YINK

Mobile Call Administration for Cisco Terminals

Department of Computer Science
Aalborg University
June 3rd 2013

AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East
http://www.cs.aau.dk

Title:

Yink — Mobile Call Administration for Cisco Terminals

Theme:

Mobile Systems

Project Term:

P8, Spring 2013

Project Group:

SW804f13

Students:

Bjarke Hesthaven Søndergaard
Esben Pilgaard Møller
Rasmus Steiniche
Thomas Kobber Panum

Supervisor:

Ivan Aaen

Copies: 6

Pages: 100

Finished: June 3rd, 2013.

Synopsis:

The use of phones is changing from landline phones to mobile phones. This change is not reflected in the IP PBX solutions available today. This report investigates how an extension can be developed to Cisco VoIP to reflect the change in phone usage. The solution, named Yink, is a scalable client-server system. The clients are mobile applications providing PBX functionality, and the server integrates the mobile phones with the existing Cisco VoIP system.

ABSTRACT

The use of phones is changing from landline phones to mobile phones. This change is not reflected in the IP PBX solutions available today. This report investigates how an extension can be developed to Cisco VoIP to reflect the change in phone usage. This extension requires the call administration features of the IP PBX solution to be available for mobile phones.

The agile development method XP is used during the development of the solution. The development method is adapted to suit the project and the team.

The project seeks to take advantage of mobile technology to provide a new approach to call administration.

The solution, named Yink, is a scalable client-server system. The clients are mobile applications providing PBX functionality, and the server integrates the mobile phones with the existing Cisco VoIP system.

It is not possible to make call administration mobile, as desired, due to limitations in Cisco VoIP. If the limitation is overcome, then Yink is a mobile call administration solution.

PREFACE

We would like to thank our supervisor Ivan Aaen for helping and supervising throughout the project. We would also like to thank Lytzen IT for proposing the project and being cooperative and helpful throughout the entire project. A big thanks to Søren Krøgaard for the front page illustrations, and Henrik Sørensen for the supervision in interaction design.

QUOTATIONS are the words of another person along with a source. The source of the citation will either be in the text immediately before or after, or could potentially be incorporated into the quote as shown in the example below:

> *" This is an example of a quotation. "*
>
> — X, p. Y

REFERENCES are references to sections, figures, code snippets, chapters or parts written elsewhere in the report. This could look like the following:

This is described in Section X.Y.

CODE EXAMPLES are written in a special environment so they are easy to read and recognize. Examples can be seen in Listing 1. Whenever there is a sequence of three dots ("...") in a listing, it means that we have omitted some content, which is not important in that specific context.

```
1  <?php
2
3  function print_numbers($from, $to) {
4
5      echo "Hello World! Here are the numbers from " . $from . " to " . $to
           ;
6      for($i = $from; $i <= $to; $i++) {
7          echo "\n" . $i;
8      }
9
10 }
11 print_numbers(1,100);
```

Listing 1: Code example of a hello world script written in PHP.

CONTENTS

ACRONYMS

VoIP Voice over IP

PSTN Public Switched Telephone Network

XP eXtreme Programming

TDD Test-Driven Development

JSON JavaScript Object Notation

XML eXtensible Markup Language

CallManager Cisco Unified Communications Manager

CallID Cisco Call Identifier

PBX Private Branch eXchange

IP PBX Internet Protocol Private Branch eXchange

Mobility Cisco Unified Mobility

GUI Graphical User Interface

UI User Interface

UUID Universally Unique IDentifier

TLS Transport Layer Security

DRY Don't Repeat Yourself

KISS Keep It Simple, Stupid

YAGNI You Ain't Gonna Need It

ER Entity Relationship

API Application Programming Interface

TCP/IP Transmission Control Protocol/Internet Protocol

C10k Concurrent ten thousand connections

JTAPI Java Telephony Application Programming Interface

DTMF Dual-Tone Multi-Frequency signaling

HTTP HyperText Transfer Protocol

ISDN Integrated Services Digital Network

1 INTRODUCTION

This report is based on a project proposal from Lytzen IT. The project proposal involves creating a mobile application for their Internet Protocol Private Branch eXchange (IP PBX) infrastructure. This chapter covers the motivation behind the project and presents the problem statement.

1.1 MOTIVATION

A Private Branch eXchange (PBX) is a telephone exchange used internally in an organization. It makes internal connections between the telephones and connects to the public telephone network through either a Public Switched Telephone Network (PSTN) line or an Integrated Services Digital Network (ISDN) line. PBX is a device, which the customer's ISDN lines are routed through, and as such all incoming calls are managed by one or more PBX devices and forwarded to the appropriate telephone terminal. This enable people within an organization to call each others' landline telephones without having to use the public telephone network. Lytzen IT is a business that sells PBX services and has noted a need for change in the services provided to their customers.

1.1.1 *Private Branch Exchange Devices*

Historically PBXs were managed by a PBX device, installed on-site at the customer. Traditional PBX solutions were expensive for the customer. The customer had to pay for the acquisition of the PBX device, which was expensive. The high price limited the potential customer base for the providers. While the solution provided the functionality needed at the time, it was not ideal for neither customer nor provider.

1.1.2 *Internet Protocol Private Branch Exchange*

As a result of advances in Internet and PBX technology, the move from a PBX to an IP PBX solution based on Voice over IP (VoIP) is now a reality. More companies are moving to IP PBX [47], [49].

This internal communication using VoIP is often done using desk phones. These desk phones are directly connected to the network via Ethernet. IP PBX solutions reduced the cost for the customer of acquiring a PBX solution. In an IP PBX the provider hosts the IP PBX services locally in a data center. The customers then rent the IP PBX function-

ality from the provider. Customers connect to the IP PBX through a PSTN line or via VoIP. The advantage of the IP PBX is that small companies can relatively cheaply acquire an IP PBX solution, if they have the required network infrastructure. If the network infrastructure is not good enough this will have to be acquired as well. Furthermore with all functionality hosted by the provider, it is easier and cheaper to maintain and update. At Lytzen IT the solution is based on the Cisco Unified Communications Manager (CallManager). Customers are provided Cisco Terminals and the network infrastructure needed to connect to the CallManager. Cisco Terminals are desk phones connected to the CallManager. Additional functionality is provided by this solution in the form of the ability to have calls forwarded to mobile phones from a Cisco Terminal.

The PBX and IP PBX solutions, while different in setups, provide the same functionality to the customers.

The IP PBX solution, Lytzen IT uses, has been in use for 6 years.

1.1.3 *Telephone Trends*

In general people are shifting towards using mobile phones as opposed to landline phones [20]. This has changed the general perception of how easy it should be to get in contact with a person. As an example it is expected that a salesman or technician is available for support regardless of whether he is sitting at his desk or is working out in the field. To handle this the CallManager contains a functionality called Cisco Unified Mobility (Mobility) [13]. If a client has enabled Mobility on his Cisco Terminal, the calls that are not answered within a set time frame, are automatically forwarded to a number he has specified, e.g. his mobile phone. This solution solves some of the issues that are associated with making PBXs mobile. Mobility does however not provide enough functionality to fully integrate PBXs with mobile phones. If a client with Mobility enabled answers a call on his mobile phone, he is unable redirect the call to a colleague in case it is needed. Therefore while Mobility makes it possible to have calls forwarded to a mobile phone, it is not possible to redirect the call to other phones within the PBX. Therefore there is a need for a solution that enables the client to have all the PBX functionality available on his mobile phone.

The functionality provided must reflect and incorporate the current technological trends. The use of mobile phones is increasing and thereby the use of smartphones. In 2012 the number of smartphones in use reached one billion, and it is estimated that in 2015 this number will be doubled [38]. This change towards smartphones should be reflected in the service provided by IP PBX providers.

To redirect a forwarded call from a mobile phone requires some form of communication with the CallManager. The user should per-

form this communication through an abstraction layer to simplify the use of the system. It is therefore interesting to investigate the idea of creating an application that provides the functionality to redirect calls forwarded from Mobility to either an internal number within the PBX, or to an external number outside the PBX.

1.2 PROBLEM DEFINITION

The problem investigated in this report is based on the idea of an application that connects mobile phones to existing PBX systems.

The solution will be called Yink, which is short for Your Link. Yink will consist of different components. Each component will be given a name. These components will be introduced in Chapter 4.

In order to formally define the problem its aspects must be examined. The problem is being investigated for Lytzen IT, and a solution will be delivered to them for maintenance and further development. This affects the solution, since it must be considered during development how to ease the process of passing on the software. Such consideration involves making the system extendable, if new features needs to be added, and making the code readable. Furthermore since the system is intended for use in production, testing techniques must be considered to ensure the solution is of a high enough quality.

Currently the solution serves more than 1,000 Cisco Terminals [5]. It is unknown whether or not this number will increase or decrease. Therefore it must be considered how to make the system scalable with regards to users. Scalability is defined as the system's ability to adapt to an increasing amount of users.

It must be considered how to account for the different types of smartphones users have. The solution should aim to accommodate all customers at Lytzen IT regardless of whether they have an *Android* based phone, an iPhone, or one based on a third platform. Portability is therefore important for Fex. A definition of portability is:

> " *A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment.* "
>
> — James D. Mooney [42]

The concretized definition used in this project is based on the definition above, and is: A system is portable across a set of mobile platforms, if the cost of porting and adapting it to a new mobile platform is less than the cost of redeveloping it [42]. A mobile platform is a mobile operating system for smartphones.

As mentioned in Section 1.1 the PBX system used by Lytzen IT is one based on the CallManager and a set of connected Cisco Terminal.

This limits the solution to one that can be integrated with the existing infrastructure at Lytzen IT.

From this it can be derived that the most important aspects is to investigate how to integrate the solution with the IP PBX solution in use at Lytzen IT. Furthermore it must be considered how to make the system maintainable, scalable, and extendable as it is handed over to Lytzen IT. Combining these aspects creates a formal definition of the problem, which can be seen in Section 1.3 as the problem statement.

1.3 PROBLEM STATEMENT

The problem definition in Section 1.2 lead to the following problem statement:

How can we develop a scalable, portable, and extendable mobile solution for call administration of Cisco VoIP telephones, while maintaining high quality in regards to the customer's needs?

From this problem statement these three subproblems are derived:

How can we use a development method to handle the uncertainty of developing a solution that extends an existing product?

How can we combine the domain knowledge of the customer and the technical expertise of the developers in order to create a suitable use context for the customer?

How can we select and use a programming language in order to fulfill the software qualities of the project?

1.4 PROBLEM DOMAIN

Based on the previous sections in this chapter a problem domain can be defined.

The initial problem posed by Lytzen IT is that they want their customers to be able to receive calls on their mobile phones and redirect these calls to coworkers. An example of this would be that a user of the company's IP PBX solution receives a call from either a coworker or an external caller on their mobile phone. The caller needs to speak to a coworker of the user, and therefore the user needs to forward the caller to their coworker, so the caller gets to speak with the coworker.

The users of Yink will be employees within companies, which have an IP PBX solution, using their mobile phones to receive calls, both from external sources and internal communication. These users should be provided with similar features to those of their Cisco Terminals,

so the change from Cisco Terminal to mobile phone does not have an impact on the user's ability to perform their work. These features should be; seeing the caller id and redirecting the call to another number. The features on the mobile phone might differentiate from those of the desk phone, but should aim to provide the same functionality albeit not in the exact same way.

It is unlikely to expect the users to memorize all the numbers they might wish to redirect a call to. Therefore there is a need in the application for contacts the users can add. Calls can be redirected to both internal and external numbers. An internal number is one found within the CallManager. Each internal number in the CallManager are associated with a user. These users can be used to add contacts for internal numbers. An employee with an internal number usually has an external number as well, which can easily be added to the employee's user in the CallManager.

To summarize, users of Lytzen IT's IP PBX solution should be able to use the call administration features of their Cisco Terminal on their mobile phone.

2

METHODS

This chapter covers the methodologies and tools used throughout the development process and the reasons for using these.

2.1 CHOICE OF DEVELOPMENT METHOD

When choosing a development method there are three primary categories: Traditional, Agile, and Lean. The focus of this evaluation will be traditional and agile based development methods. Lean based development methods are discarded based on the developers' lack of experience with them. This inexperience can lead to more risks in the project, which is preferred to be avoided. In order to determine which of the two remaining categories to use, a polar graph is created. The graph is shown in Figure 1.

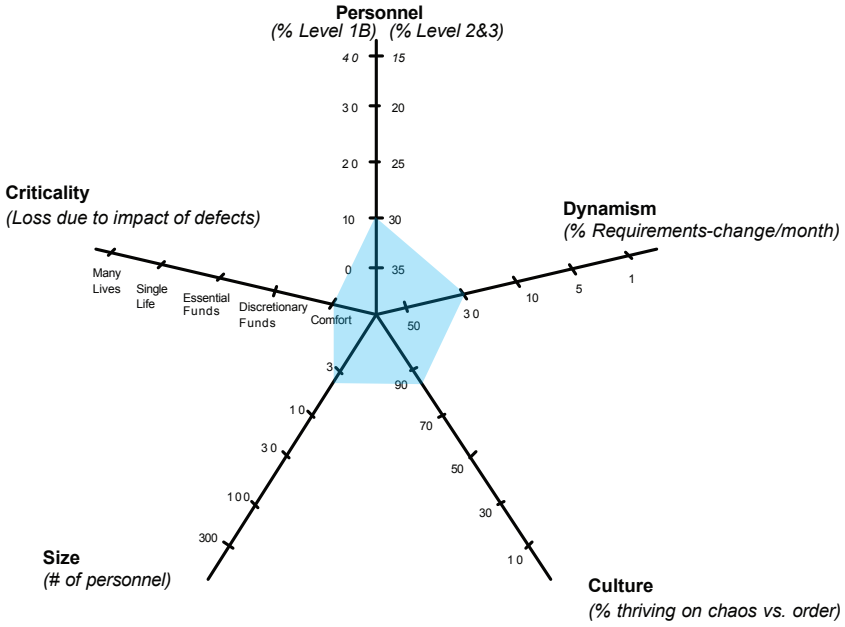


Figure 1: Polar Graph [10] of Project Profile.

The polar graph is created based on the development method selection tool proposed in the *Observations on Balancing Discipline and Agility* of Barry Boehm and Richard Turner [10]. Since the diamond has a low weight, i.e. the points being close to the center, this points towards an agile approach.

The agile development methods considered were eXtreme Programming (XP) and Scrum. Since the problem domain is unknown to the development team, there is a large amount of risks involved in the project. There is a risk of misunderstanding which problem Yink should aim to resolve. Furthermore since the solution is reliant on external resources such as the CallManager, a misunderstanding of its functionality can cause errors that propagate through the system. To accommodate Yink being passed on to Lytzen IT, measures must be taken to reduce the amount of defects in Yink. These risks will most likely result in changes to the design or implementation. These changes can be expensive in terms of resources, which in this context is man-hours [2, p. 5].

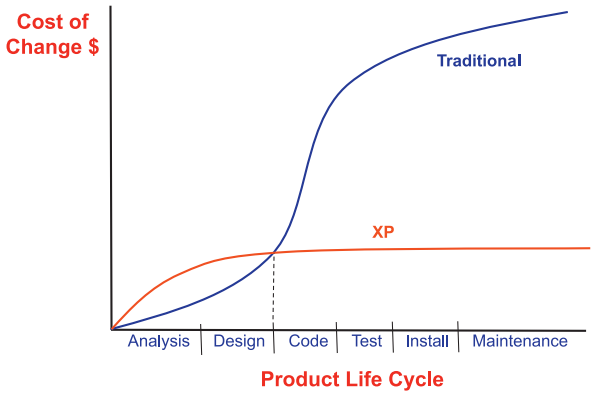


Figure 2: Cost of change: Traditional vs. XP. [2, p. 5]

XP prescribes engineering practices, such as *Pair Programming*, Test-Driven Development (TDD), and *Refactoring*, while Scrum does not [15]. This is valued important in regards to the evaluation. Based on the cost of change, seen in Figure 2, and the fact that XP prescribes engineering practices, the chosen development method is XP.

The value and the adaptation of each engineering practice of XP is described in Section 2.2.

2.2 EXTREME PROGRAMMING

This section is based on the books [7] and [9], and the article [37]. It explains the practices of XP and their adaption used in this project.

2.2.1 *Planning Game*

In the *Planning Game* the customer team and developers play a game to prioritize and estimate user stories.

The benefits of the *Planning Game* are that the customer team clearly states which user stories add the most business value. Each user story

is estimated by the developers according to the time needed to implement it. This estimate is used in planning the coming iteration.

After each iteration there will be meetings with Lytzen IT, where issues will be discussed and they will present what they find the most important to get done. This means there will be continuous reevaluation of what add the most business value. This furthermore means that Lytzen IT will be able to add, change, or remove requirements, as the software progresses.

The *Planning Game* as a whole will not be followed strictly. The customer will not be formally prioritizing requirements. These requirements will be prioritized by the developers based on the testimonies of the customer team and what the developers sees as the most risky to implement. This is done because the developers thrive on chaos in this scenario. Not being tied to a certain order of doing the tasks nor having a formal prioritizing means that the developers can pick what seems most risky or poses the largest threats.

During the development the software tool *Trello* [50] will be used. *Trello* is a non-strict management tool that emulates post-its on a wall. Tasks can be added to the wall and members can assign themselves to tasks. Using a management tool is different from what XP prescribes, since it adds management to the team.

But as *Trello* is a virtual implementation of a post-it wall it does not add unnecessary overhead, and is therefore not obstructing to use. *Trello* provides additional benefits such as being available from multiple locations, and reliability in the form of post-its not being lost.

2.2.2 *Test Driven Development*

TDD is writing tests before any code is written and is described by Kent Beck in [8].

