Norwegian University of
Science and Technology

Real-Time End-User Service Composition Using Google Wave

Espen Herseth Halvorsen

Master of Science in Communication Technology
Submission date: June 2010
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Sergiy Gladysh, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Google Wave is an open and extensible platform for real-time end-user cooperation and communication, which was first presented at the Google I/O Conference on May 27, 2009, and has already attracted the interest of both software-developers, enterprises and end-users. Google Wave is not just a product of Google, but is an open standard, with a well-documented set of protocols and algorithms, open-source software, open APIs for 3rd party developers, playground for debugging and a free service.

The task in this thesis is to examine how Google Wave can be used as a platform/tool for end-user service composition. The in-depth analysis of the Google Wave platform should be done by looking into the following questions: a) overview of the components of Google Wave technology; b) discussion of Google Wave from an end-user service composition viewpoint; c) developing a concrete proof of concept of a service composed in Google Wave based on an end-user service composition scenario; d) investigation of possibilities to make service composition more intelligent and automated (semi-automated) with the help of Google Wave Robots.

Assignment given: 15. January 2010
Supervisor: Peter Herrmann, ITEM

Abstract

This thesis explores *Wave*, a brand new communication and collaboration platform, from the perspective of end user service composition. A description of the different frameworks that are available and a study of how these can be used to create components that can be easily integrated with the platform are provided. Several examples of how the platform can be used to simplify different use cases involving multiple users collaborating on a common goal are provided. A complete solution to collaboratively organize meetings is also developed using these tools, and a detailed explanation of how one creates the necessary Wave Gadgets using web technologies like HTML, CSS and JavaScript, and the necessary back-end Wave Robots using Java are provided.

Preface

This thesis represents the finalization of my master's degree in Communication Technology at the Department of Telematics (ITEM), Norwegian University of Science and Technology (NTNU). The responsible supervisors have been Sergiy Gladysh and Prof. Peter Herrmann.

I would like to thank my fellow students Mats, Kim, Hovard and Henrik, whom I have shared an office with during the work. The semester would without doubt have been less interesting without your presence. As part of this I would also like to thank the institute for providing me with the office and equipment needed for this work.

Most of all I would like to thank my supervisor Sergiy Gladysh. You have been giving me valuable feedback and guidance through my work, provided ideas and hints that have driven my work forward, and have generally been a great help in doing this work. Thank you!

Espen Herseth Halvorsen
June 3rd, 2010
Trondheim

Contents

Abstract **i**

Preface **iii**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**
1.1 Contribution 2
1.2 Outline 3

2 Background **5**
2.1 Computer supported cooperative work 5
2.1.1 The CSCW Matrix 8
2.2 Google Wave 8
2.2.1 Google Wave as a CSCW system 9
2.3 Wave - the Open Source version 10
2.3.1 Terminology 10
2.3.2 Different open sourced parts 11
2.4 Wave Federation Protocol 14
2.4.1 Handling multiple servers 14
2.5 API's for extending Wave functionality 15
2.5.1 The Gadget API 16

2.5.2 The Robot API 19

3 User service composition in Wave 25
3.1 User service composition 25
3.2 Using service composition in communication 26
3.3 Existing solutions: The Extension gallery 27
3.3.1 Different ways to access the content in the Extension gallery 28
3.4 Example composition 1: Collaborative movie night voting . 30
3.5 Example composition 2: Wave as an aid for developers . . . 32

4 Organizing meetings collaboratively 37
4.1 What do we want to solve 37
4.1.1 The planning phase 38
4.1.2 The meeting phase 38
4.1.3 The review phase 39
4.2 Existing solutions 39
4.2.1 Clientside software solutions 39
4.2.2 Web-based solutions 40
4.3 Advantages of using a collaborative approach 41
4.4 Gadget for time scheduling 43
4.4.1 Underlying data structure 44
4.4.2 Calculating the number of possible attendees 47
4.4.3 The final gadget 48
4.5 Gadget for a collaborative agenda 50
4.5.1 Underlying data structure 51
4.5.2 The final gadget 52
4.6 Making it all come together 53

5 Enhancing communication using Wave Robots 57
5.1 Robots as equal participants 58
5.2 Enhancing the previous example using robots 59
5.2.1 Setting up the wave 59
5.2.2 Automatically decide a suitable time 62
5.2.3 Automatically decide a possible schedule based on user input 71
5.3 The finished solution 82

