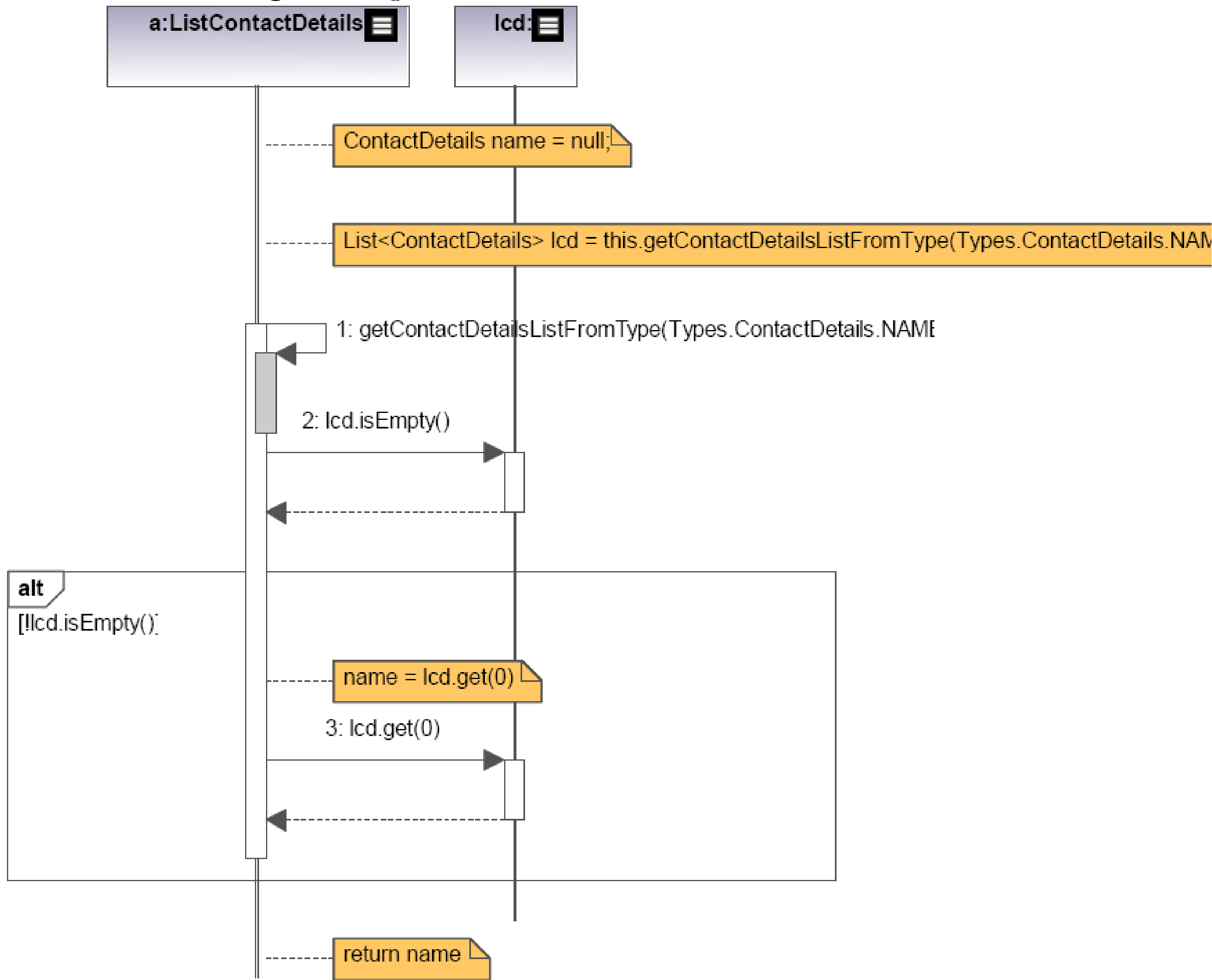
Contact – saveOnCloud()



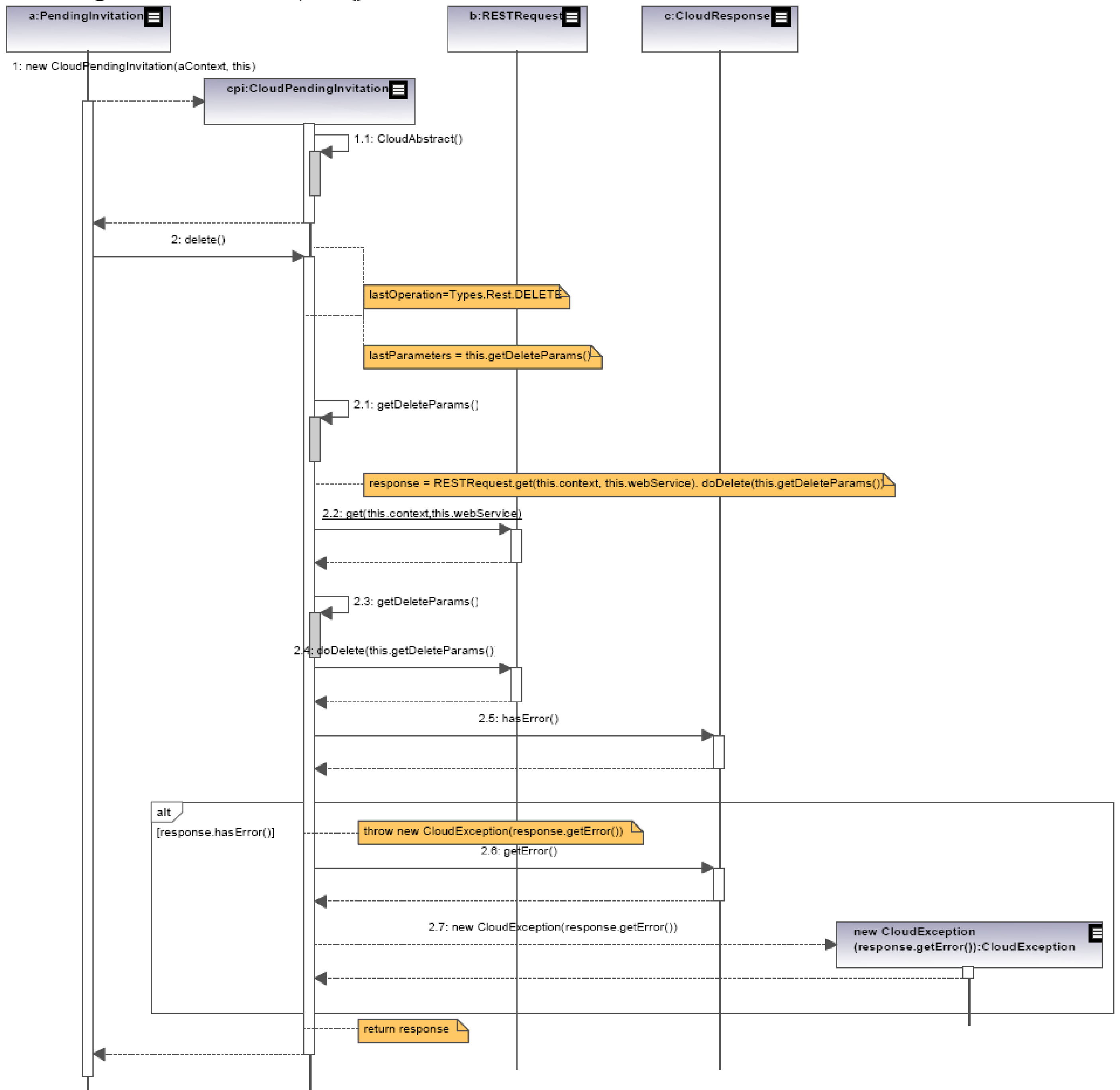
Contact – insertInDB()