The benefits of TDD are that the developer reflect over what code needs to be written, before they begin writing. When the written code passes the test, it is sufficient and can be minimized through *Refactoring*.

This project will use TDD by writing unit tests prior to code that is unit testable and acceptance tests prior to designing the Graphical User Interface (GUI). The acceptance tests for the GUI design is done to get the benefits of TDD for all development aspects of the project and not just code that can be unit tested.

2.2.3 *Whole Team*

The concept *Whole Team* describes how the entire team involved with the project needs to meet and communicate. This team consists of developers, customers, and other people who are involved.

The benefits of the *Whole Team* practice are: Team members get a better understanding of the problem domain, than if they were to research it individually, knowledge sharing is easier, and defects are caught earlier. Documentation of design is furthermore made less important since all team members involved know which decisions were made and why.

In this project it will not be possible to have the prescribed *Whole Team*, specifically the included on-site customer. However, it will be possible for this project to have a set of off-site customers reachable through *Skype* [48], telephone, or e-mail.

Some of the pitfalls with the on-site customer practice is described in this article [34]. One of these pitfalls are that a full-time on-site customer is not available. The solution recommended is "take what you can get" and fill the gaps with more communication.

Another pitfall is that one customer is not sufficient, e.g the customer is not qualified to answer technical questions. The solution recommended and used in this project is to have a customer team. This team will be composed of a technician and a manager. In this way the team has access to both technical, but also more general information about important features and the schedule of the project.

After each ended iteration the customer team and the team of developers meet, discuss the progress, and what the expectations of the next iteration are.

2.2.4 *Continuous Integration*

Continuous Integration is to continuously integrate components of a system. Several times a day, when components have been changed they should be integrated into the entire system, and the system should subsequently be tested.

Through *Continuous Integration* it is possible to find defects that are only related to system integration.

This project will use a continuous integration server called *Jenkins*. *Jenkins* is an open-source project [17]. It has plugins that provide tools such as running unit tests, console output from builds, code coverage, etc. *Jenkins* can furthermore access *Git* [11] repositories, which will be used in this project. This means *Jenkins* can pull when new commits are made and build moments after the commit. This allows developers to know if a component is faulty soon after having committed rather than the next day.

A component is faulty if the integration fails. A successful integration is when the entire system compiles and all tests pass.

There is currently no plan to handle failed integration attempts other than fixing the defects.

2.2.5 *Small Releases*

The team frequently releases well tested and running software in each iteration. It is important that the software is visible and given to the customer after each iteration in order to secure openness and tangibility.

Small Releases can help finding and fixing defects faster.

In this project there will be a meeting with the customers after each iteration, where the software developed during the iteration is shown and demoed.

2.2.6 *Coding Standard*

Coding Standard is used to ensure that all code in the system looks similar. Similarity in coding style helps enforce *Collective Code Ownership*, since developers cannot distinguish their code.

During the project there will be no *Coding Standard* defined in writing. The *Coding Standard* will be formed from having worked together over several years. There are furthermore only four developers, which means there are two teams when doing *Pair Programming*. These two teams work in close collaboration, thus the code will look similar.

The team will furthermore adhere to coding conventions of the programming languages chosen, such as how to name functions, variables, methods, classes, etc.

2.2.7 *Collective Code Ownership*

Collective Code Ownership means that nobody owns a specific piece of code, but rather that everybody is responsible for the code quality. This means that developers can improve any code at any time.

This ensures that every piece of code gets the benefit from many people's attention, which increases code quality and reduces defects.

Since the project consists of two *Pair Programming* teams in close collaboration it is natural to fix any defects, even those that occur in the other team's code. This project is furthermore conducted in an university environment, where every developer must know about all code in the project. This means everybody in the developer team is responsible for well written code.

Through this *Collective Code Ownership* it is also possible to interfere if a *Pair Programming* team is moving in the wrong direction with code.

2.2.8 *Simple Design*

Simple Design is the simplest representation of the functionality needed. Don't Repeat Yourself (DRY) [31], Keep It Simple, Stupid (KISS) [28], and You Ain't Gonna Need It (YAGNI) [36] are important principles in XP to ensure *Simple Design*.

The benefits of these three principles are that they aim to keep the design simple. *Simple Design* adds to the agility of the system, since a customer can change priorities without the system needing a complete overhaul. To ensure that the parallel development of similar components is possible, the design must be kept simple.

Instead of designing a large and complex system immediately, the project will incrementally expand the system design as more features are added. When making design decisions, the cost of changing the design must be considered. In cases where the cost of change is deemed low, the *Simple Design* practice should be used as proscribed. If the cost of changing a design decision is deemed high, an *Architectural Spike* [46] should be used to ensure a bad and costly design decision is not made.

2.2.9 *Refactoring*

Refactoring is about taking a problematic design and making it simpler. It aims to remove bad smells [6] from the code by changing it and is described by Martin Fowler in [22].

Refactoring makes the structure better, more readable, and it makes changes easier to implement. This in turn enables quicker development of code, due to the code being extensible and maintainable [22].

The project will consist of continuous *Refactoring*. If a problematic area is encountered the developers should refactor until it has been resolved.

2.2.10 *Pair Programming*

Pair Programming is the idea that any software should be build by pairs of two developers.

Pair Programming ensures that all production code is reviewed by at least one developer, which results in better design, code and tests. Pairing, in addition to better code, design and tests, also serves the purpose of knowledge sharing instead of writing documentation.

Pair Programming will be used for this project. Since the team of developers consist of four people there might be missing a person sometimes. This is solved in regards to *Pair Programming* by having the person work alone, and having the other team to consult in case of doubts. The work done by the person working alone will then be reviewed by the person who have been missing.

2.2.11 *System Metaphor*

System Metaphor is a simple description of the system. An ideal metaphor is one which is a simple evocative description of how the system works.

A simple *System Metaphor* makes it easier for the whole team to communicate what they are working on. It can, in some circumstances, help the developers on deciding object, function, and variable names.

The developers will be using a metaphor of a telephone switchboard. This metaphor is used when communicating with the customers.

2.2.12 *Sustainable Pace*

Sustainable Pace defines the pace with which XP developers work. The idea about *Sustainable Pace* is that developers should be able to work at the same pace indefinitely.

Sustainable Pace helps the developers to produce higher quality code. A high pace will drain developers for energy, resulting in demoralized developers. Therefore they will produce less code even when working more [52].

In order to keep people focused and happy the project will not have an enforced work schedule. People should meet every day at around 9 for daily stand-up meetings and would then be allowed to leave whenever they feel necessary. This can be when they finish a task, or feel burned out. Developers commit to a task, and it is their responsibility to finish it within a given time frame. It is their decision when they decide to work.

The defined pace for the project will be approximately 35 hours per week. This approximation is not a hard limit and can vary both ways, but should average at 35 hours per week.

2.3 OSMOTIC ENVIRONMENT

Osmotic Communication is a principle covering communication through an information flow in the background hearing of the developers. From this information flow they are able to pick up relevant information, and contribute to a discussion if they have relevant input.

Osmotic Environment aims to facilitate this background information flow. This usually done by placing all team members in the same room. The benefits of working in an *Osmotic Environment* is the ease of sharing information and having questions answered. As an example if a team member has a question they ask it to the room, and whoever has relevant input contributes to answering the question. Since the discussion happens in an open room all team members gain some form of knowledge by listening to the discussion. Decisions are made

through discussions. These discussions are exposed to all team members, and therefore the decisions involved are reviewed.

An *Osmotic Environment* is beneficial as a form of knowledge sharing for this project, and peer review for decisions. With two distinct components being developed having discussions about the individual components gives a form of knowledge sharing between the developers.

In this project an *Osmotic Environment* is facilitated in two ways. Many discussions are conducted around black boards, an open space is created around those, so all developers easily can enter into the discussion. Furthermore a large open space is created in the center of the room, so group discussions are conducted in an open space.

2.4 ITERATIONS

This section describes the iterations conducted during the development. Three components were developed over five iterations. The three components are introduced and described in Section 4.1. The components are: Yand, a mobile application for *Android*, Yios, a mobile application for *iOS*, and Fex, an interface server making the functionality of Yink available to the mobile applications. In Figure 3 the work done in the individual iterations on each component can be seen.

The first iteration was a combined start-up and development iteration. The development environment and tools were setup, such as the *Continuous Integration* server, *Jenkins*. The user stories were also defined in this iteration. In Yand an initial object model was designed and implemented. In Fex validation of requests was implemented using dummy responses.

With the basic functionality implemented in Yand and validation of requests implemented in Fex, the focus of the second iteration was enabling the two components to communicate. Support for Transport Layer Security (TLS) communication was added to both components to add security. In Yand the network communication was implemented using synchronous behavior. Yand furthermore had functionality for generating JavaScript Object Notation (JSON) requests for *redirect* and *update status* implemented.

When the communication between the components was implemented, authentication of users was added. Authentication functionality was added to Fex by integrating a database for storing users, and adding functionality for validating authentication requests from Yand. The authentication functionality is described in Section 4.2.3. Yand was extended to sent authentication requests through an authentication view.

Fex got extended with functionality for handling contacts, and requests with flexible parameters. Hawkeye and Java Telephony Ap-

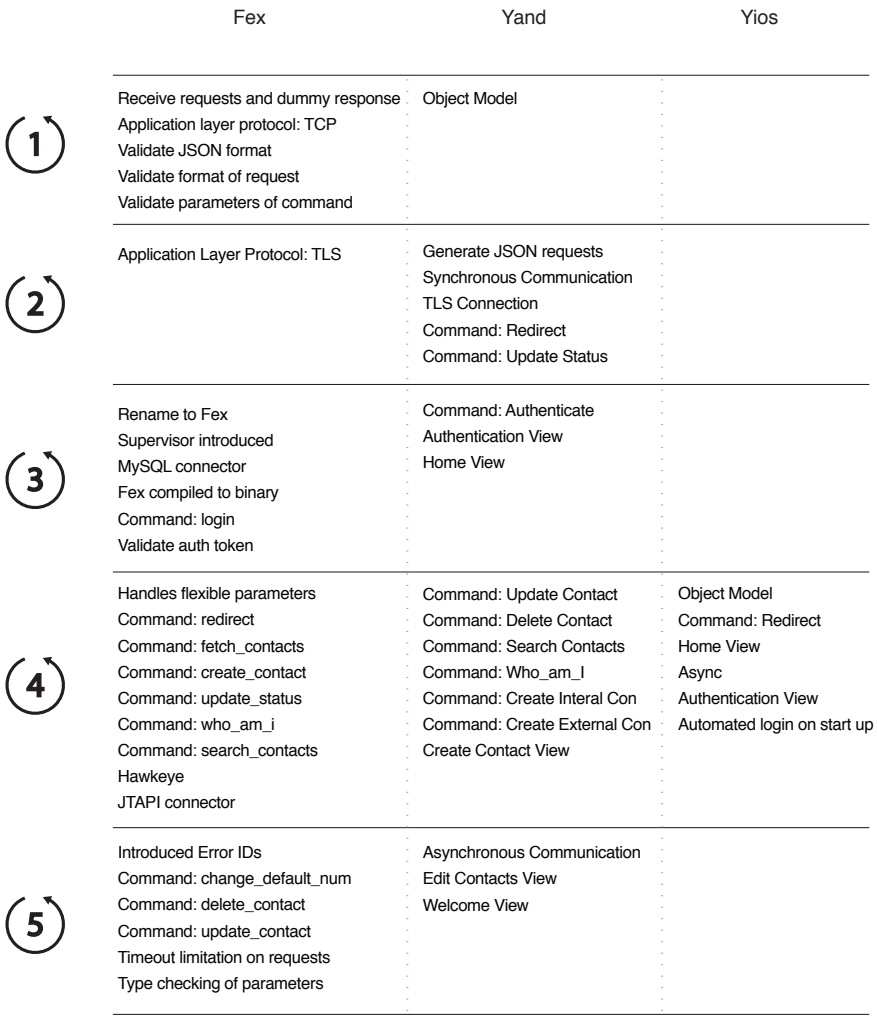


Figure 3: Overview of iterations.

plication Programming Interface (JTAPI), described in Section 4.2.1, were also introduced for tracking calls and supporting the redirect requests, respectively. The integrated solution using the database got extended to support contacts and statuses of users. Yand got extended to support the changes in Fex by adding GUI elements and functionality for finding and adding contacts, along with changing status. The developers started working on Yios in the fourth iteration. It was implemented with all the functionality of Yand. The implementation of Yios lead to a discovery of an underlying problem with the communication chosen. This was first solved in Yios before being added to Yand.

Error messages from Fex were important for debugging on the applications of Yink. Therefore these were implemented from the begin-

ning, but in the fifth iteration these were extended with error ids. Fex furthermore extended functionality for handling contacts, while Yand implemented elements in order to use this functionality. Yand received a GUI overhaul for creating, updating, and deleting commands of all types and had the asynchronous communication implemented in Yios added.

3 ANALYSIS

This chapter covers the user stories, along with an analysis of the project with regards to a set of software quality attributes.

3.1 USER STORIES

User stories is the format used by XP to express the requirements of a system. A user story is a sentence describing a feature which a customer needs.

Id	User Story
1	An employee can redirect a call to a coworker when receiving the call on their mobile phone
2	An employee can create an internal contact
3	An employee can create an external contact
4	An employee can delete a contact
5	An employee can edit the attributes of a contact, such as nickname and their default number
6	An employee can authenticate
7	An employee can view their contact list
8	An employee can redirect a call on their mobile phone to a contact
9	An employee can see which number is calling them
10	An employee can set their status, reflecting if they are available to receive calls
11	An employee can see their and others' status
12	An employee always has access to their contacts even when logging in from a new mobile phone
13	An employee is presented with the home screen if I recently logged in
14	An employee can search through internal contacts, either by name or number

Figure 4: User stories of Yink.

These have been derived from informal discussion during meetings with Lytzen IT, and then approved. This is not in accordance with the XP practice *Planning Game*, but this approach was deemed sufficient because the customer still defined the plan for which features to be implemented. The customer furthermore defined which features were the most important. The user stories seen in Figure 4, will be concerned with two primary functions; redirecting calls and managing contacts. The user stories are written according to the format defined in [14, ch. 2].

The user stories 9, 12, 13, and 14 are a result of discussions with the team prescribed by *Whole Team*. These discussions involve developers presenting ideas for how to use technical opportunities, which could improve the use context.

3.2 SOFTWARE QUALITIES

Software quality attribute are attributes a software system can be rated against before development begins. It serves as a method to identify key areas of a system, whose importance will affect design decision. This section is based on the quality attributes at [40, ch. 16]. The quality attributes with their rating can be seen in Figure 5. A reasoning for the ratings is given in the remainder of the section.

Quality Attribute	**Value**
Conceptual Integrity	5
Maintainability	4
Reusability	1
Availability	4
Interoperability	4
Manageability	1
Performance	2
Reliability	4
Scalability	4
Security	4
Supportability	3
Testability	2
Usability	3

Figure 5: Software Qualities

Conceptual Integrity — 5

Conceptual integrity is given the highest rating. As described in Section 1.2 the system will be handed over to Lytzen IT. The intention of having coherent code and design, as conceptual integrity proscribes, is to reduce the time an external person would spend learning the system. Furthermore as two applications are being developed, having coherence across them will ease the handing over process further.

Maintainability — 4

The rating of maintainability is connected to the rating of conceptual integrity, as a need may come after the completion of the project to add new features to Yink. This would have to be done by people not part of this project. To account for this maintainability is rated as a 4.

Reusability — 1

The system is a specialized system developed for a specific use context. Therefore reusability is rated as 1.

Availability — 4

The PBX functionality should be available to the users during their work hours. Therefore availability is important in Yink. Should the system crash and suffer down time, it will in most cases not stop the users from being able to do their work. Therefore it is rated as 4 instead of 5.

Interoperability — 4

Yink must communicate externally with the CallManager, and internally between the applications and Fex. The design must aim to make this communication efficient, both in terms of the amount of data exchanged, but also its formatting.

Manageability — 1

Lytzen IT has stated that manageability can be ignored initially.

Performance — 2

Performance is not an important attribute for a PBX system. It has been given the rating of 2, as some considerations will have to be

made to ensure it does not take an unreasonable amount of time to redirect a call.

Reliability — 4

Reliability is connected partially to availability. A high rating of reliability is important, since, if a user requests a redirect to a specific number, this redirect should occur. However as with availability, an error in the system will in most cases not hinder the user from doing their work, but only serve as an annoyance. Therefore it is rated as 4 instead of 5.

Scalability — 4

It is unknown how many users Yink might end up having. But as performance is given a low rating, making the system scalable is therefore not of the highest priority. Scalability is therefore given rating 4 as it is still important due to the unknown amount of users.

Security — 4

Yink contains functionality that can be abused by external people. The IP PBX functionality can be abused to make international calls at expensive rates. Security is therefore rated as 4 as considerations must be made to ensure Yink is not abused.

Supportability — 3

Supportability is rated as 3. Its rating is derived from the handing over of the software to Lytzen IT. Should errors occur, it should be possible to identify the source of the error through reasonable error messages.

Testability — 2

As the system is handed over it should not contain errors in its code. This is ensured through testing of the system. However as easing testing of the system should not affect other quality attributes, it is rated as 2.

Usability — 3

There is no general user of the system, therefore a specialized user interface cannot be designed. Usability is therefore rated as 3, as con-

sideration will have to be made, as to how the user interface can be made intuitive and easy to use.

4 DESIGN

This chapter covers the component structure of Yink, and the reason behind it. The design of each component in Yink will be covered. This involves design decisions, which are based on the user stories.