6 Conclusion and Future work **85**
6.1 Future work 86
6.1.1 Trust and security in Wave 86
6.1.2 Combining Wave and Arctis 87

List of Figures

1.1 Illustration of how one can use user service composition in Wave to create any new service to suit a specific task at hand. 2

2.1 The CSCW Matrix, as proposed by Johansen and detailed by Baecker in [1]. Figure reproduced and released under public domain by Magnus Manske, 2009. 7
2.2 The different parts of a wave, diagram from [12] 12

3.1 Screenshot of Wave user composition Use Case: Wave used for selecting films for a movienight. 31
3.2 Screenshot of a wave used by developers to collaborate on a new feature in a software system. 34

4.1 The final gadget for doing time scheduling 49
4.2 The final gadget for collaborative agenda creation. 54
4.3 The extension installer as shown in Google Wave 56

5.1 Top: the new menu item created by the extension installer. Bottom: the resulting wave containing the two gadgets added by the robot. 63
5.2 The Gadget for time scheduling, with the scores for the different time slots calculated by the robot shown beneath. 72

5.3 The agenda gadget, with voting and time specification. Beneath the gadget is the note taking template generated by the robot. 81
5.4 A wave showing the finished gadgets and results of the robots calculations, upper part 83
5.5 A wave showing the finished gadgets and results of the robots calculations, bottom part 84

List of Tables

2.1 The key differences between the two types of Wave Extensions; Robots and Gadgets [13]. 16

2.2 Events that can be catched using the Java Client Library method overriding. Information from [16]. 21

2.3 The different context scopes an event can return data for. Information from [16]. 22

5.1 The default values for the adding or substraction of points for the timeslot based on user priorities. 67

Chapter 1

Introduction

End-user service composition is an idea based around the principle that users should be able to easily create systems that solves their everyday problems in a suitable way. Various solutions have incorporated this idea into products which have had various success rates. In 2009, Google released a first version of a new product called Wave, which aims to be a platform for communication and collaboration. Due to the extensible nature of this platform, it can be used as a tool for highly collaborative end user service composition.

This thesis explores Wave from the perspective of end user service composition. We look at various use cases where Wave can be used to simplify communication by using externally developed extensions. By adding extensions such as gadgets which end users can interact with directly, and robots which takes the same role as a regular user and can be used to automate repeating tasks, we can greatly enhance tasks that involve communication and collaboration. The concept of reusing the same extensions in several different waves is illustrated in Fig. 1.1.

We also create a complete system based on Wave that aims to solve the problem of collaboratively plan and hold meetings. This solution includes gadgets that let users schedule a time and collaborate on agenda items, and also a robot that automates tasks that can better be solved algorithmically than manually by humans.

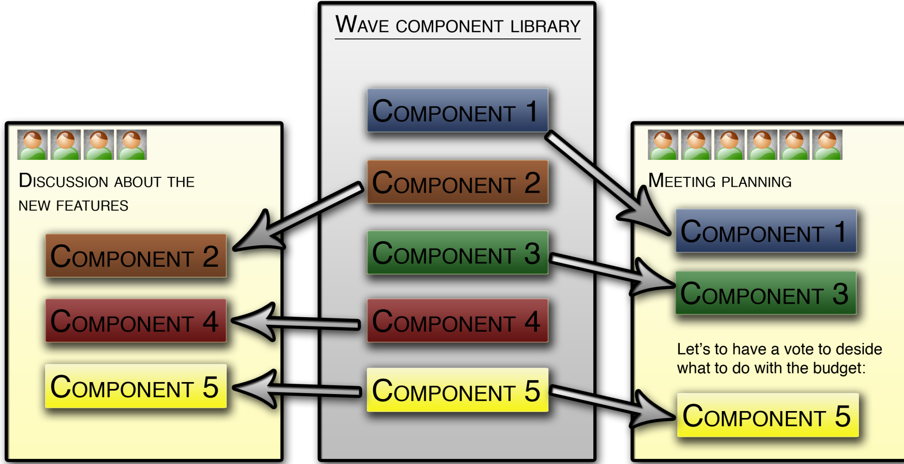

Figure 1.1: *Illustration of how one can use user service composition in Wave to create any new service to suit a specific task at hand.*

1.1 Contribution

In this thesis, we have given an in-depth overview of the Wave platform from an end-user service composition viewpoint. By giving several examples of how Wave can be used by end users to compose services to suit their needs, we have shown that the platform can be a viable option for this usage scenario. A concrete example of such a scenario have been proposed, and the necessary components have been developed from scratch to illustrate the steps needed to create new components for the platform.