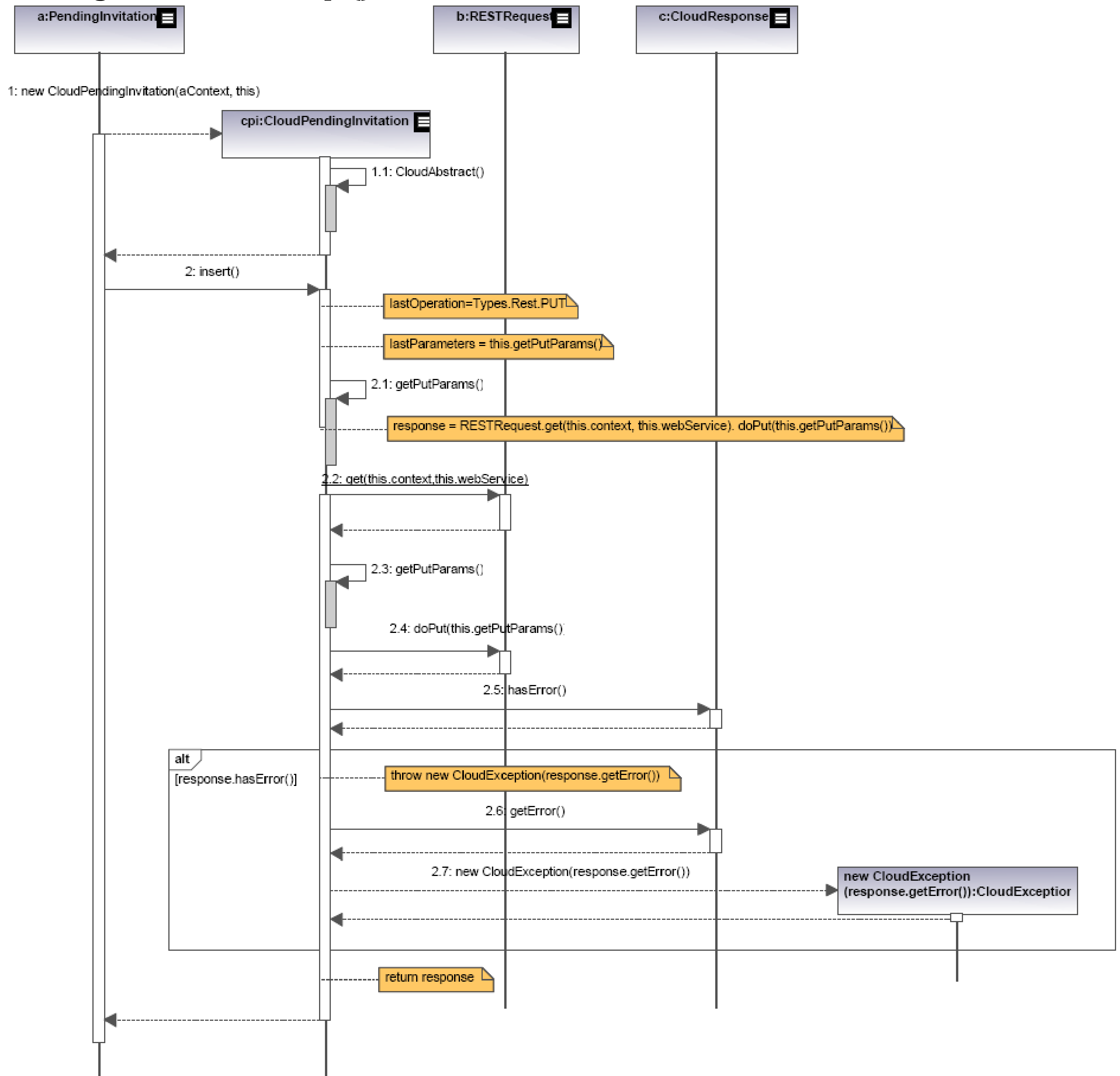
ListContactDetails – getName()



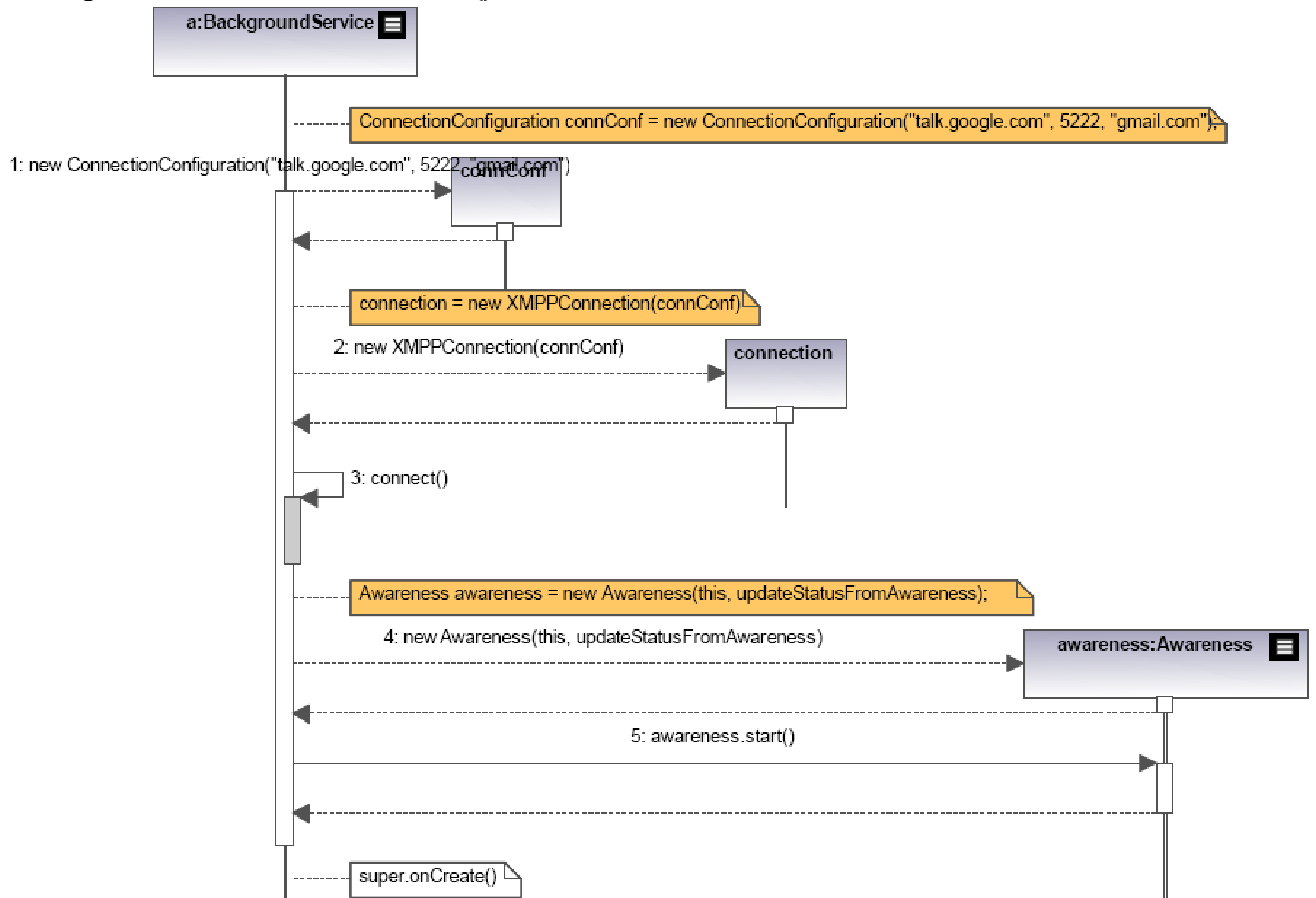
PendingInvitations – reject()



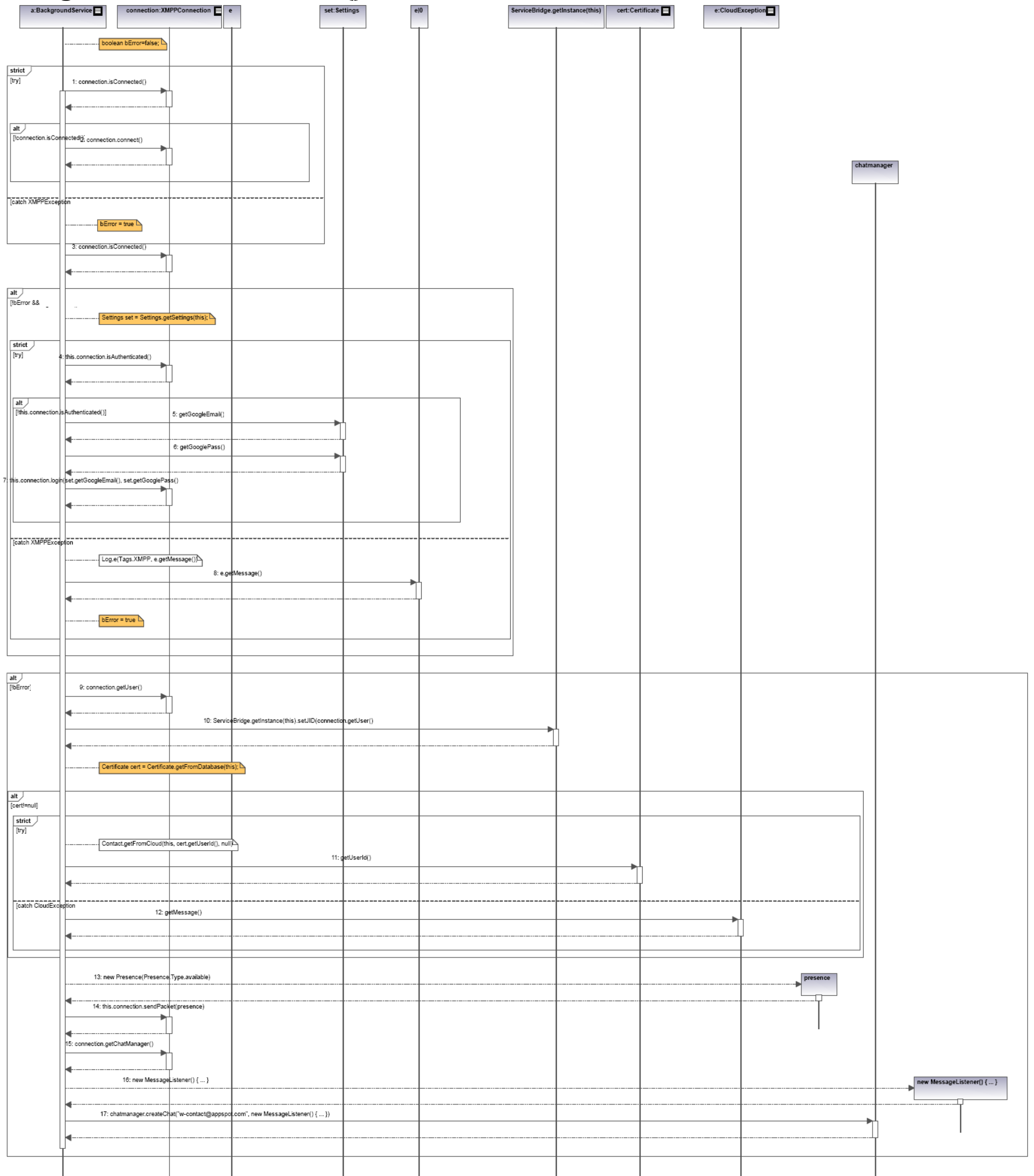
PendingInvitations – accept()