4.1 GENERAL ARCHITECTURE

Based on the user stories presented in Section 3.1, this section will discuss the architectural solutions considered, and evaluate upon them. The user stories requires a form of communication between mobile phones and the CallManager. The simplest solution considered to this, is to use Dual-Tone Multi-Frequency signaling (DTMF). The two most widely used mobile platform [27], *Android* and *iOS*, do not support the use of DTMF [1]. Lytzen IT deemed *Android* and *iOS* the most valuable platforms for Yink. Therefore the use of DTMF was discarded. This lead to the use of the mobile broadband. The use of mobile broadband supports many architectures. These architectures can be expensive to change and therefore an *Architechtural Spike* is used. The architectural solutions that will be described and evaluated are: Direct-connect, Peer-to-peer, and Client-interface.

4.1.1 *Direct-connect*

The direct-connect model is a self-conceived architecture. It builds upon every Cisco Terminal being connected to every mobile phone in the Cisco VoIP system. Using these connections users can redirect calls using the Cisco Terminals. This architecture is reliant on the mobile phones being able to send commands to the Cisco Terminals. This requires the Cisco Terminals to be available publicly on the network, which exposes them. This exposure can be seen as a security risk.

The architecture does not contain an administrative unit, therefore to fulfill the user stories, contacts must be administrated on each individual mobile phone. Therefore with this model it is only possible to share contacts across mobile phones. If the user wants to switch mobile phone they must add their contacts again. The communication form used in this architecture is simple.

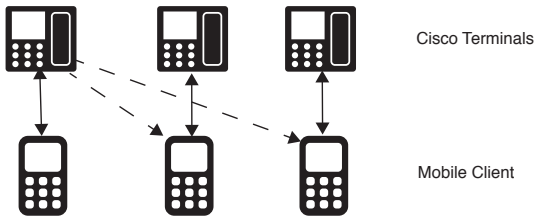


Figure 6: Direct connect architecture solution.

4.1.2 *Peer-to-peer*

The peer-to-peer model [35] supports one-to-one communication as each device can communicate with each other. This also means that every mobile phone can redirect to every other mobile in the network.

The tracker in this pattern keeps track of users' statuses, e.g. which users have active calls. Additionally this tracker allows the users to set their own status manually and see other users' statuses. This model is good for distributing work, in the form of computation, across all of the mobile phones, which is important since limited resources are available on these. Since the communication is handled across multiple mobile phones, this generates multiple points of vulnerability, that can be seen as a security risk. The communication form this architecture uses is complex, due to the communication flow not being predictable.

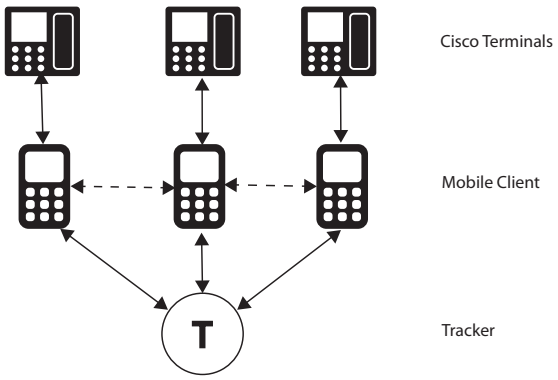


Figure 7: Peer-to-peer architecture solution.

4.1.3 *Client-interface*

The client-interface model is an adaptation of the client-server architecture [39]. It is a split architecture with two separate roles: Client and server. The clients in this context are the mobile phones running

an application and the server is an application servicing the mobile phones. In a mobile server-client setup, where resources are limited on the clients, it is preferred to move as much computation as possible to the server. Computation involves for example managing the contacts described in Section 3.1.

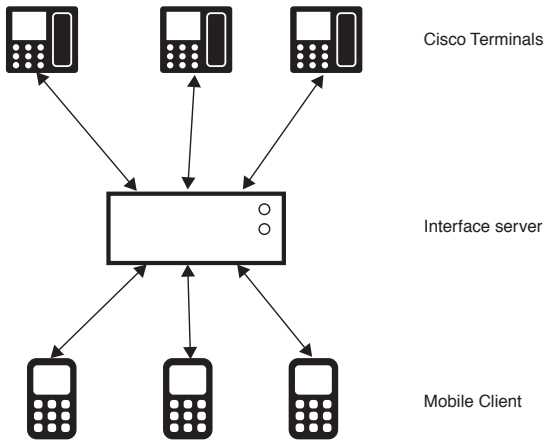


Figure 8: Client-interface architecture solution.

As seen in Figure 8 this solution builds upon a server, which enables communication between the clients and the Cisco Terminals. This interface server delivers a service in form of an Application Programming Interface (API). This API makes it possible for the clients to communicate with the PBX without knowing any of its functionality. One of the drawbacks of using a client-server architecture is that the more concurrent users a server has, the more resources it needs. Furthermore if the server ceases to function so will the mobile applications. One solution to both problems is to distribute the server, i.e. have more than one and make the underlying architecture support this. Since all communication is limited to the server, this means there is a single point of vulnerability. This one point of vulnerability means securing the server is critical, since if this is vulnerable then the entire system is vulnerable. Since there is a single point of vulnerability, this means enhancing security on this one point, enhances the entire system's security. The communication form in this architecture is complex, but the communication flow is predictable.

4.1.4 *Choice of Solution*

The choice of solution is based on the software qualities, described in Section 3.2. The primary software qualities considered is security and scalability, which is valued 4 and 4 respectively. *Direct-connect* is deemed insufficient in regards to security, and thereby discarded.

Peer-to-peer allows for interconnection between the mobile phones, meaning they can exchange information. However, *Peer-to-peer* suffers from the same security issue as *Direct-connect*. *Peer-to-peer* enables the use of multiple mobile phones for computing, which enhances scalability. The *Client-interface* solution allows for better security than the two other solutions considered, while maintaining the same level of scalability as *Peer-to-peer* since there can be multple interface severs.

The chosen solution is *Client-interface*.

4.1.5 *Yink Client Design*

Yink is classified as a firm real-time system [29]. In firm real-time systems results will be irrelevant after their deadline and this may degrade the quality of services delivered by Yink. In Yink an example is redirecting active calls. If the call is not redirected in a reasonable amount of time, the quality of this service is degraded or in the worst case non-existing. This is due to either the call being redirected so slowly that the person being redirected thinks an error has occurred, or the redirect does not happen.

The design approach for the Yink clients must be able to represent real-time data from the Cisco VoIP system. Two clients for mobile phones will be developed for Yink. One for the *Android*- and the *iOS* platform, which are called Yand and Yios respectively.

Using *Simple Design* the choice is to develop platform specific applications for Yink.

As seen in Section 3.2 maintainability and interoperability are important attributes for Yink. When designing two native applications two different approach are available: Generic design or independent design. A generic design states that the applications should have the same design, both in terms of classes, method names, GUI and behavior. Having a generic design will increase maintainability on the Yink clients. Since the developers can reuse knowledge of the design across multiple inherited generic designs. A generic design will make changing platform easier for the user, but the drawback is that the GUI will not be specialized for each platform. A completely generic design is not obtainable, therefore this will lead to inheritances of the generic design.

Another aspect to consider is the application store guidelines [33], [4]. These guidelines must be adhered to for the applications to be approved. Therefore these guidelines are the delimiter for the generic design, since all decisions must be within these guidelines. This could potentially mean that some GUI elements would not be utilized.

In Yink a generic design approach is used. The iOS design guidelines are the most strict and therefore these are deemed the lowest common denominator for the GUI. The decision is based on the con-

siderations described in Section 1.2. The software is to be handed over for maintenance and further development, and this process is eased with a generic design.

4.2 FEX

This section will cover the server component of Yink described in Section 4.1. Fex is a polling based information provider for the Yink clients. It serves as a link between the external resources, such as the Cisco Terminals and the Yink clients.

Fex should provide the functionality, shown in Figure 9, to the Yink clients

Id	Functionality	User Stories
1	Authenticate the user	6
2	Redirecting a call	1 and 8
3	Add a contact to the user	2 and 3
4	Delete a contact from the user	4
5	Get all contacts of the user	7
6	Update the user's contacts	5
7	Search through users by name or numbers	14
8	Update the user's status, stating if available or unavailable	10

Figure 9: The functionality of Fex.

Since there can be at least 1,000 Yink clients, as described in Section 1.2, Fex must be able to handle this amount of connections simultaneously. It is assumed that a user can only talk on one line at a time and that the user can be reached through multiple lines. From this it can be deduced that once a user's line is used, then the user is unavailable for further calls. A line can be either an IP PBX- or an ISDN address, e.g. *+45 96 23 03 60*.

Creating and having this amount of simultaneous connections can be computational intensive. The computational intensity can lead to misbehavior in the environment these connections exist in. Misbehavior can be a Yink client not receiving a response to its request. In order to adapt to this potential demand of computation, a model that allows maximizing the utility of modern computation structures is used. Modern computation structures refers to e.g. machines with multi-core CPU's and distributed computation across several CPU's. These structures rely on parallelism. To adapt to parallelism, a model that allows for isolated computation, i.e. computation that is not de-

pendent on or shares resources with other computational units, is used to model Fex.

4.2.1 *Architecture*

This section will cover the architecture of Fex, and how it has changed throughout the iterations by incremental design. The reasoning behind the changes will be presented along with the result of these changes and how they cause new functional dependencies.

The architectural structure is expressed in terms of processes, their communication, and relationships. The notation used can be seen in Figure 10.

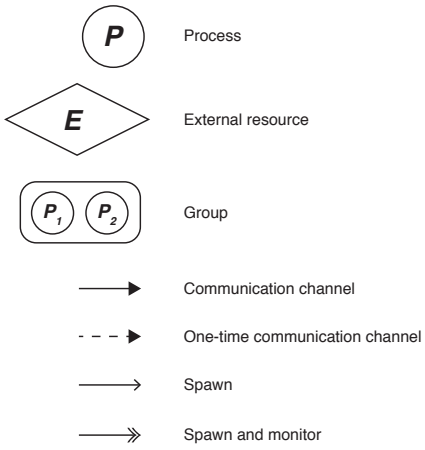


Figure 10: Terminology of Process Models.

A process represents a computational task that runs isolated. An external resource is an isolated component that appears as a black box to the model.

Communication channels represent a flow of data from a source process to a destination process. A one-time communication channel is equivalent to a communication channel, except it terminates upon data reaching the destination process.

A group is an abstract representation of a collection of processes. All communication channels going to and from a group have been labeled. This is done to explicitly state which communication channel belongs to which process when the group is expanded. If a label is used more than once in the expanded view this should be noted as separate communication channels, but to the same source or destination process.

A spawning process is the creation of a new process. This action consists of a parent and a new process, referred to as the child. The parent is the source and the child is the destination of the spawn arrow. Spawn and monitor refers to spawning a process along with

monitoring it. The child is monitored by the parent, and the parent receives information regarding the child. This means that if a child terminates the parent is notified and can perform an appropriate action, such as respawning the process, if it was an unexpected termination.

The component architecture selected in Section 4.1 defines the dependencies required by external components of Fex. These dependencies are the core of the system, if the other components are not able to get data from Fex, then no information is provided to the Yink clients. It is therefore deemed important to concretize these dependencies, both in terms of information required by external components, but also the communication protocol. This is done by making an initial version of Fex, that *faked* internal functionality. Faked in the sense that it statically emulates computation of the request. The purpose of this version is to clarify the communication protocol, and initial requirements to information by external components. The version is referred to as the *API server*, and its purpose is to allow external components to simulate communication with Fex.

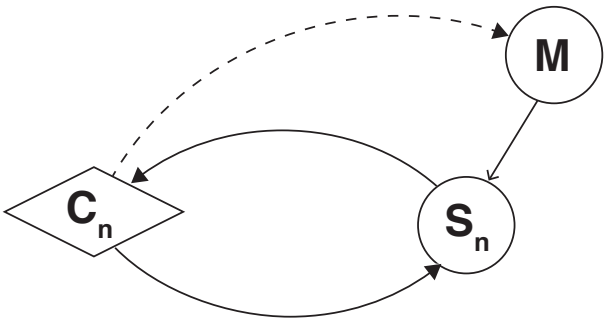


Figure 11: Process model of the initial version of Fex, the *API server*.

The process model of *API server* architecture can be seen in Figure 11. C_n represents a Yink client. M represents a master communication handler, that spawns slaves for each initiated connection. Each slave, S_n, handles all communication with the corresponding C_n. Each S_n responds to requests from the connected C_n by a static set of rules. The structure of these requests can be seen in Section 4.2.4.

In order to add more functionality to Fex, the architecture should be restructured. This restructuring is based on the fact that adding functionality to the current architecture would result in overloading the components with functionality. This would conflict with the use of a *System Metaphor*, since the naming of each part of Fex would be too abstract. Refactoring is used to rename existing processes and create a new separation of logic in the process model. This lead to the second iteration of the architecture, as shown by the process model in Figure 12.

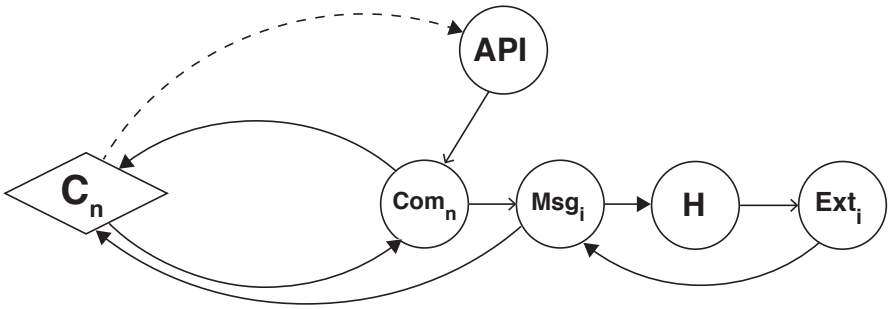


Figure 12: Process model of the second iteration of Fex, the *Reliability server*.

This version of Fex is referred to as the *Reliability server*. The process M from the *API server* has been renamed to API. S_n has been removed, and replaced by a process named Com_n.

Com_n represents the communicator process, which handles the verification of type and the Yink protocol. The Yink protocol is described in Section 4.2.4. If the verification rejects, information is sent back to C_n regarding the reasoning for the rejection. When the verification accepts the data received, it is considered a well-formed request and Com_n then spawns a Msg_i process.

Msg_i represents a message process, which handles the initial action of performing an external task and the responding to C_n.

An external task performs data retrieval from external resources. In order to create an external task a request is sent to H. H is a handler process that spawns and monitors the Ext_i processes. Monitoring consists of respawning processes that terminate unexpectedly within the timeout. The request sent to H contains information about which type of external task that should be run.

Depending on the result of the request to H, information is sent to C_n. If the result of the request is successful, the data of the result is sent to C_n. However, if the result of the request is not successful or a timeout occurs, an error message explaining the problem is sent to C_n. Timeout occurs when Ext_i does not respond within a specified time frame.

Ext_i represents an external task process, which contains functionality to connect to external resources and interact with them.

The *Reliability server* created the fundamental architecture for ensuring reliability. However, availability has not been focused on when designing the architecture of the *Reliability server*. Availability is rated 4 as a software quality, as described in Section 3.2. This means availability is a relatively high priority when designing the architecture. The architecture of the *Reliability server* furthermore does not provide a communication layer for external resources. These two areas are covered in the final architecture seen in Figure 13.

In order to increase availability a supervisor, $Sup*$, is introduced. The supervisor spawns processes within the processes *Request Con-*

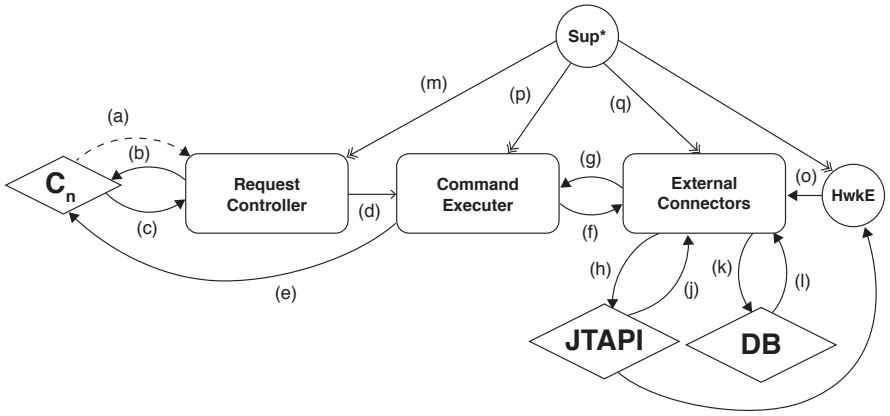


Figure 13: Process model of the final iteration of Fex.

troller, *Command Executor*, *External Connectors*, and HwkE and monitors them. The monitoring is an abstraction layer that helps to provide availability through respawning in case of unexpected terminations.

The group *Request Controller* handles the initial connection and handshake, and receives requests from the Yink client, C_n. *Request Controller* also verifies if the sent request is of a valid format. If the request is not of a valid format it responds to the Yink client without forwarding the request to the *Command Executor* group.

The *External Connectors* group is a communication layer to JTAPI and DB. The JTAPI connector will be referred to as JTAPI server. It establishes connections to the external resources, and provides an internal interface to interact with these external resources.

Command Executor is the group that handles all requests and sends the result of a request directly to the Yink client. If a request requires external resources it interacts with *External Connectors*.

The external resource JTAPI is an API that allows for controlling Cisco Terminals.

DB is an external database resource, which allows for storing information. This resource works as a communication layer between the event based input from JTAPI server and the polling based communication between *Request Controller* and C_n. This is covered in Section 4.2.6.

The process HwkE handles event signals from the external resource JTAPI server, and is described in further detail in Section 4.2.5.