We have also detailed the idea behind using robots as an aid in communication, and elaborated on some of the challenges this gives us regarding both technical complexity and end user anxiety from the feeling of losing control. An example robot has been created, showing of some of the potential this technology has to ease communication.

1.2 Outline

This thesis consists of six chapters:

Chapter 2
In this chapter we provide background information about the different technologies used in this thesis. We look at the research field of computer supported cooperative work, and also describe the communication and collaboration platform Wave, which will be used throughout the thesis. We especially look at the possibilities of extending this platform.

Chapter 3
This chapter explores the idea of end user service composition. We look at how this can be applied to communication platforms, and provide several examples of how Wave can be used as an end user service composition system.

Chapter 4
This chapter contains a description that details the process of developing a system that helps users collaboratively organize meetings using the Wave platform. We explore the different phases of the meeting process, and survey how existing solutions tries to solve the problem. Based on these insights, we create components for the system using the Wave Gadget framework.

Chapter 5
In this chapter we explains the process of further developing the solution we built in Chap. 4. We now add intelligence to the system by integrating a Robot developed using the Wave Robot framework. This lets us automate tasks which for the users can be repeating and burdensome to perform manually.

Chapter 6
This chapter summarizes the work done through the thesis. We also discuss a potential future project that could be a continuation of the work done in this thesis.

Chapter 2

Background

This chapter explores some key background technologies. We start by looking at the research field of Computer supported cooperative work in Sect. 2.1. Next, in Sect. 2.2 we look at Google Wave, a new product developed by Google that adheres to the key principles of the CSCW research field. The technology behind this is open sourced, and we look at the general solution named Wave in Sect. 2.3. We explore the technical details of the Wave communication platform in Sect. 2.4, and look at different possibilities to extend the platform in Sect. 2.5. We take a special look at the Gadget API (Sect. 2.5.1) and the Robot API (Sect. 2.5.2) - two key technologies we will employ later in the thesis.

2.1 Computer supported cooperative work

Computer supported cooperative work (CSCW) is an academic field that aims to research how the coordination of collaborative activities can be supported by computer systems. The term CSCW was first used by Irene Grief and Paul M. Cashman at a workshop held in 1984 [19].

One of the cornerstones of this research is that it aims to bring sci-

entists from a wide variety of disciplines together to better be able to consider all the different aspects of the problem. Researchers from the field of computer science are a natural fit, but individuals from fields such as psychology and sociology are also researching this topic.

Wilson has defined the term CSCW as the following:

> *CSCW is a generic term, which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques [34].*

One of the results of this research is a set of needs such systems should fulfill in order to help aiding the collaborative work. These needs can be summarized by these three key principles [7, 34]:

1. **Awareness:** When a group of people are working together, they need to be aware of the work done by others in the group. Each individual needs information about the progress and results of other participants in order to support his own work.

2. **Articulation:** If a large amount of work should be done by a group of people, one needs some way to split the workload up into smaller pieces to let parts of the group or individuals work on it by themselves. Later, one needs a way to integrate the results of this work back into the shared result.

3. **Tailorability:** A system designed to help individuals work together electronically would have to be flexible enough to let the users tailor its functionality to their needs, and in some situations use it in a manner not imagined by the creator of the system at the time of creation.