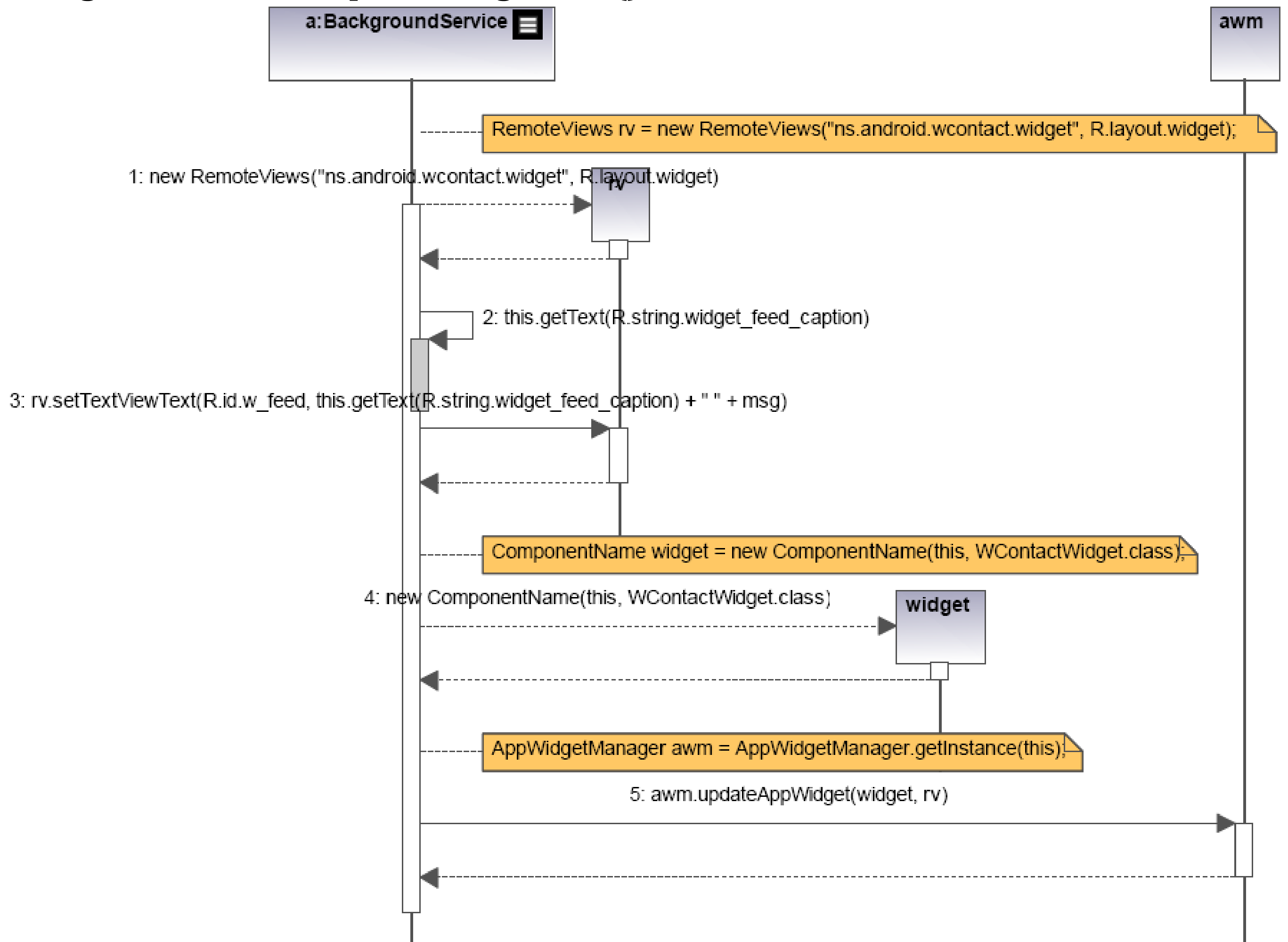
BackgroundService – onCreate()



BackgroundService – connect()



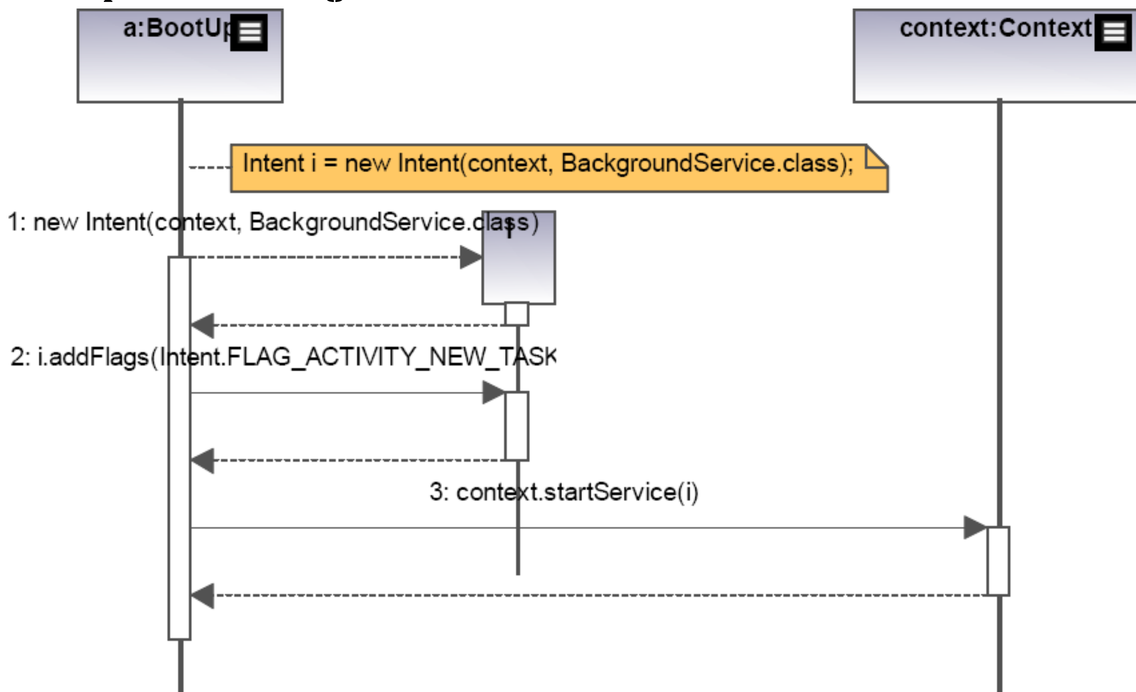
BackgroundService – updateWidgetFeed()