The group *Request Controller* consists of the processes Com_n and API, as seen in Figure 14. All communication channels to, from, and within this process was already described when describing the *Reliability server*.

Command Executor is the collection of Msg_i, H, and Ext_i, and can be seen in Figure 15. The communication channels between *Command Executor* and *Request Controller* were previously explained as well as

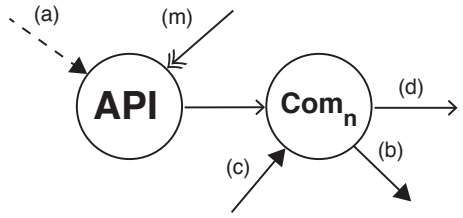


Figure 14: The *Request Controller* group.

communication within the group. However, *Command Executor* has communication channels with a new group of processes introduced in the final architecture of Fex, i.e. the *External Connectors* group.

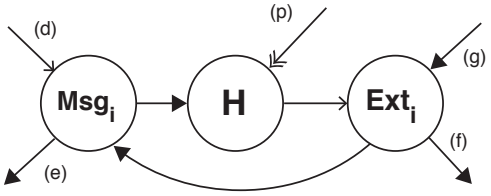


Figure 15: The *Command Executor* group.

The *External Connectors* group consists of `JtiCon` and `SqlCon`, as seen in Figure 16. The process `JtiCon` and `SqlCon` are both communication layers to JTAPI server and DB respectively. They both establish a connection with their external resource, and provide an internal interface to interact with these external resources.

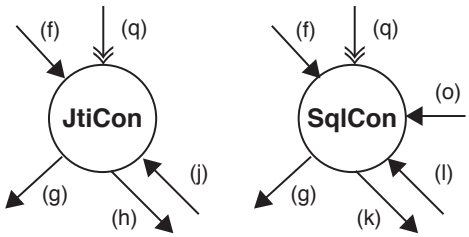


Figure 16: The *External Connectors* group.

The full version of the final architecture, without groups, can be found in Appendix C.

4.2.2 *Application Layer Protocol*

This section describes the application layer protocol used by Fex, how it changed throughout the iterations, and the reasonings for these changes.

The initial protocol is Transmission Control Protocol/Internet Protocol (TCP/IP), due to its guarantee that all data sent reach its destination, as long as a connection is established. The protocol enables any type of data to be sent, as it transfers bytes. This choice was made due to the simple design practice of XP, described in Section 2.2.

However, TCP/IP does not provide any form of security. The data packages sent by TCP/IP contain raw bytes. This can potentially allow unauthorized access to private data, which is a security issue. Security is rated as 4, as described in Section 3.2, and therefore decisions have to be made to increase security.

An extension of TCP/IP is TLS [19], that provides data encryption. TLS version 1.2 guarantees prevention of unintended access to the data transferred. An additional security feature provided by TLS is verification of the peers of a connection. This feature can be used for authentication.

4.2.3 *Authentication*

This is based of user story 6, and describes the considerations made with regards to authentication of users. Authentication is the ability to validate users' identity by distinguishing them from malicious sources. A malicious source is an entity trying to access data unintended for that entity, e.g. a hacker trying to gain access by emulating a user's identity. Distinguishing between users and malicious sources means, if data is sent to X, then X is expected to be the only recipient of this data. This principle will be referred to as destination integrity. Furthermore if data is received from Y, then Y is expected to be only source of the data. This principle will be referred to as source validation. Achieving destination integrity and source validation in a system is a complex task, due to the natural behavior of network communication. Both principles are concerned with communication, and therefore connected to all components of the system. Making changes to the communication can have severe consequences and these consequences can be expensive to correct. To avoid this, an *Architectural Spike* is used to determine how to achieve these principles.

The following approaches are considered: distributed TLS keys, TLS key distribution server, TLS using an authentication token, and TLS using user credentials. The TLS key approaches are designed specifically for Yink, but make use of the functionality provided by TLS described in [32].

Distributed TLS Keys

A distributed TLS key system is based on the peer verification described in Section 4.2.2. The peer verification is done using keys provided by TLS for client authentication. To achieve this Fex must dis-

tribute a unique key to each Yink client that is installed by a user. The Yink client will then use this key to authenticate all communication with Fex. With each user having their own unique TLS key the principles of destination integrity and source validation are upheld. Destination integrity is upheld, as only the Yink client with valid key and Fex can decrypt the data sent via the TLS connection, and a valid key is required to initiate a connection with Fex. Source validation is upheld as each Yink client is given a unique key, and they can therefore be distinguished between. The concept of this approach is illustrated in Figure 17. Using a distributed TLS does however, require that Fex is able to distribute these keys through the application store the Yink client is installed from.

Figure 17: Communication between app and server with distributed unique key.

TLS *Key Distribution Server*

TLS key distribution server is an extension of the Distributed TLS Key approach. Instead of distributing the TLS key with the Yink client through the application store, each Yink client must interact with a key distribution server to retrieve a key. This interaction involves a method to validate the user's identity. The key distribution server provides a unique TLS key to each Yink client which is then used to authenticate all communication with Fex. Fex cannot serve as a key distribution server, as the key is needed to initiate any form of communication with Fex. The key distribution server must therefore be an additional server besides Fex. The approach is illustrated in Figure 18

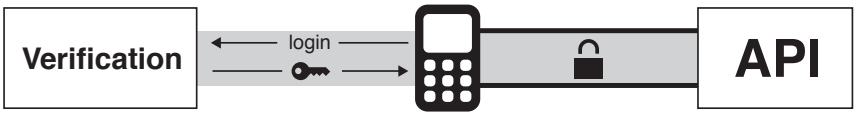


Figure 18: Communication between app and server with distributed key server.

TLS *with User Credentials*

This approach uses TLS to encrypt the data sent. In this approach the Yink client sends user credentials as a part of every request. User

credentials consist of username and password, which together represents a user identity. This approach requires the Yink client to store the user credentials on the device, to avoid having the user enter their user credentials for each request. The security of each Yink client is unknown, it can therefore pose a security threat to use this approach. This approach relies on the users' ability to remember their user credentials. To make this more fault tolerant would require an administration tool for managing the user credentials of Yink's users.

TLS with an Authentication Token

This approach is an extension of the *TLS with User Credentials* approach. The authentication token approach's structure is designed specifically for Yink. The administration tool mentioned in the *TLS with User Credentials* approach, is not represented in the user stories. It is therefore assumed to be an external component Fex containing and administrating the user credentials of Yink's users. In order to reduce the amount of authentication request to the external component, the approach with an authentication token is used. The concept of this approach, illustrated in Figure 19, is for users to authenticate themselves using their user credentials when they first initiate a connection with Fex. Fex validates the user credentials against an administrative component, and if the validation is successful the user credentials are stored with an authentication token. An authentication token is a Universally Unique IDentifier (UUID). The security threat mentioned in the *TLS with User Credentials* approach have moved from the Yink client to the external database resource. This creates only one point of vulnerability, instead of each Yink client being a security risk, with regards to user credentials. In any subsequent requests made by the Yink client the authentication token is sent along, ensuring the principle source destination is upheld. This is the chosen approach for authentication used by Fex

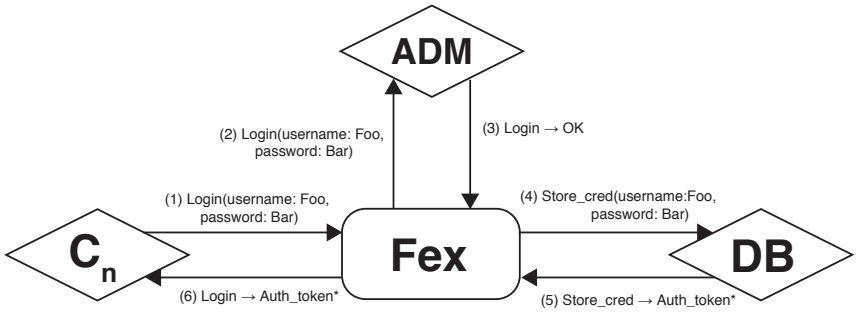


Figure 19: Auhtentication Token Structure

4.2.4 *Yink Protocol*

This section will describe the request and response protocol used by Fex, and the reasoning behind its format. The Yink clients, that Fex communicates with, are mobile phones with a low amount of resources. It is therefore necessary to consider how to minimize the amount of data passed through the communication.

The data formats considered for the protocol are JSON [16], eXtensible Markup Language (XML)[51], and a self-defined markup language. Of these JSON is chosen, due to its human readability, its wide use, its lightweight markup semantics [41], and simplicity. Having a widely used data format improves maintainability, which is rated as 4 in Section 3.2. The light markup semantic of JSON reduces the amount of data required to express and organize request and response content.

To minimize the communication between Fex and the Yink clients the frequency with which it is needed to communicate must be considered as well. Two techniques are considered: event-based- and polling-based communication.

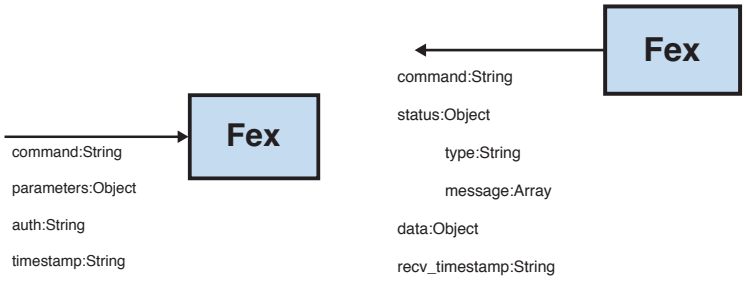
Using an event-based approach requires a connection is kept open at all times between Fex and the Yink clients. Whenever an event occurs on Fex, the results of this event is sent to each Yink client it is relevant to. Such an event can be a Yink client updating the user's status. The advantage of event-based communication is that the Yink clients always have data representing the current state on Fex. The disadvantage of event-based communication that the connection must be kept open at all times. On mobile phones an application can be shutdown at any time, resulting in events not being sent to the Yink client. Furthermore mobile phones can have inconsistent Internet connections resulting in some events not being sent successfully to the Yink clients. To accommodate for a lost connection Fex would need to maintain data representing a real-time state of Fex, that is sent to Yink clients when they initiate a connection with Fex.

With a polling-based approach the Yink clients must request information to receive an updated state of Fex. To achieve this Fex must maintain real-time data to return to the Yink clients when they poll. Polling can be done either periodically or on-demand. The benefits of polling is that no resources are spent on keeping a connection open, and the number of data exchanges between Fex and the Yink clients are reduced. The disadvantage of polling is that the data in the Yink clients is not always identical to the data in Fex. This can have an effect since the user of a Yink client might try to redirect to another user that is unavailable.

Polling was chosen as approach, since it is the simplest and least resource intensive of the approaches.

The structure of the requests sent to and from Fex is the last aspect considered for minimizing the amount of data sent. The request

structure must be as simple and minimalistic as possible, to minimize data transfer.



(a) Request. (b) Response.

Figure 20: The Yink Protocol.

The structure of requests sent to Fex will have four fields: *command*, *params*, *auth*, and *timestamp*. The structure is illustrated with the data type of the fields in Figure 20a. The *command* field is required to specify the type of the request sent to Fex. The *params* field is required to pass along the parameters of the command if it has any. The *auth* field is required to confirm the sender of the request's identity as described in Section 4.2.3. Lastly the *timestamp* is required by the Yink client to distinguish between responses to commands of the same type.

Requests sent from Fex to the Yink clients have four fields: *command*, *status*, *data*, and *recv_timestamp*. The structure is illustrated in Figure 20b. The *command* and *recv_timestamp* are used by the Yink clients to distinguish between the received responses. *status* is used to report if the request sent by the Yink client was successfully handled or not. The *data* field contains the response to the request if there is any.

The types of commands and their parameters can be seen in Appendix D.

4.2.5 *Hawkeye*

The purpose of Hawkeye is to transform the event-based messages from the JTAPI server, an external resource, to data that can be polled by Yink clients through Fex.

The messages received describe events that happen in regards to the Cisco Terminals and their calls. These events contain information about which line or Cisco Terminal it occurred from, along with data about what action occurred, and the action's associated data. An example could be the event of talking, where a line states it is talking with another line on a certain call id.

Since the Yink protocol uses polling, and the messages from the JTAPI server are event-based, it is necessary to store the events. These events modify the state of the Cisco Terminals, connected to Yink. This state is a real-time snapshot, and represented in the database.

The messages that modify this state are: *Talking* and *Idle*. The *Talking* event occurs when a Cisco Terminal's line starts a call. *Idle* occurs when a Cisco Terminal's line leaves a call and returns to an idle state. When a line is in an idle state, it is able to initiate and receive calls.

When Hawkeye receives a *Talking* event, it changes the state of the participating Cisco Terminal in the database. A *Talking* event associated data contains a source, a destination and a Cisco Call Identifier (CallID). The source and destination represents a line. The CallID is used to identify the started call. An entry in the *Line_call_ids* schema is created for each line, source and destination, together with the CallID. The *Line_call_ids* schema is described in Section 4.2.6.

An *Idle* event's associated data consists of a CallID. When an *Idle* event is received, the entry for source line of the event and the CallID is deleted.

4.2.6 *Data Management*

Fex must manage data in the form of e.g. the authentication tokens described in Section 4.2.3. Managing the data requires storing it in a structured manner, and for this purpose a database is used.

The database is required to store authentication tokens, the event-based data from Hawkeye described in Section 4.2.5, and information to support the functionality defined in the user stories, described in Section 3.1 such as internal and external contacts.

Data integrity must be ensured in the database to guarentee Yink clients are not sent corrupt data. Data integrety means data must be of the correct type, and within the correct range. Fex typechecks all data before inserting it in the database. However, as the database is an external resource, data can be inserted without using Fex. Therefore the database must be responsible for data integrity.

The event-based data from Hawkeye consists of two elements, which line is active, and the call id it is active on. A line represents a number, which can either be internal or external. The call id is an active channel in which two lines are communicating with each other.

A call is therefore represented as two entries, one for each active line, where a entry consists of the line and call id. Finding two lines connected in a call, is done by finding two lines with the same call id. Whenever a line hangs up the call, or leaves it, its entry is deleted from the database.

Using this approach enables the polling of event-based data as described in Section 4.2.4, since a current state of all lines is represented

and can be retrieved through a single request.

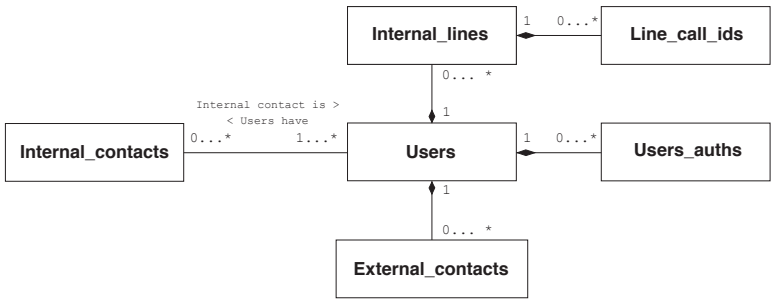


Figure 21: Snippet of the ER model of the Fex database.

A snippet of the ER model of the Fex database is shown in Figure 21.

The table called *User_auths* is used for the authentication tokens. This table links authentication tokens to users.

The table *Line_call_ids* is used to keep track of what lines are currently in a call. These lines are linked to the lines in the table *Internal_lines*. This enables Fex to tell which users and their lines are currently busy.

The tables *Internal_contacts* and *External_contacts* are used to represent contacts, internal and external respectively. Internal contacts are users of the system, while external contacts can be external numbers of users within the system or external numbers to non-users, such as a number to the local pizza place. The tables associated with *External_contacts* are intentionally left out as they are implementation specific.

4.2.7 *Fault Prevention, Error Handling, and Recovery*

This section covers how and why Fex handles and recovers from errors. The terminology used is defined by [18].

In order to enhance the reliability in Fex, the concepts fault tolerance and fault prevention are introduced. Fault tolerance is added by enabling Fex to recover from unexpected errors.

The recovery is done by having a supervisor, denoted Sup* in Section 4.2.1 Figure 13. The supervisor monitors processes and receives information from the monitored processes upon unexpected termination. Unexpected termination occurs, when a fault occurs in the given process. This means if an error occurs in a process, e.g. API, and it thereby terminates unexpectedly, the supervisor is then able to react. The reaction can be defined by the type of error that occurred. If it is an non-critical error, the supervisor will re-instantiate the terminated process with a new state. This state can be based on the type of error

that occurred. If a critical error is received by the supervisor, it does not re-instantiate the process since it is an error the system cannot recover from, e.g. if the communication channel required for Fex to run is blocked in the system environment.

This concept is applied to API, H, Ext_i, JtiCon, SqlCon, and HwkE.

On a non-critical error; API, H, JtiCon, SqlCon, and HwkE are all re-instantiated by Sup^* with a predefined initial state. Ext_i is re-instantiated by H with the task it had prior to its unexpected termination.

The re-instantiation of Ext_i will only occur within a time limit, referred to as timeout, specified by Msg_i. This timeout is applied due to the real-time aspect of Yink, as described in Section 4.1.5. The timeout is returned to Msg_i, which is an error in the perspective of Msg_i. Errors received by Msg_i from Ext_i are caught. When an error is caught, it is returned to C_n as a message containing a description of the error.

Fex is aware of the external resources that it utilizes, therefore a layer is introduced to prevent faults in these resources. The layer consists of type checking the arguments of a command before they are passed onto the external resources. This can done since there exists some preliminary logic about the external resources, e.g. an error occurs when trying to save a string where the external database resource expects an integer. This is done to prevent faults, i.e. fault prevention, and is represented in the Com_n process of Fex.