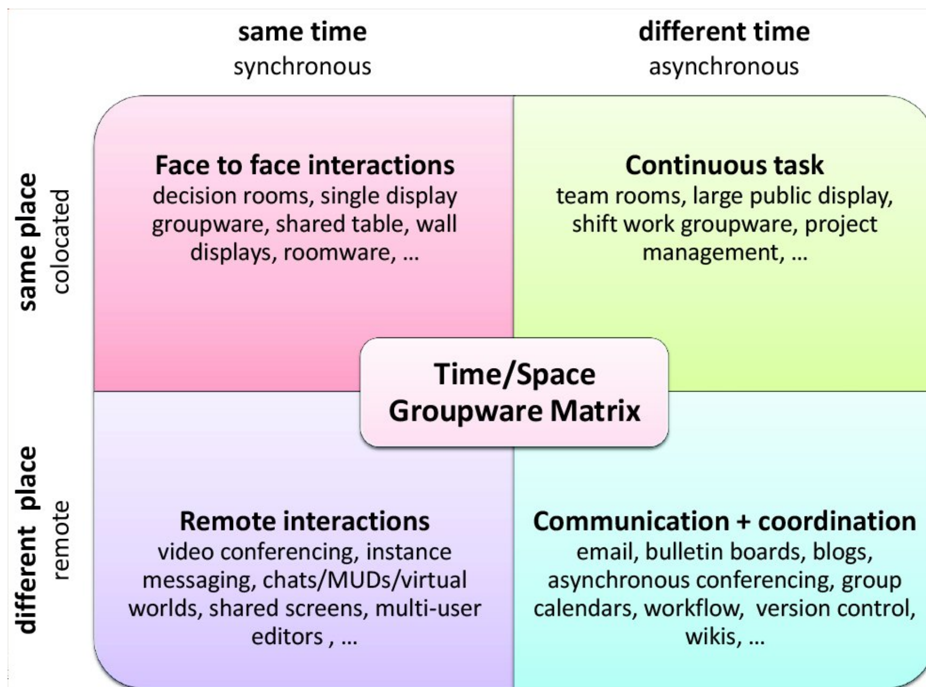

Figure 2.1: *The CSCW Matrix, as proposed by Johansen and detailed by Baecker in [1]. Figure reproduced and released under public domain by Magnus Manske, 2009.*

Research has shown that even though systems fulfills these principles, and are built using knowledge learned from the field of CSCW, there are no guarantee that they will be successful. There are too many unknown factors that are hard to generalize. The success of a CSCW system is based on a variety of factors, where the social context of the users using the system and the specific problem domain it is employed to solve plays a large role. For this reason it is difficult to predict whether a specific CSCW systems will succeed or fail before actually trying it out [18].

2.1.1 The CSCW Matrix

A result of the work done in the field of CSCW is a classification to categorize different types of collaborative tools. The separation of different tools into a CSCW Matrix was first proposed in 1988 by Johansen, and a specific figure appeared in [1]. The idea is to separate collaborative tools by looking at whether they enable collaboration to happen between people residing at the same place (colocated), or people working at different places (remotely). By also dividing the systems in those that support real-time synchronous communication and those where communication happens asynchronous, we get a 2x2 matrix where systems can be placed according to the details of their workings.

The CSCW Matrix, with a selection of different collaborative systems placed in the different regions, can be seen in Fig. 2.1. It is important to note that many systems can be placed in the intersection between the different parts of the matrix if they provide functionality that enables different types of communication regarding location and time.

2.2 Google Wave

Google Wave is a specific implementation of what Google sees as the future of communication and collaboration over the Internet. It is thought of as a successor to e-mail, and is marketed as ”e-mail reinvented in the twenty-first century” by its developers. And by looking at the different features, it can also be seen as a merging of the concepts behind different communication paradigms such as e-mail, instant messaging and wiki-systems [30].

One of the ideas of Google Wave is that it should be a platform in which developers are given the power to extend the system in different ways to facilitate communication and collaboration. Thanks to this flexibility, the end result will be that communication that happens through

this new system is happening in a richer and more user-friendly way than communication happening over older mediums it is meant to replace [29].

Another big difference between e-mail and Wave is that on Wave, the entire conversation is stored on a server. Hence the client just has to retrieve the small deltas that happen when people contribute to the conversation, instead of transferring the whole history every time, as many people do when sending e-mails and quoting the entire thread. Each conversation is stored on the server of the user who started it, but is also synced to the servers of other users participating in the wave.

Both the specific implementation of a Wave front end provided by Google, and the details of the infrastructure that lies behind it, is fairly new. It was first revealed at a conference hosted by Google in May 2009, and was unavailable to regular users until September the same year, when a technical preview was released on an invite-only basis. In May 2010, Google Wave was finally released as a Google Labs product, and registration is now open to all [20].

2.2.1 Google Wave as a CSCW system

Google Wave is an idea that fits perfectly with the description of a CSCW system. But if one tries to position Wave in one of the four corners of the CSCW Matrix, one gets a problem with which one to pick. In a way, Wave is a perfect example of a CSCW system that positions itself in the intersection point between the four different classifications. Wave is suitable to use as a same time/same place system, where it can be used as a way to share information between participants at the same location real-time. The real-time component also facilitates the same time/different place component. It fits into the different time/same place category as well, since it is ideal for task listings and such functionality. And since Wave also allows asynchronous communication, it fits in to the different time/different place category as well.