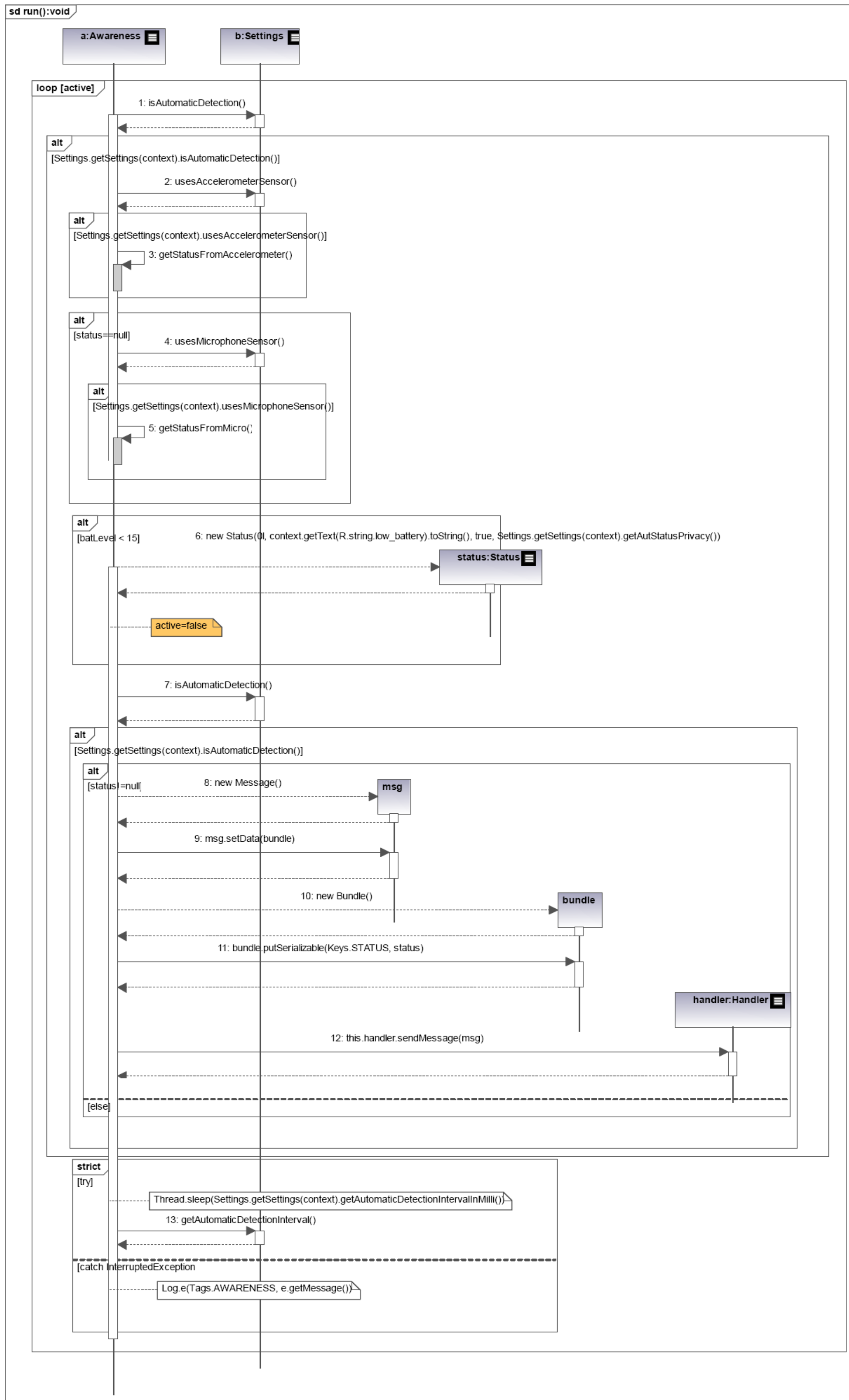
BackgroundService – showNotificaton()

a:BackgroundService
b:Parse
nm
NotificationManager nm = (NotificationManager) this.getSystemService(NOTIFICATION_SERVICE
1: this.getSystemService(NOTIFICATION_SERVIC
android.app.Notification notification = new android.app.Notification(R.drawable.icon, "wContact notification received", System.currentTimeMillis());
notification
2: new android.app.Notification(R.drawable.icon, "wContact notification received", System.currentTimeMillis())
Intent intent = new Intent(this, WContact.class);
3: new Intent(this, WContact.class)
intent
4: intent.putExtra(Tags.NOTIFICATION, tru
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, intent, Intent.FLAG_ACTIVITY_NEW_TASK);
5: notification.setLatestEventInfo(this, text, text, contentIntent)
6: longToInt(id)
7: nm.notify(Parse.longToInt(id), notification)

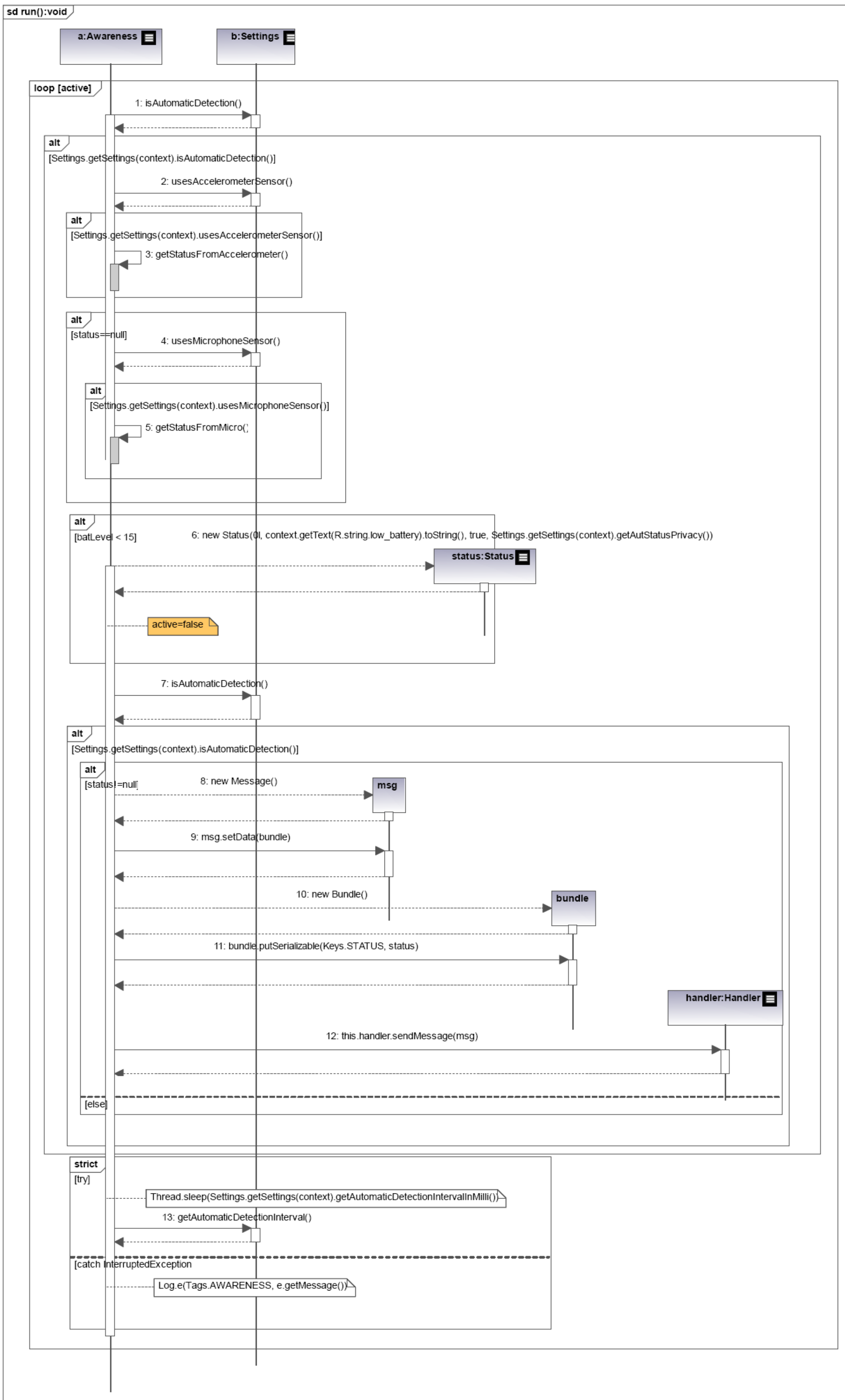
BootUp – onReceive()