4.3 YAND

The design of Yand is described to cover the design of the Yink clients. The decisions described in this section applies to Yios as well. Yand is an application that enables the user to interact with Fex, and hereby gain access to the PBX functionality of the underlying CallManager infrastructure. It serves as a graphical layer between the user and the functionality made available by Fex. The functionality of Yand is thus as required by the user stories described in Section 3.1.

In order to understand the functionality required of the Yand application the user stories are analyzed. Upon analyzing these user stories functionality is derived. This derived functionality that Yand should provide is shown in Figure 22.

4.3.1 *Communication with Fex*

Yand is a Yink client making the functionality of Fex available to the users on the *Android* platform. Therefore Yand communicates with Fex in order to make this possible. This communication is via TLS as described in Section 4.2.2. In Yand this communication can be handled synchronously or asynchronously.

Id	Functionality	User Stories
1	A user interface to add contacts through	2 and 3
2	A user interface to search for internal contacts through	2 and 14
3	A user interface to delete a contact through	4
4	A user interface to edit an external contact through	5
5	A user interface to view a user's contacts and their general information	7, 11, and 12
6	A user interface to redirect a call to a contacts	8
7	Functionality to be automatically logged in if the given authentication is still valid	13
8	A user interface to set the user's status through	10
9	The home screen interface should contain the calling number, if such exists	9
10	A user interface that enables the user to redirect to any number	1

Figure 22: Yand's functionality derived from user stories.

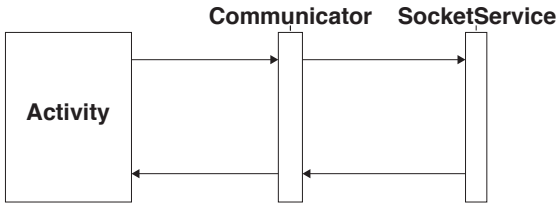


Figure 23: Synchronous behavior.

In synchronous behavior Yand sends a request to Fex, and then waits for a response before performing any other actions. Synchronous behavior is shown in Figure 23. If a response is not received within a predefined time limit, the request would timeout and Yand would be able to perform other actions again. With synchronous behavior no other requests are performed until the request is processed, i.e. a response has been received. Synchronous behavior is useful in applications where error handling is important, since the results of all requests are known. If several large requests, requests that take a long time for Fex to process, are performed in sequence, these will hold any other requests until they finish. This does not take advantage of the asynchronous behavior of Fex, which can process several requests in parallel.

In asynchronous behavior Yand sends requests to Fex and continues performing actions while waiting for a response. Asynchronous

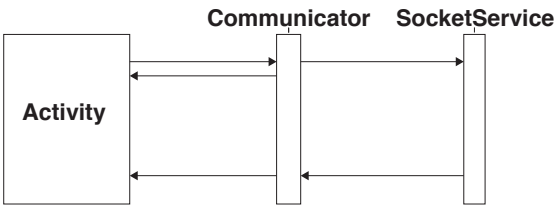


Figure 24: Asynchronous behavior.

behavior is shown in Figure 24. When a response is received from Fex that requires visual output Yand updates the GUI. With asynchronous behavior the response is not considered until it is received.

Asynchronous behavior is useful for application where the GUI needs to be updated often based on either user input or periodic tasks. It is important to note with asynchronous behavior that responses are not always received in the order the requests are sent. Yink is a soft real-time system. As described in Section 4.1.5 the quality of responses from Fex degrades over time. The issue with asynchronous behavior is that the result of each request is not known. In Yand this is important for some requests such as *login* and *redirect*, where as for polling requests such as *fetch_contacts* and *who_am_i* it does not affect the use of Yand.

Yand initially used synchronous behavior, but was later changed to use asynchronous behavior, due to the issues with synchronous behavior, such as not taking advantage of the parallelism of Fex. These issues became apparent when Yios was being developed, and therefore the refactoring occurred as a result of comparative refactoring, discussed in Section 7.1. For Yink comparative refactoring was used during the development of Yand and Yios. The term comparative refactoring covers: If a need for refactoring is discovered in Yand, the same need will apply to Yios and vice versa.

Asynchronous behavior enables periodic polling based requests, such as *fetch_contacts*, while not blocking for user issued requests, such as *redirect*. Because Fex has parallelism, thus supports asynchronous calls well, and asynchronous behavior allows for non-blocking requests, Yand will use asynchronous behavior for its communication with Fex.

When developing applications for mobile phones there are two solutions for communication channels, which must be considered. One is to keep the channel open at all times. This means that the channel is opened once and is used for all subsequent requests. Alternatively a channel can be opened and closed in intervals, when Yand sends requests to Fex.

The concern about mobile phones is that they have limited resources in the form of battery life. Therefore it is important to consider which of the two described solutions for communication channels is the most power efficient. This is however difficult to know for all cases, since some platforms might be optimized for one or the other.

Therefore Yand's communication channel solution will instead focus on how sending and receiving between Yand and Fex would work. Since there is no solution that trumps the other, Yand will be designed to use the simplest solution, which is open and close the connection for every request or group of requests. This solution is simplest, because the uncertainty of when the application will get paused and unpaused is not a concern.

4.3.2 *Controllers*

In Yand a controller refers to an *Android* Activity [24]. An *Android* Activity is a class in an *Android* application associated with one or more views. In the activity class the functionality of the associated views is defined, e.g. what should happen if a button in the active view is pressed.

An *Android* application cannot exist without an activity. Yand has an initial controller called the *Welcome* controller. In the *Welcome* controller it is checked if the user has a valid authentication token. The *Welcome* controller then starts the *Authenticate* controller or *Main* controller depending on the result of the authentication token check.

The interaction between other activities is based on user input. E.g. if the user provides valid credentials and presses login in the *Authenticate* controller, the *Main* controller is launched and the user forwarded to it.

The controllers must handle the asynchronous behavior, as described in Section 4.3.1, when sending requests to Fex. Controllers in Yand have two approaches to handling this, one is using *AsyncTask* of *Android* [25], which is a class for handling computations run asynchronously outside the main User Interface (UI) thread. The other is to create a service [26], which is an application component for handling long running computational tasks in the background of an application.

A service still runs in the main UI thread, but it does not make an application unresponsive by blocking the user interface. Services can start new threads to ensure network communication does not result in any blocking.

The controllers in Yand will be based on services, since the *Android* documentation [25] states that *AsyncTask* is not intended for operations that might run longer than a few seconds. It is not possible to guarantee the communication with Fex will be completed within this time frame, and for this reason *AsyncTask* is discarded.

Since the communication happens asynchronously the results of requests cannot be parsed to the controllers through a function's return value. The results are therefore sent to the controllers using *BroadcastReceivers* [3]. Broadcasts on an *Android* device are messages broadcasted globally on the device. If a controller has a *BroadcastReceiver* it can be set to listen for specific messages, and perform predefined actions if a message is broadcasted.

The asynchronous behavior functions as follows: A function call is made from a controller. This generates a message and sends it to the *Service*. The *Service* sends the request to Fex and receives the response. The response is then processed and a broadcast is sent to inform the controller that it needs to perform an action.

4.3.3 *GUI Design*

The mockups of the design of the views associated with the controllers described in Section 4.3.2 can be seen in Appendix B.

In Section 4.1.5 it was described how Yand and Yios should have identical user interfaces, through a generic design. It must be considered how to make the design intuitive to use on both platforms.

Android applications often utilize the buttons found on *Android* devices. However, since these are not present on an iPhone, this is not an option.

The delimiter for the design is the iOS design guidelines, since they must be followed for an application to approved for the *App Store*. The user interfaces are therefore designed based on the design guidelines in [33].

The implemented user interfaces for Yand and Yios can be seen in Appendix F.

4.4 SUMMARY

Yink consists of two types of components: Fex and the Yink clients: Yand and Yios. The structure of the components is illustrated in Figure 25. Fex is the interface server described in Section 4.1. It provides an API to the Yink clients containing the functionality described by the user stories in Section 3.1. To provide this functionality Fex interacts with external resources: a JTAPI server for controlling the Cisco Terminals and DB for administrating data such as contacts and users.

The Yink clients provide a user interface through which the functionality of Yink is available. They interact with Fex through a wireless internet connection, making the solution mobile.

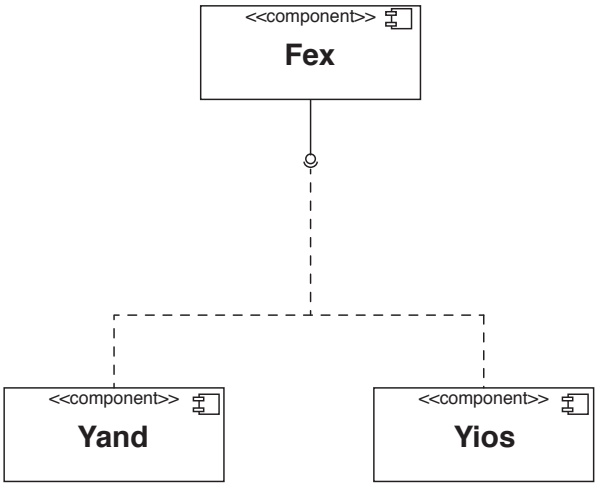


Figure 25: Component diagram of Yink.

5 IMPLEMENTATION

This chapter covers the challenges that happened throughout the implementation. It will describe what the challenges were and how they were dealt with.

5.1 PROGRAMMING LANGUAGE FOR FEX

The programming language for Fex was chosen based on a set of requirements. As seen in Section 3.2; reliability, availability, security, and scalability are the software qualities important for the choice of language. Scalability in this project is defined as Yink's ability to manage an increasing amount of users, as described in Section 1.2. In order to manage an increasing amount of users it is necessary to be capable of handling user requests concurrently. This is because concurrency is equivalent to the amount of users a system is able to serve at the same time, therefore higher concurrency improves the capabilities for serving a larger user base. Therefore the programming language for Fex needs to handle concurrency well in order to be scalable.

The programming languages mentioned in *wsdemo* [43] were considered. The *wsdemo* test involves benchmarking performance of programming languages built for concurrency, using the HyperText Transfer Protocol (HTTP), in regards to the Concurrent ten thousand connections (C10k) problem. The C10k problem is a benchmark milestone in regards to concurrency, and involves being able to establish and maintain 10,000 concurrent connections.

The results of the test showed *Erlang* was the only programming language able to reach this milestone, while maintaining a response time that is deemed suitable for Yink. *Erlang* also provides some unique features in regards to error handling, which improves reliability. Since reliability of Yink is set to 4, as described in Section 3.2, this is deemed a valuable feature. It was decided in Section 4.2.2 that the programming language must support TLS, which *Erlang* provides via a built-in module [21]. The chosen language for Fex is *Erlang*.

5.2 CISCO VOIP LIMITATION

User story 1, presented in Section 3.1, describes how it should be possible to redirect calls from the Yink clients. However, this is not functioning in Fex due to a limitation discovered in the Cisco VoIP

system.

Figure 26 shows the sequence of events, when a call is received on a line with Mobility is enabled. When a call is received by the CallManager, it redirects the call to the device with the intended receiver, i.e. if a call is received with 761 as receiver, then the Cisco Terminal with line 761 receives the call. If Mobility is active for this line the CallManager will wait until a specified timeout, and then transfer the call to a new source, specified by the configuration of Mobility. This source can either be another terminal, or an ISDN source, e.g. a mobile phone. In this case, the new source will be a mobile phone.

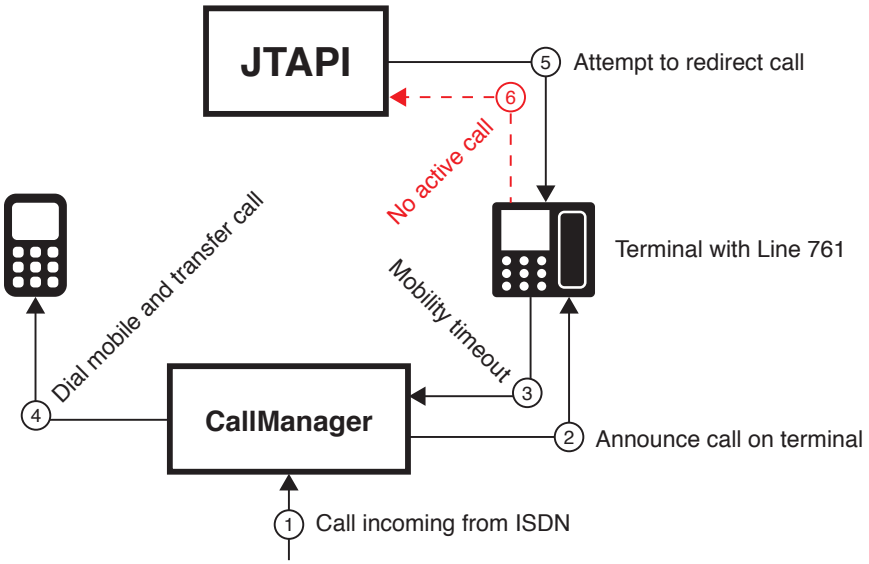


Figure 26: Cisco Mobility Scenario with JTAPI.

If the call is picked up on the mobile phone then the Cisco Terminal cannot control the call. This control involves use of various features, e.g. redirect call. This means that the JTAPI server, which allows for using the control features of the Cisco Terminals, is unable to provide any control features to the call.

When the call is hung up on the mobile phone it returns to the Cisco Terminal for a specified timeout before it is permanently hung up. When it is returned to the Cisco Terminal, it is in the state of hold. If the call is resumed on the Cisco Terminal before the timeout runs out, it is not hung up, but is instead permanently given to the Cisco Terminal. During the period of hold it is possible for the JTAPI server to control it.

The limitation of the CallManager, which traverses to the JTAPI server, is that it cannot hang up the call on the mobile phone, but it must be done manually by the user of the mobile phone.

In order to achieve the redirect functionality desired by user story 1 the use scenario must be altered. Instead of calling the redirect command in the JTAPI server and have it end the call on the mobile phone before redirecting the call, the user must instead issue the command and then hang up on the mobile phone. The command that redirects the call should, when issued, continuously poll the status of the associated Cisco Terminal, until it receives an on hold status or a timeout is reached. When the on hold status is received the JTAPI server can resume the call and redirect it to the desired number.

With this altered use scenario, the user story could have been implemented, but it was discovered too late in the development process to be implemented.

5.3 MODIFICATION OF JTAPI

By examining the source code of the external resource, the JTAPI server, it is deemed that the functionality it provides in regards to reliability is not sufficient. This insufficiency is centered around having no fault tolerance of the connection between the JTAPI server and HwkE. HwkE is the process described in Section 4.2.1. Meaning that if the JTAPI server and HwkE are disconnected, events sent from the JTAPI server will be lost. This lead to a modification of the JTAPI server in order to improve its reliability by increased fault tolerance. The fault tolerance was implemented by creating a request queue on the JTAPI server, which would collect events if no connection had been established. This ensured that requests would only be attempted to be sent when a connection had been established. It furthermore meant that if the connection was disconnected, then instead of losing the events they would be queued until the connection had been reestablished. Once the connection was reestablished the queue would be emptied by sending the events through the newly established connection. The events will be sent to HwkE as described in Section 4.2.5.

The following is an example of when this modification is relevant and can be seen in Figure 27: A user *X* makes a call, the event is sent to Hawkeye, which updates the database. The connection is then lost between Hawkeye and the JTAPI server, and before the connection is reestablished the user *X* has already hung up. If the modification had not been done to the JTAPI server then Hawkeye would not have received that user *X* had already hung up, and would show user *X* as occupied. However, the modification ensures that Hawkeye receives all events, and therefore the event that user *X* hung up is sent to Hawkeye as soon as the connection is reestablished.

An additional modification was done to the commands of the JTAPI server in order to prevent inconsistencies when using the `TRANSFER` command. Instead a command called `CALLTRANSFER` was introduced,

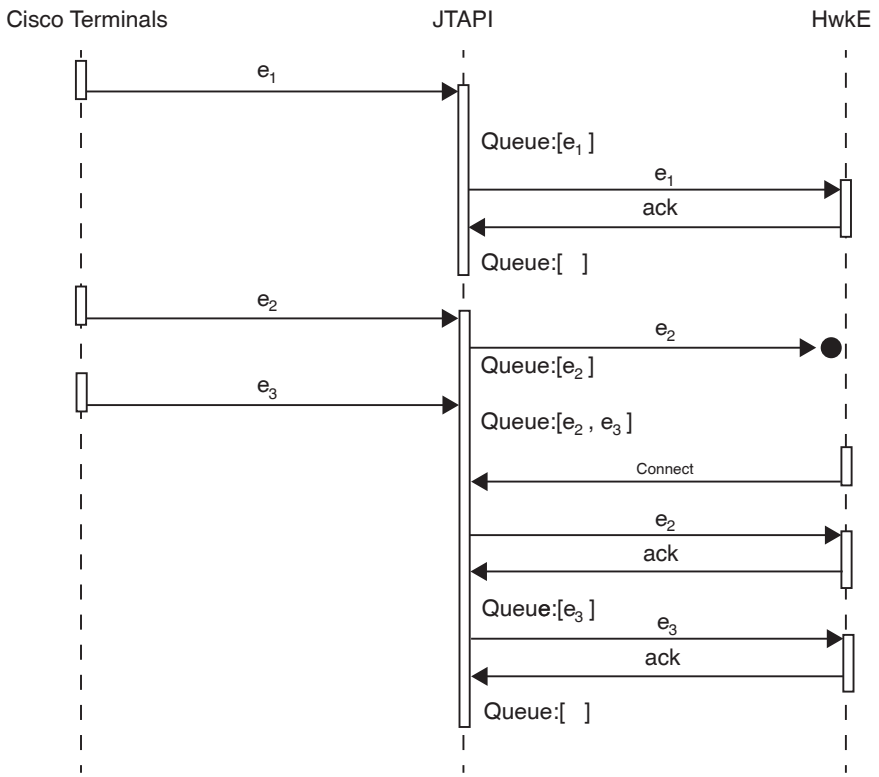


Figure 27: Sequence diagram illustrating the queue in JTAPI.

which can emulate the transfer as a Cisco Terminal makes.