2.3 Wave - the Open Source version

Despite the fact that Google Wave is a product introduced and developed by Google, it is important to note the openness of the development. As mentioned, Google sees Wave as a possible successor to e-mail, and to make that happen they have decided that the technologies behind Wave should be open sourced [5].

2.3.1 Terminology

To be able to distinguish between the specific implementation provided by Google, and the open source version which can be deployed by anyone, we will use different terms when describing them:

- **Google Wave:** This term will be used to describe the commercial product developed and hosted by Google. As an analogy, this is what Google's GMail is to the e-mail platform.

- **Wave:** This term is used to describe the open source part of the platform. This includes the specifications of the communication protocol, the extension APIs and development methodologies, and the specification for server and client requirements. To use the same analogy, this is what SMTP, POP, IMAP and different other specifications are to e-mail.

Google has coined a few new terms when they created Wave, and it is important to know them to be able to distinguish between the different parts of a conversation happening in Wave. This is outlined in Fig. 2.2.

- **Wave (capitalized):** Refers to the whole communication system. Google Wave is here the specific implementation of Wave made by Google.

- **wave:** Refers to a single threaded conversation, hosted by a Wave server. This conversation can have several participants, and the wave is federated to the Wave servers of all participants.

- **wavelet:** Inside each conversation, the participants can create several distinct threads. Each of these is called a wavelet, and lives inside of the single wave. It is important to notice that inside a wave with a lot of participants, one can create a wavelet with just a subset of these as participants. This wavelet is then just synced between the servers of these distinct participants.

- **blip:** Each distinct message posted by the participants is called a blip. This lives inside a wavelet, and they can be nested inside each other.

2.3.2 Different open sourced parts

The approach taken by Google is to wait with open sourcing the different parts until they become polished, to avoid confusion. As of now, the following parts are either open sourced already or in the process of being open sourced. The license used for this code is the Apache 2.0 license [10].

The Operational Transformation code, with test suite

The Operational Transformation code is the part responsible for the way collaboration works in Wave. As part of the code for concurrency control, it is the "heart and soul of Wave's collaboration" [3]. Wave is based on an idea that most of the editing work will be done and represented locally to provide a high level of interactivity. To be able to synchronize all the different changes between the servers and then to the end users, deltas of the changes need to be distributed in a fast and atomic way.

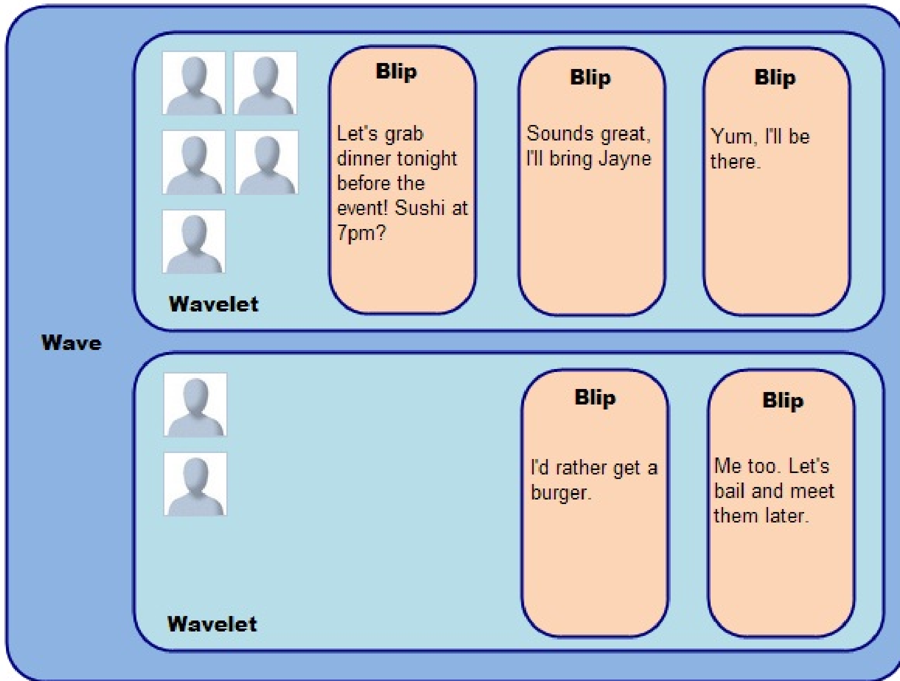

Figure 2.2: *The different parts of a wave, diagram from [12]*

Google has released this code as open source to enable other parties creating Wave servers to be able to reuse it, since it is important that all servers federate the changes in the exact same way. Any little difference can propagate since there will be a large number of change deltas distributed over the lifetime of a single wave. They will also release a comprehensive test suite to aid developers creating their own implementation, or porting the code to other languages [3, 5].