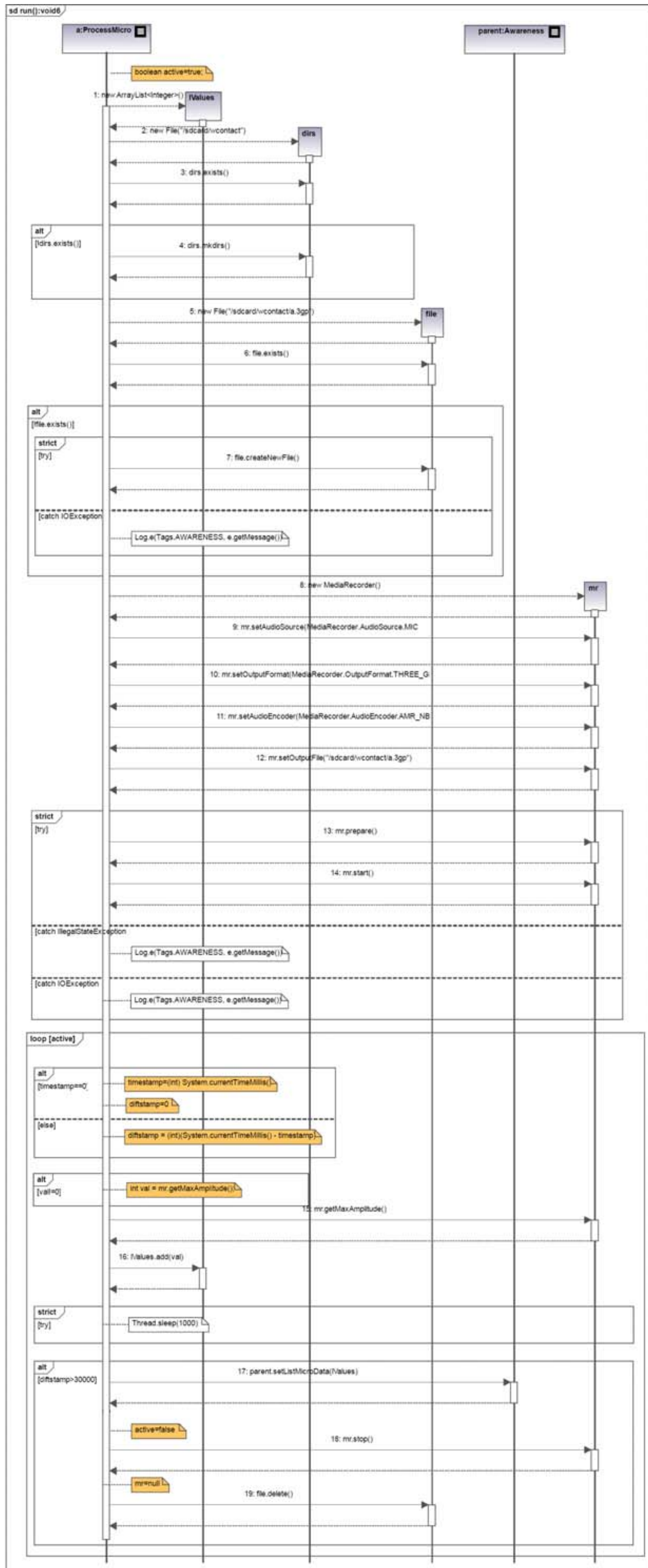
AwarenessProcess – run()



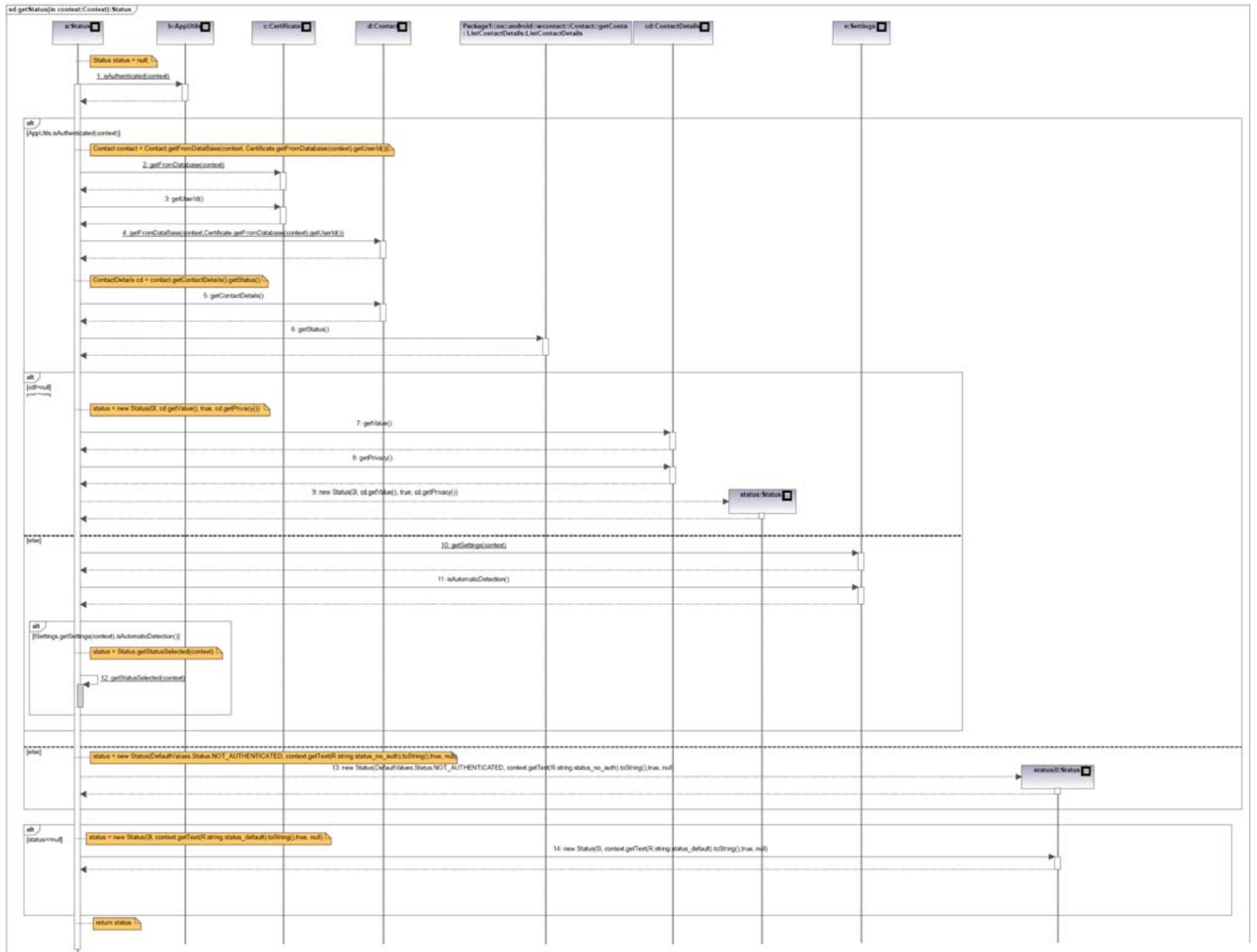
AccelerometerProcess – run()



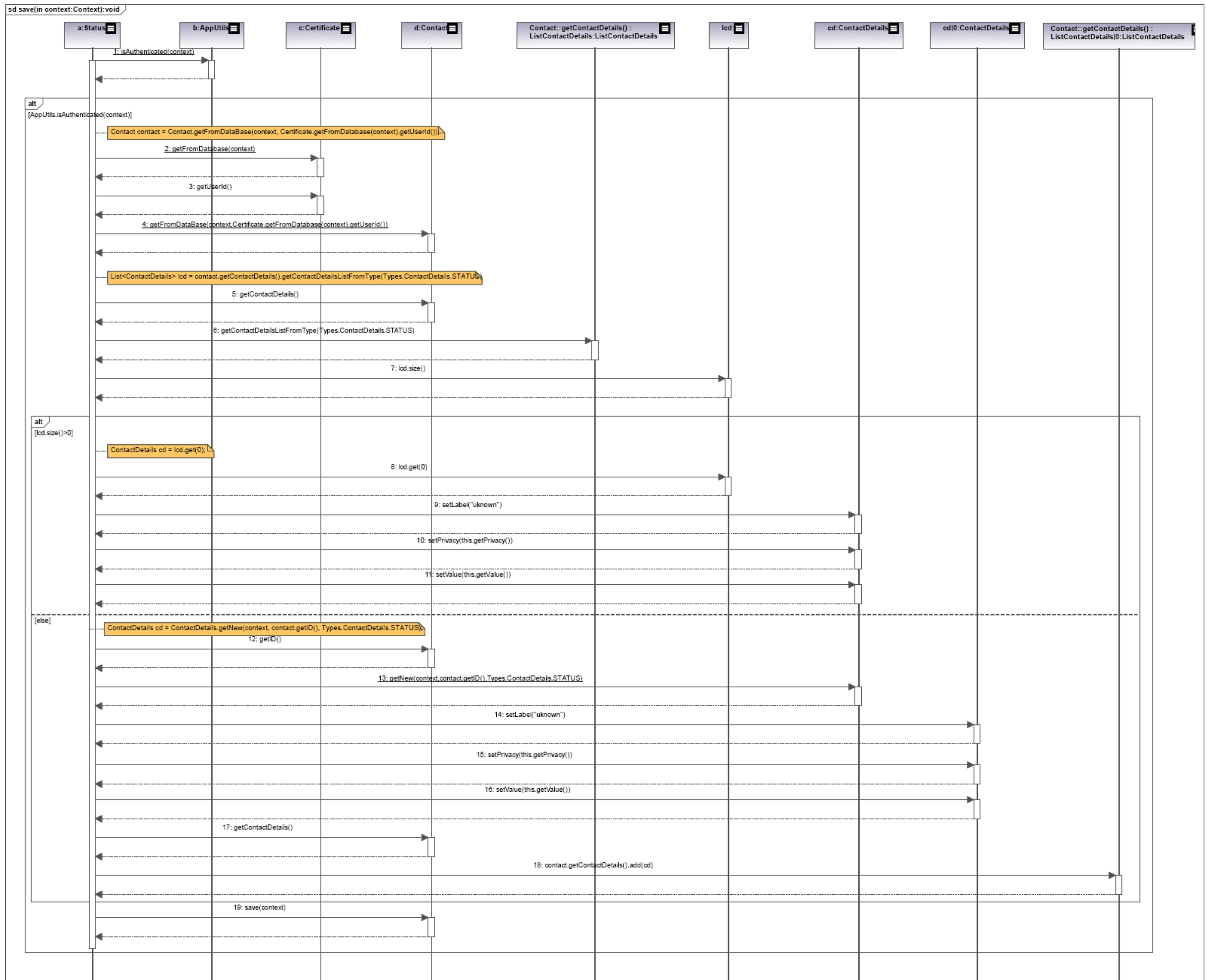
MicrophoneProcess – run()