The modifications can be seen in Appendix E

5.4 ADAPTED TEST-DRIVEN DEVELOPMENT

This section covers the use of TDD and automated testing, along with how they affected the implementation.

Automated testing is an integrated part of the used continuous integration server, *Jenkins* as described in Section 2.2.4. This involves running all unit tests of each component, such as Fex and Yand. The testing tools used allowed for calculating the code coverage of the given component. *Jenkins* provides the option of saving the build history along with artifacts created with each build. Since *Jenkins* archives each build, it is possible to create graphs that show code coverage in relation to the chronological order of builds. Code coverage describes to which degree code have been tested, in this case lines of code.

The graphs for Fex and Yand are shown in Figure 28 and Figure 29, respectively.

The initial use of TDD was to create unit tests prior to any code. However, as the design got more complex the requirements and com-

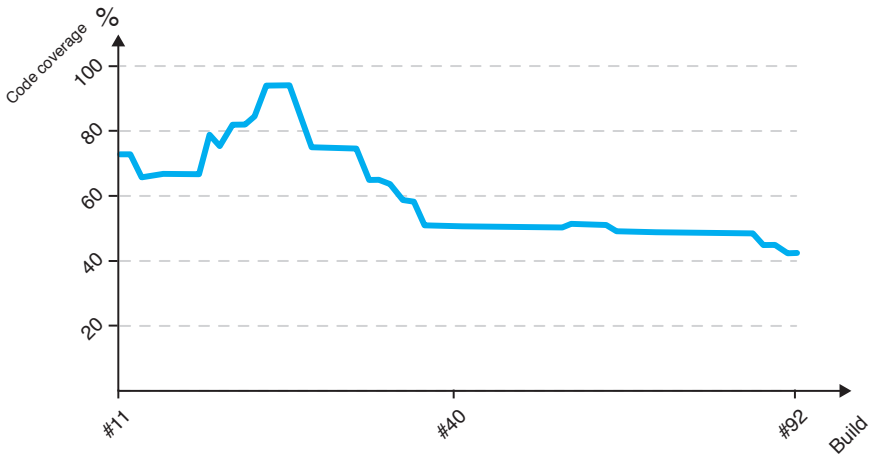


Figure 28: Code Coverage Graph of Fex.

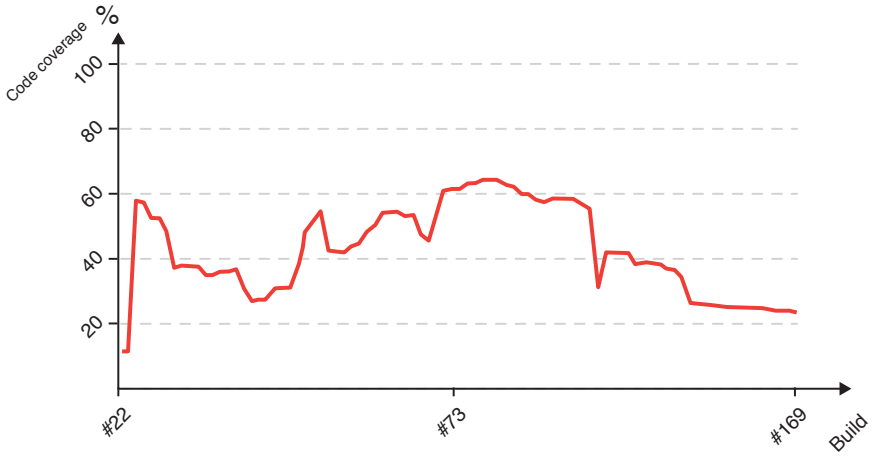


Figure 29: Code Coverage Graph of Yand.

plexity of the implementation rose. In order to test these complex parts of the system through unit testing, the concept of stubs and mocks was introduced. The definition for mocks and stubs from [23] is used. Stubs returns predefined answers to calls made under a test, and usually only answer to what they are programmed for. Mocks are objects with predefined expectations or assumptions which form a specification for calls they are expected to receive. Having this initial goal combined with complex code and mocking meant that the quality of the tests degraded to a point where the tests became meaningless. It was therefore decided to use alternative types of tests. These tests were acceptance tests conducted on paper and through verbal communication within the pair programming teams. This is why the code coverage decreased through the later builds, since more code was added while the tests of the new code were done through verbal communication and informal acceptance tests, which is not represented in the code coverage.

6 TESTING

This chapter covers the concurrency tests made and the reasons for them. Yink should be able to be used on the current user base of more than 1,000 users, described in Section 1.2. The implementation will be tested to verify if Fex is able to fulfil this. Fex is assumed to be the bottleneck of Yink, as it functions as a single point of failure, i.e. if it is unavailable the system is not functioning. The test will create a worst case scenario of load on Fex, with 1,000 users each having one mobile phone. This scenario has all users using the system simultaneously, and thereby creating 1,000 concurrent connections. Concurrency, in regards to connections to Fex, can therefore be seen as a benchmark for testing scalability for Yink.

6.1 CONCURRENCY TEST

Three tests were conducted to test the concurrency of Fex. The tests were conducted using a modified version of the client software *wsdemo* [44]. The modified version of *wsdemo* is found at [45].

The setup used were two computers running *Debian* 6. The computers were called connector and receiver. The connector was the computer that ran *wsdemo* and receiver was running Fex and the JTAPI server. The specifications of the connector were: Quad core CPU with 2.66GHz on each core and 8GB RAM. The specifications of the receiver were: Duo core CPU with 2.66GHz on each core and 4GB RAM. Both computers were connected to a 100Mbit switch via Ethernet. The network settings of both computers were modified according to [44].

Each test created 1,000 connections that ran concurrently and sent requests to Fex. The tests were conducted ten times each.

The design of the processes $\mathtt{Com}_n$, $\mathtt{Msg}_i$, $\mathtt{Ext}_i$, SqlCon, and the group *Request Controller* is as described in Section 4.2.1.

Test *A* was conducted to test if the internal structure of Fex was able to handle 1,000 concurrent connections, each with one request, meaning that no communication with external resources was required. This test consisted of sending the request seen in Listing 2.

```
1 {
2   "command": "login",
3   "params": {
4     "username": "bjarke",
```

```
    "password": "12356"
  },
  "authd": "",
  "timestamp": "*DYNAMIC-TIMESTAMP"
}
```

Listing 2: Test A Request

`*DYNAMIC-TIMESTAMP` is a timestamp generated by the connector, used to verify the response. Fex is supposed to return an error message announcing that there is a missing attribute `auth`, since the *Request Controller* catches that auth is misspelled in the request. The results of test *A* are shown in Figure 30.

Test Id	Responses Received	Connection Timeouts
A1	986	14
A2	985	15
A3	986	14
A4	985	15
A5	989	11
A6	983	17
A7	989	11
A8	991	9
A9	989	11
A10	991	9
Total	9874	126
Average	987.4	12.6
Percentage	98.74%	1.26%

Figure 30: Test *A* – 1,000 Connections with a wrong format.

The results show that not all connections were established. The timeouts occur when Fex cannot accept anymore connections, thus Com_n is never used in these cases. The consistency of the amount of connections established suggests that it is the network interface of Fex that is the limitation, since it has been proved that a server implemented in *Erlang* using TCP/IP can handle 10,000 connections. Therefore the limitation is deemed to be either the hardware of Fex or the TLS connection [43]. This means that the TLS connection could be a bottleneck and will be discussed further in Section 6.2.

Test *A* is not sufficient to test the concurrency of Fex, since Fex depends on external resources to perform tasks for the Yink clients. Therefore we perform test *B*.

Test *B* was conducted to test if Fex can handle 1,000 concurrent connection, each with one request, that requires the use of the external database resource, MySQL. This is relevant since MySQL is a fundamental part of Fex and thus a potential single point of failure for the functionality of Fex. The request of test *B* contained a valid login command with incorrect user credentials, seen in Listing 3.

```
{
  "command": "login",
  "params": {
    "username": "bjarke",
    "password": "12356"
  },
  "auth": "",
  "timestamp": "*DYNAMIC-TIMESTAMP"
}
```

Listing 3: Test B request

The results are shown in Figure 31.

Test Id	Responses Received	Connection Timeouts
B1	979	21
B2	982	18
B3	983	17
B4	985	15
B5	990	10
B6	979	21
B7	991	9
B8	987	13
B9	984	16
B10	991	9
Total	9851	149
Average	985.1	14.9
Percentage	98.51%	1.49%

Figure 31: Test *B* – 1,000 Connections with the right format, but with a timeout on Message.

Since the results of test *B* are similar to those of test *A* it raises questions about results. This similarity occurs because Msg_i is responsible for responding to the client, if it is a valid command. Msg_i is supposed to always respond to the client, therefore it has a timeout of two seconds. This means that, if Msg_i does not receive a response

from Ext_i within two seconds, it will respond to the client that Fex is experiencing heavy load. Therefore in order to test how many connections the database can handle the timeout on Msg_i was removed in test *C*.

Test *C* is the same test as *B*, except for the removed timeout in Msg_i. The results are shown in Figure 32. These results show that SqlCon cannot handle more than 89 connections on average. The bottleneck of SqlCon is discussed further in Section 6.2.

Test Id	Responses Received	Connection Timeouts
C1	89	18
C2	89	11
C3	90	14
C4	89	6
C5	89	6
C6	89	15
C7	89	18
C8	90	7
C9	89	12
C10	89	16
Total	892	123
Average	89.2	12.3
Percentage	8.92%	1.23%

Figure 32: Test *C* – 1,000 Connections with the right format, but with no timeout on Message.

6.2 BOTTLENECKS

The implementation of SqlCon, described in Section 4.2.1 and shown in Figure 16, is done by having a single connection to the external database resource MySQL. This is assumed to be the reason for the bottleneck highlighted by the test results in Table 32. The reason for this assumptions is that requests are assumed to be blocking actions. Blocking actions block the connection from handling other requests until a response is received to the current open request. A proposed solution to reduce this bottleneck, is to create multiple connections to the MySQL resource and thereby possibly improving capabilities of concurrent data exchange.

The implementation of the secure connection used from C_n, described in Section 4.2.1 and shown in Figure 13, to the *Request controller* is done using TLS. Through stack traces, available in Erlang, it was determined that the TLS driver implementation in Erlang was the cause of connection timeouts, shown in Figure 30, Figure 31, and Figure 32. This is assumed to be caused by the data intensity of the handshake done by TLS. A solution to reduce this bottleneck would be to distribute the *Request controller* across multiple machines, and thereby take advantage of more network interfaces and increased processing power.

6.3 SUMMARY

The concurrency test conducted showed that the current implementation does not suit the use domain. Yink is supposed to serve more than 1,000 users, but in Test *C* less than 100 connections were successfully handled. The number of users is expected increase to increase, and therefore this poses a significant issue. However, due to the design of Fex that involves modeling isolated processes, it is possible to distribute processing across multiple machines. This would improve the computational power of the implementation, and could improve Yink in order to meet the expectations, since the bottlenecks originate from computation related issues.

7 REFLECTION

This chapter contains our reflection on the used development method and the design choices made. The use of XP and our adaption of it, described in Section 2.2, meant that *Pair Programming* would be an integral part of our project. The effect *Pair Programming*, TDD, and *Continuous Integration* had is described further in Section 7.3.

Refactoring and *Simple Design* have been used as described. *Simple Design* enabled us to start designing a solution for a complex problem domain with a lot of uncertainty.

The use of *Refactoring* meant that the internal structure of the Yink clients was significantly altered, which is described in further detail in Section 7.1.

Simple Design was not used in cases where the cost of changing the was deemed to high. This is discussed in Section 7.5.

Collective Code Ownership and *System Metaphors'* impact is covered in Section 7.2

Planning Game and *Small Releases* were used as described. The meetings at the end of each iteration determined the expectation of the next release. The releases were kept small by having short iterations. Not using the actual game proscribed by *Planning Game* was not a hindrance to the planning of each iteration.

The *Whole Team* practice was upheld as described. The communication tools used, described in Section 2.2.3, allowed for knowledge sharing with the customer and help during important design decision, such as the choice of language for Fex.

Coding Standard was followed as described. The coding guidelines of the programming languages used were followed as knowledge of these guidelines were obtained.

Sustainable Pace prevented burn outs throughout the development period and maintained the motivation of the developers.

7.1 COMPARATIVE REFACTORING

Comparative refactoring, as mentioned in Section 4.3.1, is a term used to describe the refactoring that occurred as a result of development of the two identical components Yios and Yand. The comparative refactoring occurred when development was initiated on Yios after Yand had been in development for a while. As the components should be based on the same generic design, as described in Section 4.1.5, the developers working on Yios questioned the structure and functionality of the implementation in Yand. As a result the design was refactored

when developing Yios, and these changes were reflected in the implementation of Yand. The comparative refactoring resulted in a general increase in the design quality, achieved at a low cost.

In this case the refactoring affected the communication behavior in Yand, since it was changed from synchronous to asynchronous. The cost of this refactoring was a day and this is deemed a low cost in regards to the improved quality. Discovering the flaw at this stage was important, however, since the cost of change would have increased over time.

For future projects where similar components such as the Yink clients are being developed, the concept of comparative refactoring can be a useful technique to improve design. It provides insight into previous implementations, and prevents "system goes sour", and in addition provide an implicit design and implementation review. The "system goes sour" concept is defined as:

> " *System goes sour — the system is successfully put into production, but after a couple of years the cost of making changes or the defect rate rises so much that the system must be replaced* "
>
> — Kent Beck [7, p. 3]

7.2 COLLECTIVE OWNERSHIP

The *Collective Code Ownership* practice, described in Section 2.2, was challenging to uphold, due to the complexity of the implementation. The Yink clients and Fex did not share the same programming paradigm.

The programming paradigm used for Fex was functional. The developers were inexperienced in the functional paradigm, but it was chosen despite this, due to implementation benefits, e.g. connection performance [43]. Learning a new paradigm requires time, and it was deemed an issue in regards to the deadline of the project. This problem was solved by assigning one pair to work exclusively on code related to the functional paradigm. This causes issues in regards to *Collective Code Ownership*. This issue was addressed by having a workshop, where the developers which worked in the functional paradigm introduced the other developers to the paradigm. The workshop functioned as knowledge sharing, with a practical approach of coding exercises.

Creating team-specific *System Metaphors* were used as a tool to concretize communication, e.g. Yink, Yios, Yand, Fex and Hawkeye. Since these *System Metaphors* are team-specific and only understood by the team members, they create a common and more concrete understanding of the system. Having this advantage, compared to non-team

members, tied the team members together and strengthened the core of the team.

Working in an osmotic environment meant that the pairs were always close together when developing. This lead to every pair programming team being either directly or indirectly part of the conversations about the design or implementation of other teams' tasks. Because every team was present during these informal conversations it meant that the teams would all have an opportunity to provide feedback. This gave every team a sense of ownership of other teams' code, since they indirectly reviewed the decisions made during the conversations. This indirect review gave all teams a sense of responsibility for the decisions made.

7.3 CONTINUOUS REVIEW

This section describes the experience with using XP practices to prevent "system goes sour" and ending up with a bad design. The "system goes sour" concept is described in Section 7.1. The used definition of bad design, is taken from Extreme Programming Explained [7, p. 48] by Kent Beck.

> " *One conceptual change requires changes to many parts of the system. Logic has to be duplicated. [...] You can't add new function without breaking existing function.* "
>
> — Kent Beck [7, p. 48]

Bad design can originate from the *Simple Design* practice, where a design choice can lead to a need for design changes later, which can be expensive. The methods used to avoid this primarily involved review and reflections on decisions.

The TDD practice of XP was the primary tool used to prevent the system from going sour. In the initial part of the development TDD was used as prescribed, that is unit tests were written before writing code. TDD provided two things in the initial part of the project. Firstly, as unit tests were written it was ensured defects were not introduced into the system. Eliminating defects early in the development reduces the risk of expensive refactoring. Secondly, writing unit tests requires reflecting on the code being written. This reflection lead to an implicit review of the design decisions made with regards to the code.

The TDD practice was adapted to fit the development through the project, as described in Section 5.4. This was due to two realizations with regards to TDD. Firstly, some code does not benefit from unit tests. An example is network communication which has to be over-stubbed [30] in order to be unit tested.

> *" Over-stubbing makes your tests weak, which means they pass even if they're unsynchronized with the code, verifying behavior that isn't real anymore. "*
>
> — Lucas HÃºngaro [30]

Stubbing all the behavior in a unit test removes the purpose of the test since the only thing tested is the stubbing it self. Secondly changing the design of code, means the unit test must be changed as well. As an example, when the signature of a function is changed, the unit tests must be changed to fit the new signature. This process of changing both the code and the unit test is expensive. To accommodate this an informal approach was used for unit testing and TDD, when unit tests were deemed expensive or to provide no benefit. Before writing code a unit test or acceptance test was written in collaboration between members of a pair on a piece of paper. Using this method gave the reflections on design decisions TDD gives without the overhead of writing and maintaining costly or not beneficial unit tests.

The *Osmotic Environment*, described in Section 2.3, was another principle used to ensure review of design decisions. A risk associated with developing two components that communicate with each other, such as Fex and the Yink clients, is that the developers of one component have misunderstood an aspect of another component. Such a misunderstanding can affect design decisions, since they will be based on wrong assumptions. The *Osmotic Environment* prevented this by having all discussions regarding design decisions in the same room. This meant that the pairs could correct each other, if they overheard a misconception in the other pair's discussion. Having *Osmotic Communication* therefore helped ensure misunderstandings did not propagate into the design.