Status – getStatus()

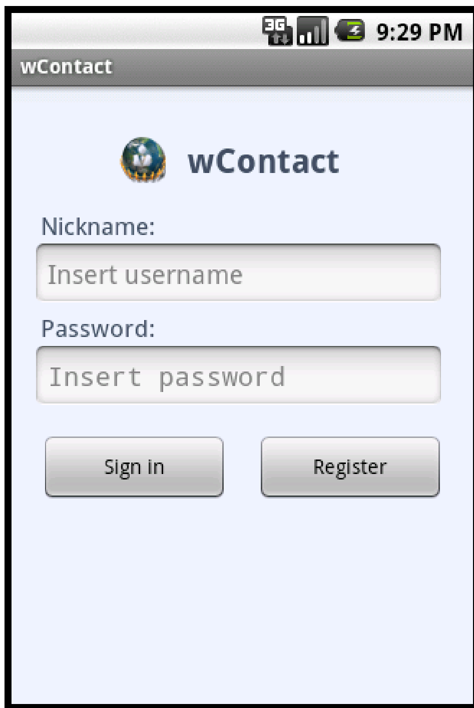


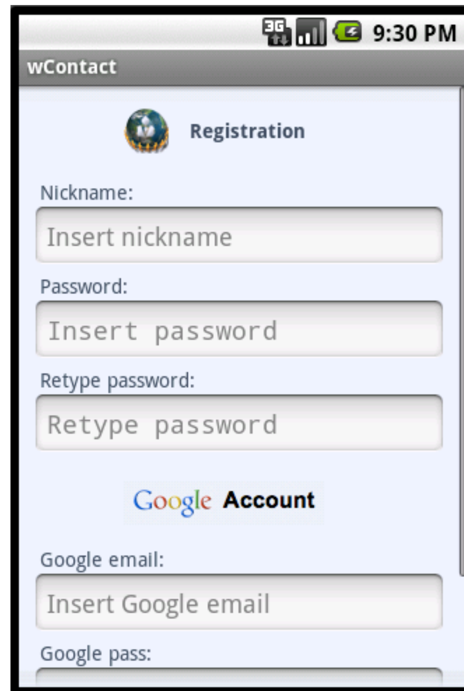
Status – save()

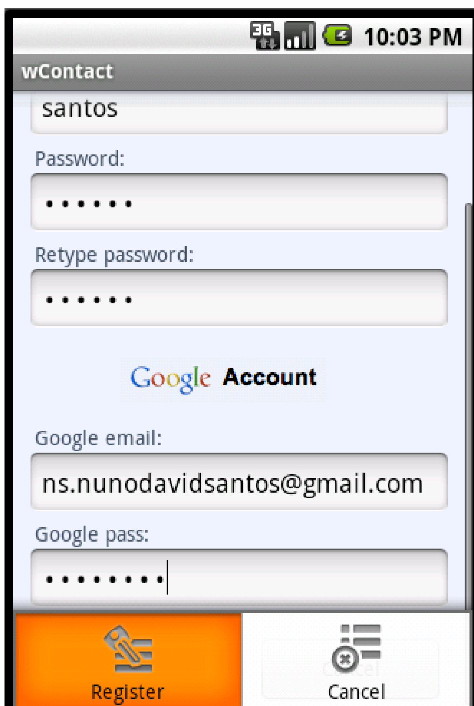


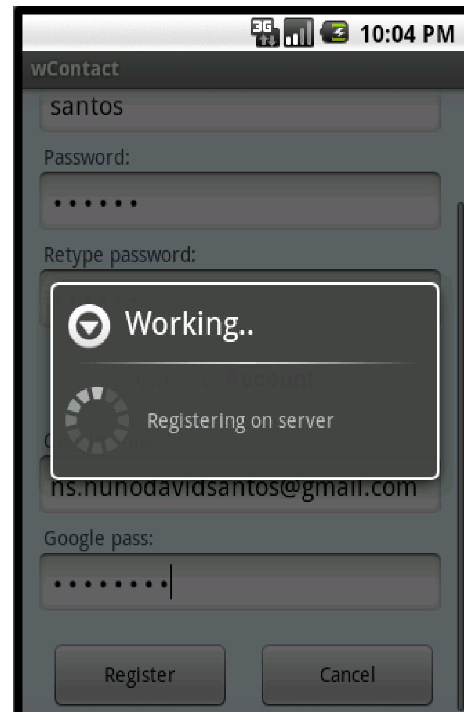
APPENDIX G

Application Screenshots







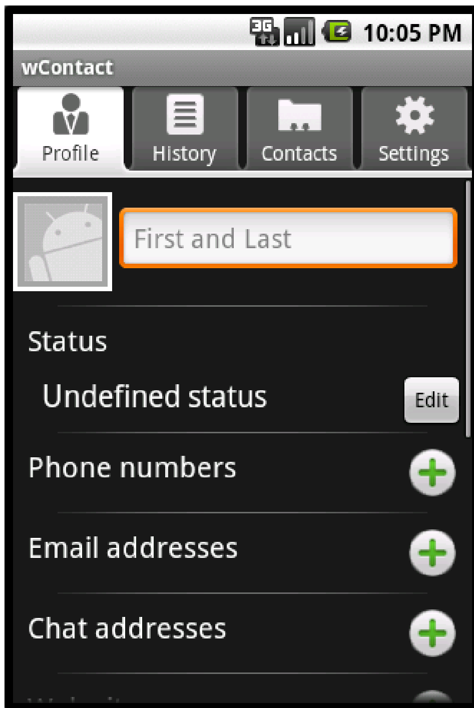


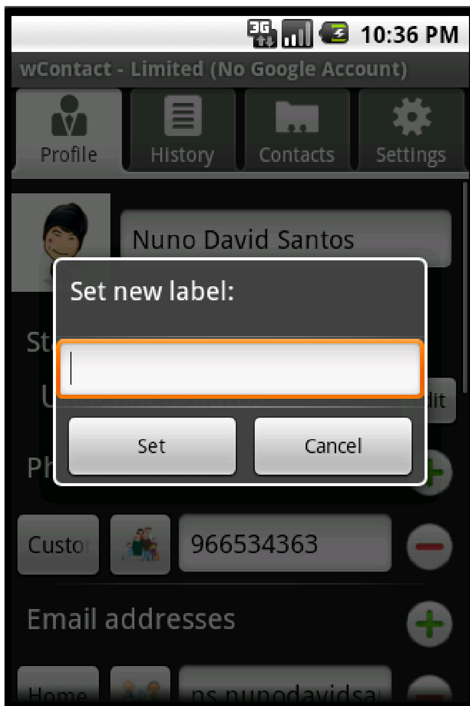
Top left: Authentication layout

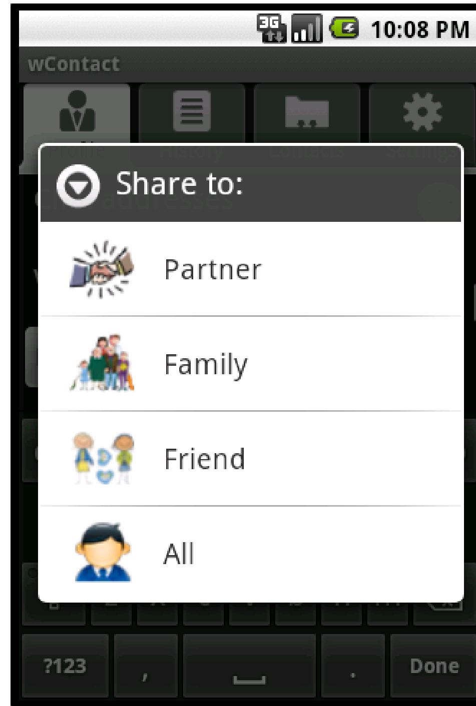
Top right: Registration layout

Bottom left: Submit registration

Bottom right: Process information





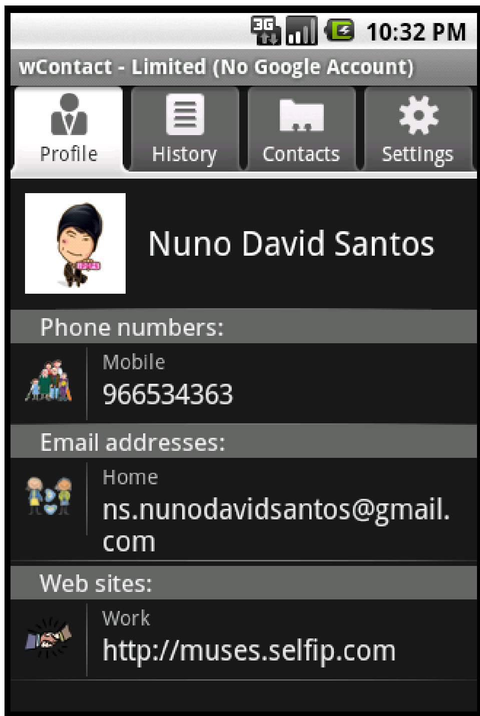


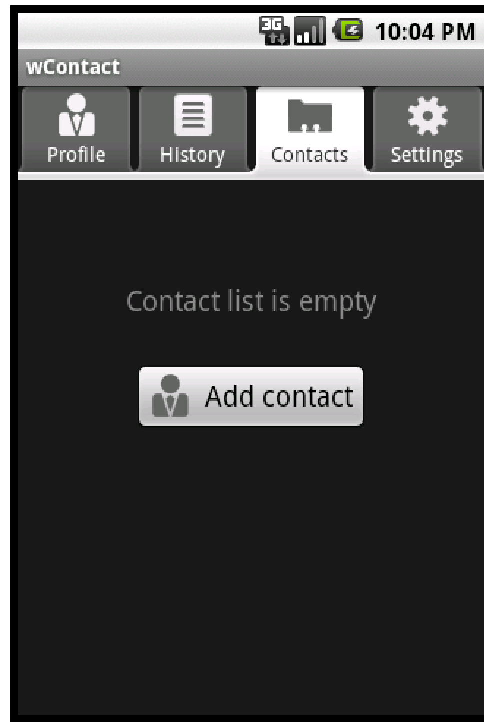
Top left: Edit profile layout

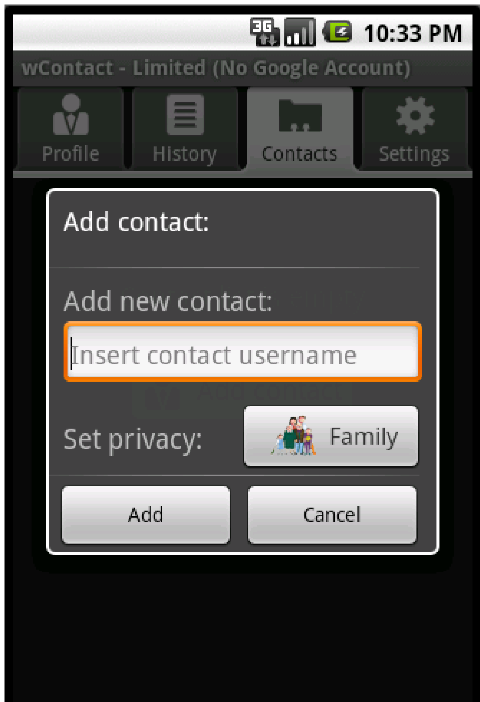
Top right: Pick avatar layout

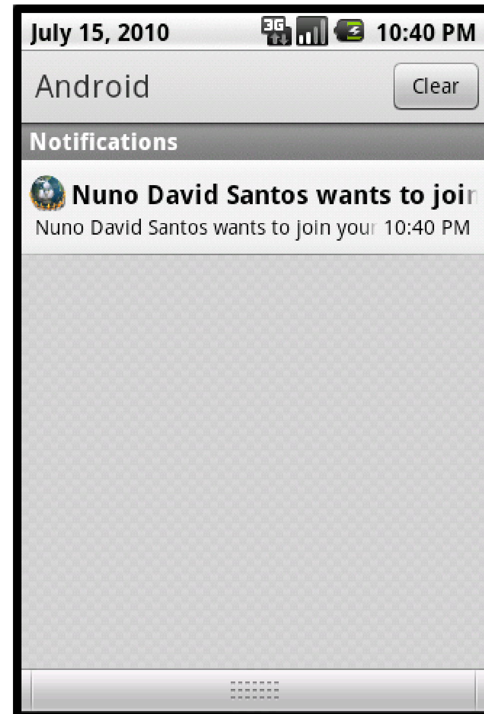
Bottom left: Set label for a contact detail