The *Pair Programming* practice of XP was followed throughout the entire development and provided two forms of review. Since review of design decisions and implementation was conducted by more than one developer, this led to beneficial discussions and knowledge sharing.

Continuous Integration provided a form of implementation review and insurance that defects were not introduced in the components. The continuous integration server, *Jenkins*, as described in Section 2.2.4, was used and setup to build and test the individual components whenever they were updated. This kind of testing is referred to as automatic testing and is described in Section 5.4. This gave the advantage of discovering defects when they were introduced.

All of these practices were used to ensure that decisions made due to the *Simple Design* practice were of high quality. This was done

to make sure that the system did not go sour or had a bad design. This was ensured by using the XP practices and *Osmotic Communication*. This assured continuous review and knowledge sharing. In most cases reviewing through *Pair Programming* and TDD was sufficient to ensure high quality design decisions. This was evident as few expensive changes had to be made to any components of Yink. *Osmotic Communication* was used in many cases, one of them was the designing of the GUI for the Yink clients. The interaction design discussion was conducted at a black board, and all developers became involved to ensure the right choice was made.

7.4 EXTENDING AN UNKNOWN ENVIRONMENT

This section covers our reflection upon extending an existing software environment, and dealing with the uncertainty surrounding it.

Extending a complex environment, such as a Cisco VoIP system requires a lot of knowledge in order to understand the capabilities and limitations it provides. Obtaining this knowledge is a tedious task, and can be overwhelming. The use of *Whole Team*, and having the customer available through communication tools, allowed for continuous expanding knowledge of the external resources, which a Cisco VoIP system consists of, as the design evolved incrementally.

This incremental process of gaining knowledge of the external resources, and the customers' insecurity on the capabilities of the external resources, lead to the discovery of a critical limitation. This limitation is described in further detail in Section 5.2.

This limitation could have been foreseen, if the functionality needed from the CallManager was tested earlier. This functionality is mobility for the Cisco Terminals, which enables calls to be received on mobile phones, see [12]. When the call appeared on the mobile phone it should then be possible to redirect the call. Instead the calls were redirected from the Cisco Terminal, without having had the call on the mobile phone. However, with the use of *Simple Design* and the incremental design process, this occurred late in the development process.

A method for dealing with the uncertainty, which originates from the user stories, is to have spikes. Spikes allow developers to experiment within uncertain domain in order to learn enough about a new technology to accurately estimate user stories. A spike is used when a user stories has a lot of uncertainty associated with it, or when a design decision is deemed expensive to alter. The concept of spikes has not been used throughout this project, since it involves estimating user stories, which is a part of the *Planning Game* that has not been used. This could maybe have helped with issues concerning the limitations of the external resources with regards to mobility.

Dealing with uncertainty in a complex environment can be tough. One of the reasons is that the quality of the documentation describing the environment may vary and some aspects of the environment may not be covered. Getting complete knowledge of the environment being extended is ideal, but often impractical. Often it requires becoming a specialist in the domain in order to understand the concepts. This requires a large amount of time.

We deem XP a good choice, despite being a factor in noticing this critical limitation late in the development process. The sustainable development pace, that XP provided was deemed to outweigh the late domain knowledge gain.

7.5 ARCHITECTURAL SPIKES

Using *Simple Design* is starting off with a simple design and continuously expanding and refactoring it. As described in the article Architecture Spikes [46], this usually involves "Small Design Up Front".

> *" I can't get away from the impression that this approach corresponds with a "Small Design Up Front", validated with experimental code. The problem I have is that it seems to ignore modeling as a means to validate the architecture. In my view, it is a mistake to assume you always need working code for a proof of concept "*
>
> — Erik Philippus [46]

This approach of validating design by experimental code can be expensive in regards to design decisions that affect the architecture. Some designs are tough to implement from scratch, even though they have been used in a similar system. This was the case with the initial design of Yink, described in Section 4.1, and created an *Architectural Spike*. Implementing a simple version of a peer-to-peer network protocol is complex.Knowing this, and based on the article by Erik Philippus, we used the practice *Architectural Spike*.

The practice of *Architectural Spike* uses models to represent design, and allowed considering multiple solutions instead of choosing the most simple design. An *Architectural Spike* was used to choose the communication structure of the components, as described in Section 4.1. This was due to the estimated cost of changing communication structure of the components. We deemed this a good choices, as it clarified the fundamental structure of the system. Having the fundamental structure in place, allowed the developers to reach the same expectations of each component in the system, as they were developed in parallel. The use of the *Architectural Spike* practice, did not mean the simplest design was not chosen, but it allowed for a shared thought pro-

cess of the developers regarding some fundamental design choices, that affected multiple components in the system.

8 CONCLUSION

This report covers the process we went through developing a solution to the following problems:

How can we develop a scalable, portable, and extendable mobile solution for call administration of Cisco VoIP telephones, while maintaining high quality in regards to the customer's needs?

How can we use a development method to handle the uncertainty of developing a solution that extends an existing product?

How can we combine the domain knowledge of the customer and the technical expertise of the developers in order to create a suitable use context for the customer?

How can we select and use a programming language in order to fulfill the software qualities of the project?

This project involved developing a solution for call administration of Cisco VoIP telephones. The developed solution is named Yink. Developing this solution required a definition of the problem domain. The problem domain is: Customers of a Cisco IP PBX solution who wants to be able to use the call administration features of their Cisco Terminal on their mobile phone.

Developing an extension to an external resource, such as the IP PBX from Cisco, causes uncertainty in the project. To manage this uncertainty an adaption of the agile development method XP was used. This method involves a set of practices.

Through the *Whole Team* practice of XP a technician with expertise in Cisco VoIP became part of the development team to provide technical assistance and domain knowledge.

The practices *Simple Design* and *Architectural Spikes* were used to manage the uncertainty in regards to design decisions. *Architectural Spikes* were used when changing a design decision was considered expensive, and it was required to gain a more extensive understanding of the problem domain. To ensure that the choices made with *Simple Design* were of a high quality, they were exposed to review by using *Osmotic Communication*, *Pair Programming*, and TDD. *Simple Design* decisions were improved by refactoring and through the development process comparative refactoring was discovered. Comparative refac-

toring covers how refactoring can take place across components when these are similar in functionality.

TDD was adapted from having unit tests prior to code being written to a more informal approach to testing. This informal test approach involved verbal communication and writing acceptance tests on paper. The reason for this adaptation was that a realization occurred that the unit tests are not needed in order to get the desired effect of TDD. It is possible to achieve the primary purpose of TDD, which is to obtain clarity about the code before writing it, with informal tests describing desired input and output.

The use context being expanded upon was that any call administration had to be done via Cisco Terminals. This use context is expanded to include call administration from mobile phones. This allowed users to become mobile with regards to call administration. The use context was further expanded by the developers' technical expertise, which uncovered opportunities for new features.

Making a mobile solution for call administration of Cisco VoIP telephones requires enabling communication between mobile phones and Cisco Terminals. This communication was enabled by making an interface server, Fex, through which the mobile phones could interact with the Cisco Terminals and administrate calls. To enable the mobile phones to interact with Fex, a set of mobile applications named Yink clients were developed: One for *iOS*, called Yios, and one for *Android*, called Yand.

Yink was made scalable with regards to users by developing Fex with focus on handling a large amount of concurrent connections. For Fex to achieve the desired software qualities it was necessary to consider a range of programming languages that focused on concurrency and reliability. The choice of *Erlang* meant that Fex could handle this amount of concurrent connections, and was able to recover from faults. Even though Fex fulfilled the software qualities by being developed in *Erlang*, there were bottlenecks discovered with regards to the technologies used. Fex uses TLS for communication with the Yink clients, and external resources, such as a database, which limits the concurrency. The concurrency issues could be solved by distributing Fex over different machines and improving the external resources to support a higher level of concurrency.

Portability was introduced through the Yink clients. Having the Yink clients use a native application design makes them portable across all mobile platforms, since a unique Yink client is developed for each platform.

Yink is not able to administrate calls, as desired by the customer, due to a limitation discovered in Cisco VoIP. Implementing the functionality requires the limitation of Cisco VoIP to be removed. If the limitation is overcome, then Yink is able to fulfill the customer's needs.

Part I

APPENDIX

A

OBJECT MODEL

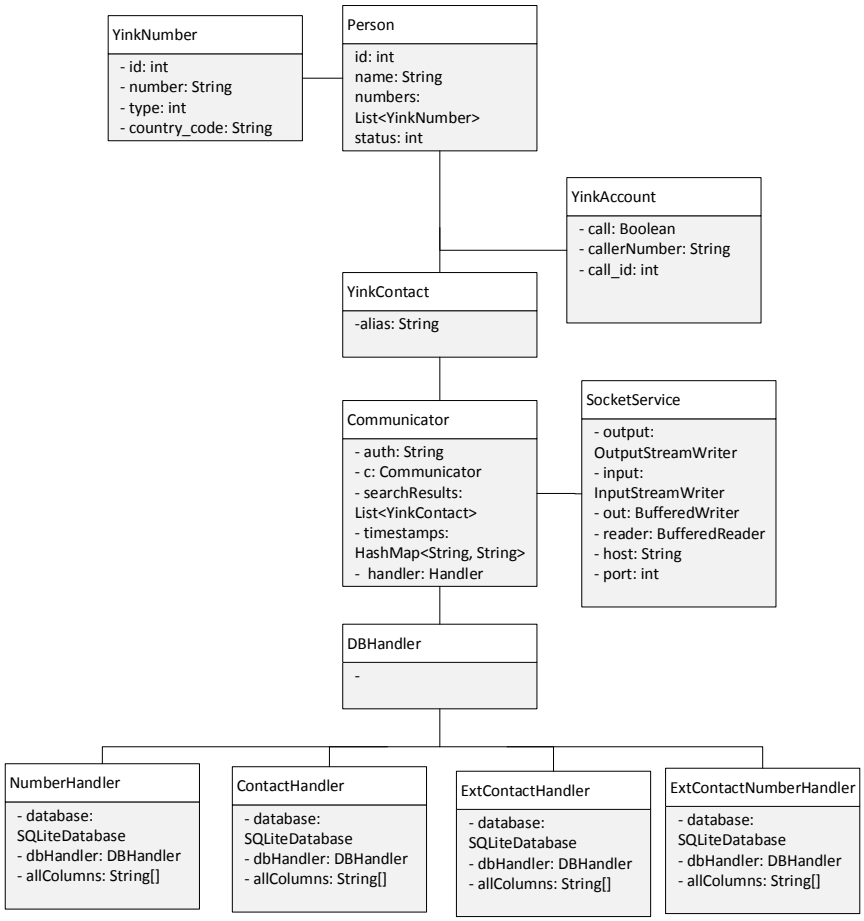


Figure 33: Object Model of Yand

B

MOCKUP OF GUI IN YINK CLIENTS

Figure 34: Mockups of the views of Yand and Yios.

C

FEX ARCHITECTURE

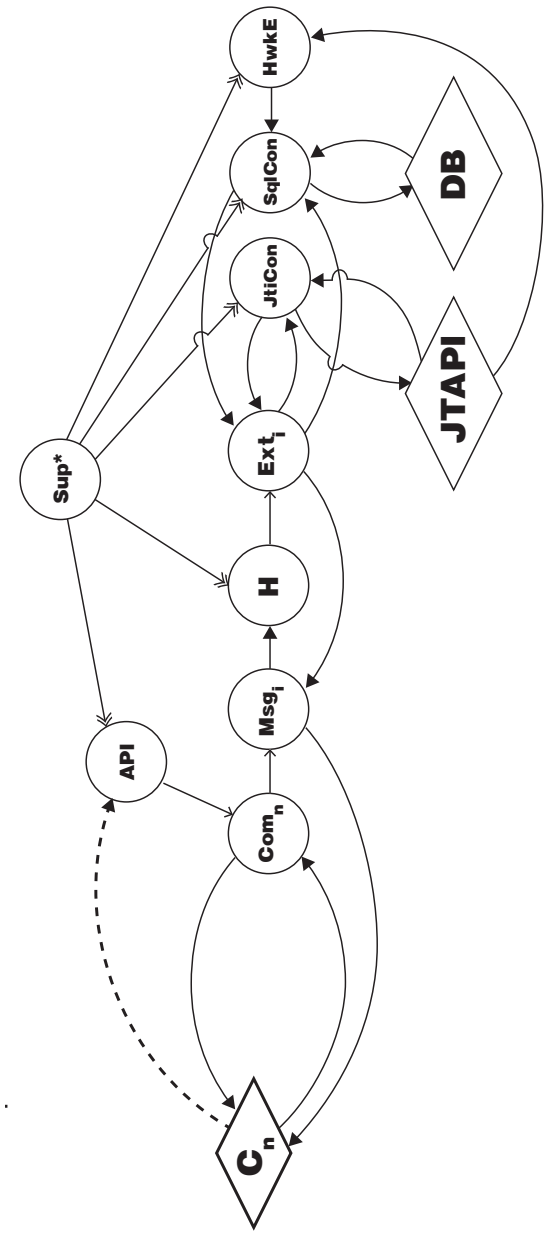


Figure 35: The final architecture of Fex.

D

FEX COMMANDS

LOGIN

```
Input: ["username"(string),"password"(string)]
Output: ["username"(string),"auth"(string)]
```

Listing 4: login

- Input
  - username: String containing the user's username.
  - password: String containing the user's password.
- Output
  - username: The username that the user is logged in with.
  - auth: The auth token, which is a session key for that user and device.

REDIRECT

```
Input: ["int_id"(string)] or ["ext_id"(string)]
Output: ["username"(string),"authToken"(string)]
```

Listing 5: redirect

- Input
  - int_id: Internal line id to redirect the call to.
  - ext_id: External line id to redirect the call to.
- Output
  - num: Number that has been redirected to.

WHO_AM_I

```
Input: Takes no parameters
Output: ["user"(User)]
```

Listing 6: who_am_i

- Output
  - user: The user of the given auth.

UPDATE_STATUS

```
Input: ["status"(int)]
Output: ["status"(int)]
```

Listing 7: update_status

- Input
  - status: The status wanted by the user.
- Output
  - status: The status granted to the user.

FETCH_CONTACTS

```
Input: Takes no parameters
Output: ["contacts"(contact)]
```

Listing 8: fetch_contacts

- Output
  - contacts: A list of contacts for that user.

CREATE_CONTACT

```
Input: ["num"(string), "country_code"(string), "alias"(
    string)] or ["user_id"(integer), "alias"(string)] or ["
    user_id"(integer)]
Output: ["contact"(contact)]
```

Listing 9: create_contact

- Input

  - num: The number of the contact you wish to create.
  - alias: A string that is used by the user to identify the contact.

- Output
  - contact: The created contact is returned.

SEARCH_CONTACTS

```
Input: ["search"(string), "limit"(integer)]
Output: ["contact"(contact)]
```

Listing 10: search_contacts

- Input
  - search: A string used to search the databased, searches through numbers and names.
  - limit: The maximum search results you want returned.
- Output
  - contacts: List of contacts is returned.

UPDATE_CONTACTS

```
Input: ["ext_cid"(int), "num"(string), "country_code"(string
    ), "alias"(string)] or ["user_id"(integer), "alias"(
    string)]
Output: ["contact"(contact)]
```

Listing 11: update_contacts

- Input
  - num: The number of the contact you wish to create.
  - alias: A string that is used by the user to identify the contact.
- Output
  - contact: The created contact is returned.

DELETE_CONTACTS

```
Input: ["user_id"(int)] or ["ext_cid"(int)]
Output: ["status"(int)]
```

Listing 12: delete_contacts

- Input
  - user_id: The id of the internal user.
  - ext_cid: The id of the external contact.
- Output
  - user_id: The id of the deleted internal user.
  - ext_cid: The id of the deleted external contact.

CHANGE_DEFAULT_NUM

```
Input: ["int_id"(int)] or ["ext_id"(int)]
Output: ["int_id"(int)] or ["ext_id"(int)]
```

Listing 13: change_default_num

- Input
  - int_id: ID of internal number.
  - ext_id: ID of external number.
- Output
  - int_id: ID of internal number set as default num.
  - ext_id: ID of external number set as default num.