Bottom right: Pick privacy to share a specific contact detail







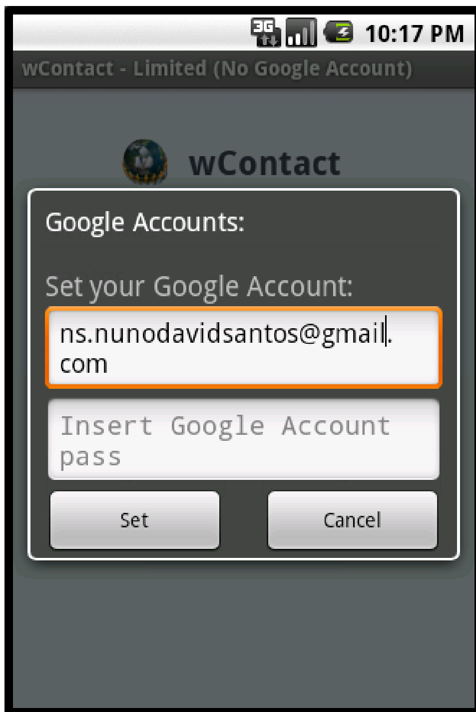


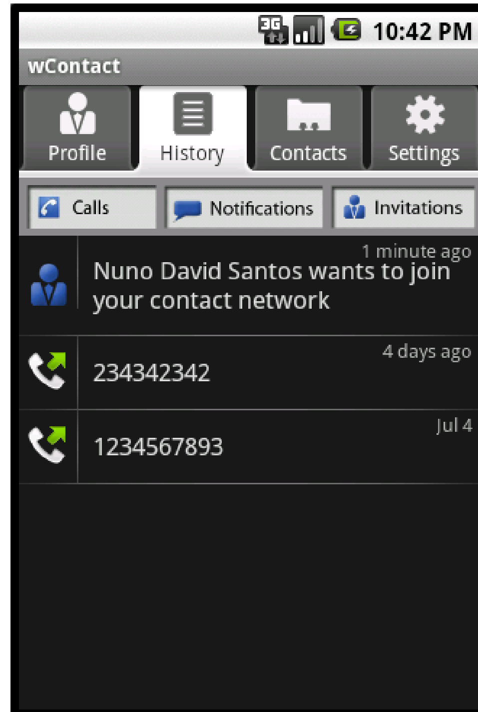
Top left: View profile layout

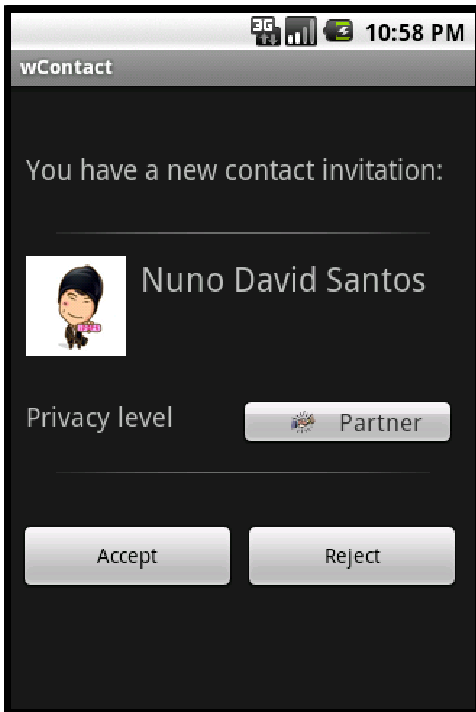
Top right: Empty contact list layout

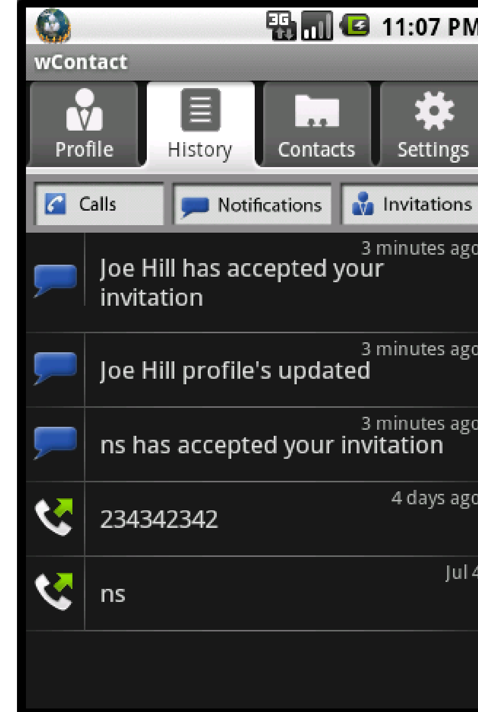
Bottom left: Add contact form

Bottom right: Notification information







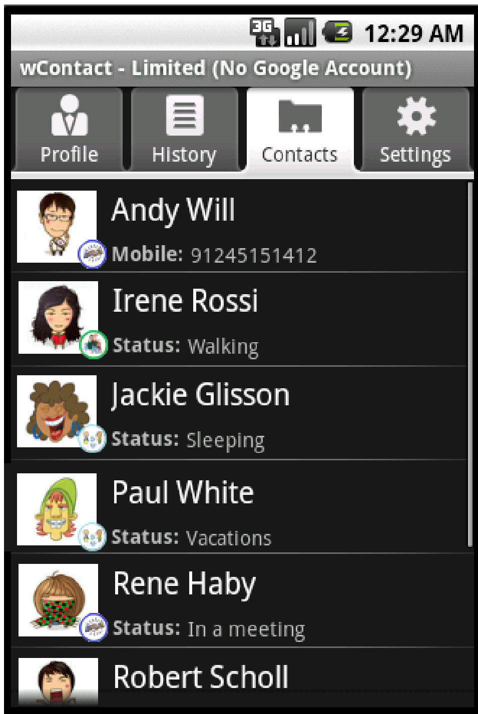


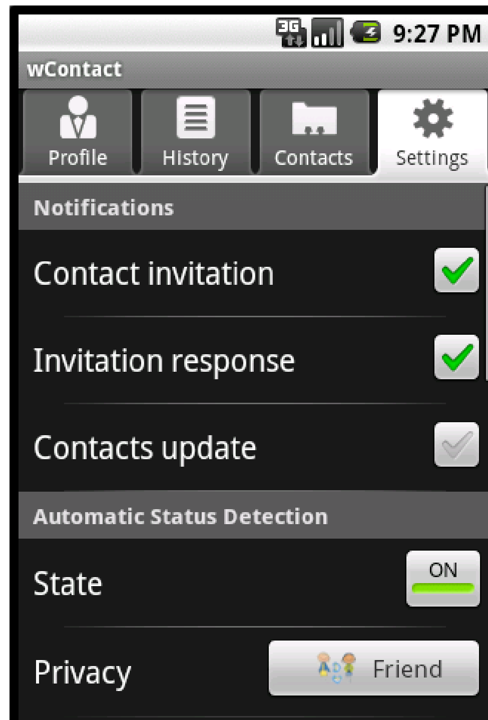
Top left: Setting Google Accounts for contacts synchronization

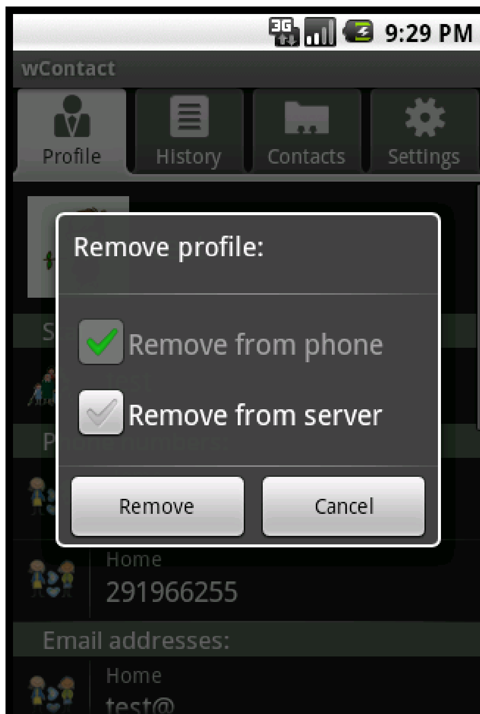
Top right: History layout

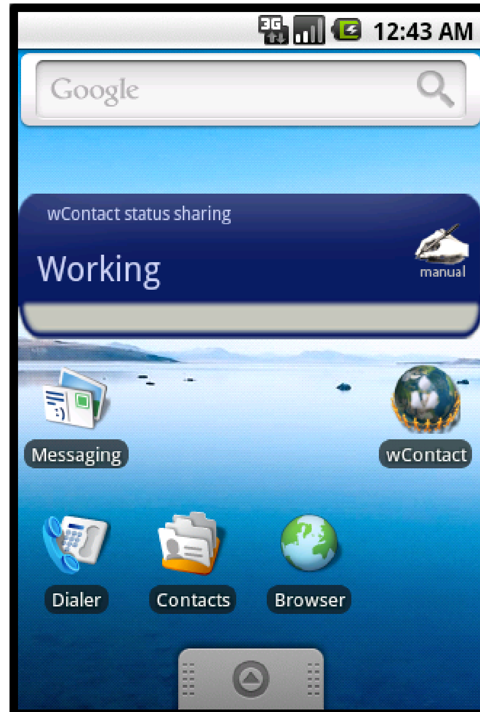
Bottom left: Contact response from a invitation

Bottom right: History layout







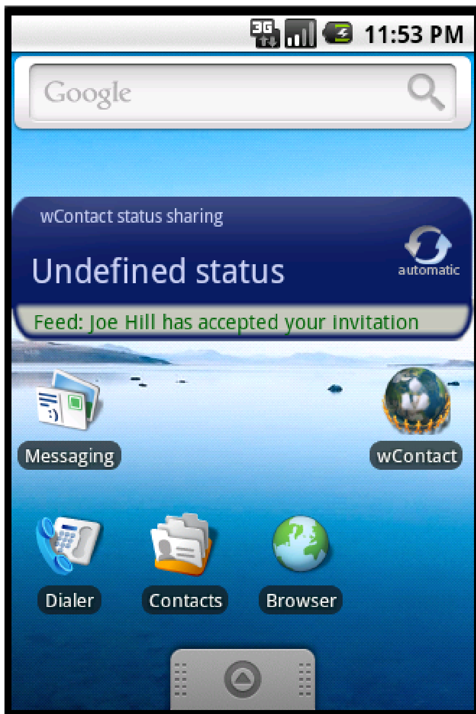


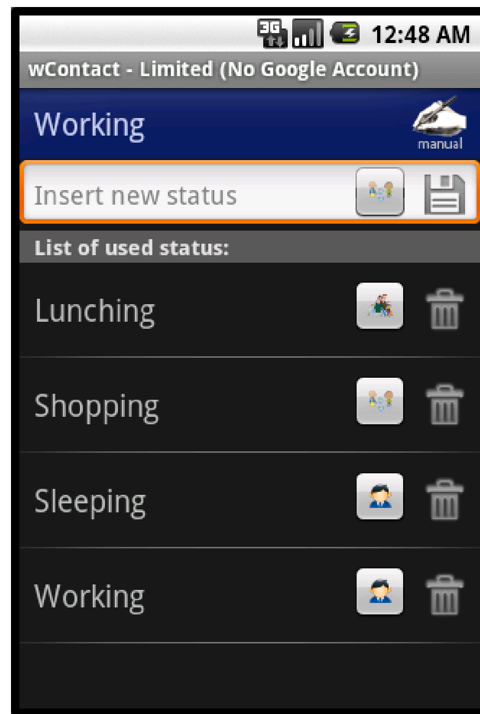
Top left: Contact list layout

Top right: Settings layout

Bottom left: Remove profile form

Bottom right: Home screen widget





Left: Home screen widget with feed information

Right: Selec status layout

APPENDIX H

User Research Survey

Personal information:

1- Age ____

2- Gender (male/female)

3- Job _____

Mobile phone usage:

1- Do you have a cell phone? (yes/no)

2- How many, in average, daily calls did you do?
-> 1 - 2
-> 2 - 5
-> 6 - 10
-> More then 10

3- Do you prefer sms or voice? (sms/voice)

4- In case of sms, name the reasons (ex: asynchronous, quick, do not interfere with other actions, etc.):
____
____

5- Can you tell me a example for a recently non welcome call? (ex. while was driving, while was jogging, when was piss off with my wife, etc.)
____
____
____

6- What is your source of calls: (1-not called 5-frequently called)
- friends
- family
- services
- work

7- What is for you the annoyest situation when making a call: (1-don't care 5-hate it)
- no answer
- busy signal
- sleepy answer
- annoyed answer

Improving wanted calls

1- What you feel about sharing a status, like IM? (1- useless 5-useful)

2- What is your first impression about using a status in your mobile phone?____
____

3- How wold you maintain status in your cell phone?
- Emphasizing based on what I do
- Daily
- Weekly
- whenever it occurs to me

4- What if your mobile device auto detects which situation are you in real time and update the status. Please set ehat do you think important to detect (1-insignificant 5-significant):
- Recently active
- Walking
- Driving
- Phone charging
- Still
- Engaged

5- Beside these ones, which other actions wold you like to be able to auto detect? (name at least three)

6- In case of having this application on your cell phone, would you:
- turn it on?
    - Because avoiding unwanted calls is a good thing
    - Because I like other people know what I do
    - Because will help other judgement before make a call
    - other:___________________________________
- turn it off?
    - I don't feel cozy knowing that other people might know what I'm doing in this right moment, although they don't know where and how.
    - Is totally unnecessary. Is just a call, I'll answer or not as I'm up to it
    - other: ___________________________________