E

JTAPI

```
1  public void StartServer() {
2  int intPort = 7777;
3  ServerSocket sckServer = null;
4  @@ -89,6 +90,17 @@
5  this.provider = provider;
6  }
7  + private CiscoCall findCall(CiscoTerminal cTerm, String
       callId){
8  + TerminalConnection[] conns = cTerm.getTerminalConnections
       ();
9  + for(TerminalConnection c : conns){
10 + CiscoCall call = (CiscoCall) c.getConnection().getCall();
11 + if(call.getCallID().toString().equals(callId)){
12 + return call;
13 + }
14 + }
15 + return null;
16 + }
17 public void run() {
18 String line;
19 BufferedReader in = null;
20 @@ -256,6 +268,21 @@
21 ctrlCall.transfer(traSplit[1]);
22 }
23 }
24 + }
25 + else if (strCommand.equals("CALLTRANSFER")) {
26 + String[] traSplit = strCommandArguments.split(" ");
27 + CiscoCall src = findCall(ciscTerm, traSplit[0]);
28 + CiscoCall tar = findCall(ciscTerm, traSplit[1]);
29 + if(src != null && tar != null){
30 + ctrlCall = (CallControlCall) src;
31 + ctrlCall.transfer(tar);
32 + }
33 }
34 else if (strCommand.equals("FORWARDALL")) {
35 String[] fwdSplit = strCommandArguments.split(" ");
```

Listing 14: “Added CallTransfer” in src/jtapimonitor/CallServer.java

```
@@ -196,15 +196,15 @@
}
}
else if(strCommand.equals("HANGUP")) {
- for (int ansCount=0;ansCount<ciscTerm.
    getTerminalConnections().length;ansCount++) {
- CiscoCall cCall = (CiscoCall)ciscTerm.
    getTerminalConnections()[ansCount].getConnection().
    getCall();
- if (cCall.getCallID().toString().equals(
    strCommandArguments)) {
- ctrlCall = (CallControlCall)cCall;
- log.logEntry("HANGUP: ciscTerm: "+ciscTerm.getName()+"
    ciscAddress: "+ciscAddresses[addrCounter].getName(),4);
- }
- else {
- log.logEntry("HANGUP: ciscTerm: "+ciscTerm.getName()+"
    callid No longer on the terminal",4);
- }
+ CiscoCall srcCall = findCall(ciscTerm, strCommandArguments
    );
+ if (srcCall != null) {
+ ctrlCall = (CallControlCall) srcCall;
+ srcCall.drop();
+ log.logEntry("HANGUP: ciscTerm: " + ciscTerm.getName() + "
     ciscAddress: " + ciscAddresses[addrCounter].getName(),
    4);
+ } else {
+ log.logEntry("HANGUP: ciscTerm: "+ciscTerm.getName()+"
    callid No longer on the terminal",4);
}
}
else if (strCommand.equals("HOLD")) {
@@ -223,7 +223,7 @@
}
}
else if (strCommand.equals("UNHOLD")) {
// Expects the callid to passed in the strCommandArguments
    variable
for (int ansCount=0;ansCount<ciscTerm.getTerminalConnections
    ().length;ansCount++) {
CiscoCall cCall = (CiscoCall)ciscTerm.getTerminalConnections
    ()[ansCount].getConnection().getCall();
@@ -259,28 +259,33 @@
else if (strCommand.equals("TRANSFER")) {
String[] traSplit = strCommandArguments.split(" ");
```

```
- for (int ansCount=0;ansCount<ciscTerm.
    getTerminalConnections().length;ansCount++) {
- CiscoCall cCall = (CiscoCall)ciscTerm.
    getTerminalConnections()[ansCount].getConnection().
    getCall();
- if (cCall.getCallID().toString().equals(traSplit[0])) {
- ctrlCall = (CallControlCall)cCall;
+ CiscoCall srcCall = findCall(ciscTerm, traSplit[0]);
- ctrlCall.transfer(traSplit[1]);
- }
+ if (srcCall != null) {
+ ctrlCall = (CallControlCall) srcCall;
+ ctrlCall.transfer(traSplit[1]);
}
}
else if (strCommand.equals("CALLTRANSFER")) {
+ Call tar = provider.createCall();
+ ctrlCall = (CallControlCall)tar;
String[] traSplit = strCommandArguments.split(" ");
+ String callId = traSplit[0];
+ String caller = traSplit[1];
+ String calledNumber = traSplit[2];

- CiscoCall src = findCall(ciscTerm, traSplit[0]);
- CiscoCall tar = findCall(ciscTerm, traSplit[1]);
+ CiscoCall src = findCall(ciscTerm, callId);

- if(src != null && tar != null){
- ctrlCall = (CallControlCall) src;
- ctrlCall.transfer(tar);

+ for (addrCounter = 0;addrCounter < ciscAddresses.length;
    addrCounter++) {
+ if (ciscAddresses[addrCounter].getName().equals(caller)) {
+ ctrlCall.connect(ciscTerm, ciscAddresses[addrCounter],
    calledNumber);
+ Thread.sleep(1000);
+ CallControlCall srcControlCall = (CallControlCall) src;
+ srcControlCall.transfer(tar);
+ log.logEntry("CALL: ciscTerm: "+ciscTerm.getName()+"
    ciscAddress: "+ciscAddresses[addrCounter].getName()+"
    Number:"+calledNumber,4);
+ }
}
}
```

Listing 15: "New redirect that takes callid, from, and to" in src/jtapimonitor/CallServer.java

```
@@ -14,4 +14,5 @@
public class TcpClient {

private static TcpClient instance = null;
private boolean evc = false;
- private Socket echoSocket = null;
- private BufferedWriter out = null;
- private BufferedReader in = null;
- private boolean conSuccess = true;
+ private Socket echoSocket;
+ private PrintWriter out;
private String serverHost = "";
private String serverPort = "";
private String evcEnabled = "";
- private static DebugLog log = null;
+ private String lastCommand = "";
+ private static DebugLog log = new DebugLog();
+ private ArrayList<String> queue = new ArrayList<String>();


protected TcpClient() {
- this.log = new DebugLog();
ConfigXml readConfig = new ConfigXml();
this.serverHost = readConfig.getConfigField("eventclient", "
    evc_server");
this.serverPort = readConfig.getConfigField("eventclient", "
    evc_port");

private void createConnection() {
+ System.out.println("Entering createConnection");
try {
- /*
- echoSocket = new Socket(serverHost, Integer.parseInt(
    serverPort));
- echoSocket.setTcpNoDelay(true);
- echoSocket.setSoTimeout(200);
- */
InetAddress addr = InetAddress.getByName(this.serverHost);
SocketAddress sockaddr = new InetSocketAddress(addr, Integer
    .parseInt(this.serverPort));
this.echoSocket = new Socket();
this.echoSocket.connect(sockaddr, 200);
- this.log.logEntry("Established a connection with: "+this.
    serverHost+".",4);
- if (this.echoSocket.isConnected()) {
```

```
41 - OutputStreamWriter output = new OutputStreamWriter(this.
       echoSocket.getOutputStream());
42 - this.out = new BufferedWriter(output);
43 - } else {
44 - this.log.logEntry("Connection wasn't established");
45 - }
46 + this.log.logEntry("Established a connection with: "+this.
       serverHost+".", 4);
47 + System.out.println("Established a connection");
48 + this.out = new PrintWriter(this.echoSocket.getOutputStream
       (), true);
49
50 + this.emptyQueue();
51 } catch (UnknownHostException e) {
52 - this.log.logEntry("Don't know about host: "+this.
       serverHost+".",4);
53 + this.log.logEntry("Don't know about host: "+this.
       serverHost+".", 4);
54 } catch (IOException e) {
55 - this.log.logEntry("Couldn't get I/O for the connection to:
        "+this.serverHost+".",4);
56 + this.log.logEntry("Couldn't get I/O for the connection to:
        "+this.serverHost+".", 4);
57 + System.out.println(this.echoSocket);
58 }
59 -
60 }
61
62 public void sendEvent(String command) {
63 - if (this.echoSocket.isConnected()) {
64 - try {
65 - command = command.trim();
66 - this.log.logEntry("Sending message: "+command+".",4);
67 - this.out.write(command+"\r\n");
68 - this.out.flush();
69 - }
70 - catch (Exception e) {
71 - System.err.println("BufferReader failed. Reason: "+ e);
72 + if (this.evc) {
73 + this.log.logEntry("Adding message to queue: " + command +
       ".", 4);
74 + this.queue.add(command);
75 + this.log.logEntry("EVC Enabled", 4);
76 + if (!this.echoSocket.isConnected() || this.echoSocket.
       isClosed() || this.echoSocket.isOutputShutdown()) {
77 + this.createConnection();
```

```
78 + } else {
79 + this.emptyQueue();
80 }
81 - } else {
82 - this.createConnection();
83 }
84 + }
85
86 - // try {
87 - // out.close();
88 - // in.close();
89 - // //stdIn.close();
90 - // echoSocket.close();
91 - // }
92 - // catch (IOException e) {
93 - // System.err.println("Closing sockets failed. Reason: "+
      e);
94 - // }
95 + private void emptyQueue() {
96 + if (!this.queue.isEmpty()) {
97 + System.out.println("Queue is not empty");
98 + ArrayList<String> tempQueue = new ArrayList<String>(this.
      queue);
99 + ArrayList<String> removedItems = new ArrayList<String>();
100 + this.queue.clear();
101 + try {
102 + for(String item : tempQueue) {
103 + System.out.println("Sending message: "+item);
104 + out.println(item);
105 + if (!out.checkError()) {
106 + removedItems.add(item);
107 + lastCommand = item;
108 + } else {
109 + System.out.println("It died. Resurrect!");
110 + if (!removedItems.isEmpty()) {
111 + removedItems.remove(removedItems.size()-1);
112 + }
113 + tempQueue.removeAll(removedItems);
114 + if (!lastCommand.equals("")) {
115 + this.queue.add(lastCommand);
116 + lastCommand = "";
117 + }
118 + this.queue.addAll(tempQueue);
119 + // If connection fails, then try to reestablish a
      connection to listener
120 + this.createConnection();
```

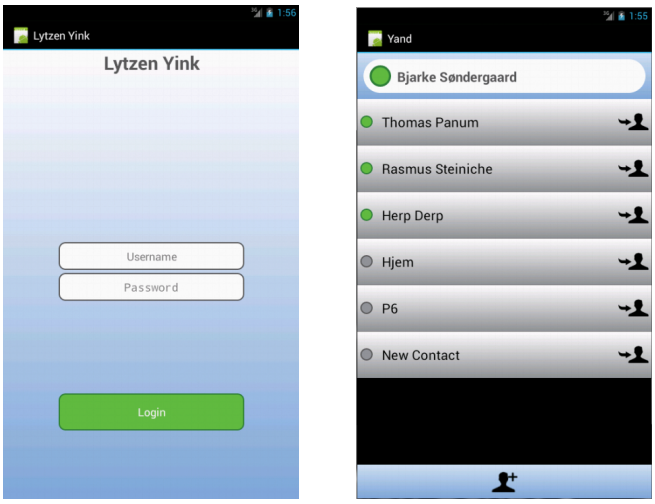
```
+ }
+ }
+ } catch (Exception e) {
+ System.out.println("Outputting failed. Reason: "+ e);
+ }
+ }
}
}
```

Listing 16: “Fixed so that it queues messages until a new listener is available” in src/jtapimonitor/TcpClient.java

F

YINK CLIENTS GUI

F.1 YAND



(a) Yand login screen. (b) Yand main screen.

Figure 36: The screens of Yand.

F.2 YIOS

(a) Yios login screen. (b) Yios main screen. (c) Yios main screen in call.

Figure 37: The screens of Yios.

BIBLIOGRAPHY

[1] feature request: Sending dtmf tones over existing phone call to the far end. 2008. URL http://softwaremonkey.org/Article/Computer/Json-Vs-Xml.

[2] Ivan Aaen. Xp & testing. 2007. URL http://people.cs.aau.dk/~ivan/SOE2007/MM5.pdf.

[3] Android. Broadcastreceiver. 2013. URL http://developer.android.com/reference/android/content/BroadcastReceiver.html.

[4] Android. Launch checklist. Last viewed: 27-05-2013. URL http://developer.android.com/distribute/googleplay/publish/preparing.html.

[5] Lytzen IT A/S. Lytzen it hosted. 2013. URL https://github.com/tpanum/sw804f13/blob/master/presentations/external/lytzen_slides/Hosted%20ppt.pptm?raw=truel.

[6] Jeff Atwood. Code smells. URL http://www.codinghorror.com/blog/2006/05/code-smells.html.

[7] Kent Beck. *Extreme Programming Explained: Embrace Change*. 1999. ISBN 978-0201616415.

[8] Kent Beck. *Test-Driven Development*. Addison–Wesley, 2003.

[9] Kent Beck and Martin Fowler. *Planning Extreme Programming*. 2000. ISBN 978-0201710915.

[10] Barry Boehm and Richard Turner. Observations on balancing discipline and agility. 2003. URL http://people.cs.aau.dk/~jeremy/SOE2011/resources/Boehm.pdf.

[11] Scott Chacon. –local-branching-on-the-cheap. 2013. URL http://git-scm.com/.

[12] Cisco. More flexible mobile communications. 2011. URL http://www.cisco.com/en/US/products/ps6567/index.html.

[13] Cisco. Cisco unified mobility. 2013. URL http://www.cisco.com/en/US/products/ps6567/index.html.

[14] Mike Cohn. *User Stories Applied: For Agile Software Development*. 2004. ISBN 978-0321205681.

[15] Mike Cohn. Differences between scrum and extreme programming. 2007. URL http://www.mountaingoatsoftware.com/blog/differences-between-scrum-and-extreme-programming.

[16] Douglas Crockford. The application/json media type for javascript object notation (json). 2006. URL http://tools.ietf.org/html/rfc4627.

[17] R.Tyler Croy, Andrew Bayer, and Kohsuke Kawaguchi. An extendable open source continuous integration server. 2013. URL http://jenkins-ci.org/.

[18] Alexandre David. Tsw – reliability and fault tolerance. 2011. URL http://arkive2011-sict.moodle.aau.dk/file.php/175/Course_Material_subject_02/lect02-chap02.pdf. This requires an AAU login.

[19] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. 2008. URL http://tools.ietf.org/html/rfc5246.

[20] Erhversstyrelsen. Telestatistik 1. halvÃ¥r 2012. 2012. URL http://www.erhvervsstyrelsen.dk/file/294939/telestatistik-2012-1.pdf.

[21] Erlang. Transport layer security (tls) and its predecessor, secure socket layer (ssl). URL http://www.erlang.org/doc/apps/ssl/ssl_protocol.html. Last viewed: 30-05-2013.

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1999. ISBN 978-0201485677. URL http://refactoring.com/.

[23] Martin Fowler. Mocks aren't stubs. January 2007. URL http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs.

[24] Android Developers Guide. Activity. 2013. URL http://developer.android.com/reference/android/app/Activity.html.

[25] Android Developers Guide. Asynctask. 2013. URL http://developer.android.com/reference/android/os/AsyncTask.html.

[26] Android Developers Guide. Service. 2013. URL http://developer.android.com/reference/android/app/Service.html.

[27] Matt Hamblen. iphone, android account for 82May 2012.

[28] Filip Hanik. The kiss principle. URL http://people.apache.org/~fhanik/kiss.html. Last viewed: 22-05-2013.

[29] Martin Hiller. Software fault-tolerance techniques from a real-time systems point of view. November 1998.

[30] Lucas Hungaro. Over-mocking, over-stubbing and avoiding over-stubbing with mocha. May 2009. URL http://lucashungaro.github.io/tdd/2009/05/13/avoiding-over-stubbing-with-mocha.html.

[31] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. 1999. ISBN 978-0201616224.

[32] IBM. How ssl and tls provide identification, authentication, confidentiality, and integrity. 2012. URL http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/index.jsp?topic=%2Fcom.ibm.mq.doc%2Fsy10670_.htm.

[33] Apple Inc. ios human interface guidelines. 2013. URL http://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/MobileHIG.pdf.

[34] Cunningham & Cunningham Inc. Xp on site customer pitfalls. 2005. URL http://c2.com/cgi/wiki?XpOnSiteCustomerPitfalls.

[35] Cory Janssen. Peer-to-peer architecture (p2p architecture). URL http://www.techopedia.com/definition/454/peer-to-peer-architecture-p2p-architecture. Last viewed: 31-05-2013.

[36] Ronald Jeffries. You're not gonna need it! 1998. URL http://www.xprogramming.com/Practices/PracNotNeed.html.

[37] Ronald Jeffries. What is extreme programming? 2001. URL http://www.xprogramming.com/xpmag/whatisxp.

[38] Neil Mawston. Worldwide smartphone population tops 1 billion in q3 2012. 2012. URL http://blogs.strategyanalytics.com/WDS/post/2012/10/17/Worldwide-Smartphone-Population-Tops-1-Billion-in-Q3-2012.aspx.

[39] Microsoft. Client/server architecture. URL http://www.http://technet.microsoft.com/en-us/library/cc917543.aspx. Last viewed: 31-05-2013.

[40] Microsoft. Microsoft application architecture guide 2nd edition. 2009. URL http://msdn.microsoft.com/en-us/library/ff650706.aspx.

[41] Software Monkey. Json vs. xml for data interchange. 2011. URL http://softwaremonkey.org/Article/Computer/Json-Vs-Xml.

[42] James D. Mooney. Bringing portability to the software process. 1997. URL http://www.cs.wvu.edu/~jdm/research/portability/reports/TR_97-1.pdf.

[43] Eric Moritz. C10k websocket test. 2012. URL https://github.com/ericmoritz/wsdemo/blob/results-v1/results.md.

[44] Eric Moritz. wsdemo. June 2012. URL https://github.com/ericmoritz/wsdemo.

[45] Thomas Kobber Panum. wsdemo. May 2013. URL https://github.com/tpanum/wsdemo.

[46] Erik Philippus. Architecture spikes. July 2009. URL http://www.agile-architecting.com/Articles/Architecture%20Spikes.pdf.

[47] Think Seven. 70 percent of uk businesses will be using voip by 2013. 2012. URL http://www.think7.co.uk/voip/70-percent-of-uk-businesses-will-be-using-voip-by-2013/.

[48] Skype. Skype. 2013. URL http://www.skype.com/en/.

[49] Speech Technology. Voip penetration forecast to reach 79 percent of u.s. businesses by 2013. 2010. URL http://www.speechtechmag.com/Articles/?ArticleID=60918.

[50] Trello. Organize anything, together. 2013. URL https://trello.com/.

[51] W3C. Extensible markup language (xml) 1.0 (fifth edition). 2008. URL http://www.w3.org/TR/REC-xml/.

[52] Don Wells. Set a sustainable, measurable, predictable pace. 1997. URL http://www.extremeprogramming.org/rules/overtime.html.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of June 3, 2013 (`classicthesis` version 0.1).

DECLARATION

Aalborg, June 2013

Bjarke Hesthaven Søndergaard

Esben Pilgaard Møller

Rasmus Steiniche

Thomas Kobber Panum