Underlying Wave Model

The Underlying Wave Model is the code that provides the data structures necessary to hold all data related to Wave. This is also a fairly complex

matter, since Wave provides a complete history of the activity related to every single wave [3].

There is a difference between holding the waves in memory when one needs to perform operations on them, such as applying a delta provided by the operational transformation code, and storing them for later retrieval. The model aims to solve both these problems, and by releasing it as open source, Google takes away much of the burden of constructing suitable data structures from developers aiming to create Wave clients and servers [4].

A client/server prototype

To provide a reference implementation of the two previously mentioned parts, Google has also provided a prototype of a server and a client. These are fairly simple at this time, for once the server only holds the waves in memory, it can't store them in a reliable fashion. The client is a command line interface version of Wave, aimed at showing that Wave clients can be coded for any platform - it is not just a web based tool [3].

Even though these prototypes are fairly simple, they show of the protocols and data structures, and hence act as a working demo of the other open sourced code.

The Wave rich text editor

Wave provides a way to annotate the text in a wave to make changes to its appearance. This can be done using an interface similar to the ones one find in rich text editors such as Microsoft Word. Since Google Wave is a web based product, some sort of JavaScript code is needed to allow for such functionality in the browser. In May 2010, Google open sourced this code to let people use it as a reference implementation when developing their own Wave clients [20].

2.4 Wave Federation Protocol

Wave is meant to be a highly cooperative environment, with a strong focus on real-time updates of the content. To be able to achieve this, a new protocol was needed. The proposed protocol is called the Wave Federation protocol [2], and is an extension of XMPP.

Extensible Messaging and Presence Protocol (XMPP) is an open protocol based on XML, created for facilitating real-time message passing and presence control. It was formerly named Jabber, and has mostly been used to power instant messaging programs [11].

The Wave Federation protocol is open and makes it possible for anyone to be a Wave provider. Just as it is the case with mail of today, where companies can provide a mail server as a service to its users or users can host one themselves, anyone with the interest to do so can also set up a server which acts as a Wave server by speaking the Wave Federation protocol.

Even though the full specification for the protocol is still a work in progress, it is published as a proposed standard through a series of whitepapers [26, 4, 33, 22, 32, 25].

2.4.1 Handling multiple servers

To be able to handle the unlimited amount of servers which can possibly participate in any single wave, the protocol is designed with a highly decentralized architecture in mind. Each server has to have a unique identifier, in most cases this will be a URL. To uniquely identify a single wave, this URL is used as an identifier in conjunction with an identifier unique only on that server. This makes it easy to have unique identifiers without each server having to negotiate this with each other [26, 2].

Each wave is owned by the server of the user who started it, and the wave is then called a local wave on this server. Through the use of the Wave Federation protocol, this wave is kept in sync with the servers of other participating users. The wave is called a remote wave on these servers.

The security model is also improved over the basic implementation in XMPP. This is mostly done to improve the verification of identities, and aims to make it impossible to perform a man-in-the-middle attack. Where XMPP only specifies security on the transport layer, the Wave Federation protocol takes this one step further and provides end-to-end security between users [22, 32].

2.5 API's for extending Wave functionality

One of the design philosophies of Wave is that the basic functionality should be fairly simple, and just contain the essence of what is needed for any type of conversation and collaboration [30]. Instead of including functionality for special requirements directly into the platform itself, there exists API's which provides any developer with the tool to extend the product to fit their needs. This platform is called the *Extension API's*.

To let users install an extension in Wave, the developer provides an *Extension installer*. This is a manifest which specifies which back-end elements are included in the extension package, and how users can interact with them using new front-end elements. It is possible to install parts of an extension without having to package it as an Extension installer as well, but this is mostly intended as a feature to ease development.

There are two types of extensions one can develop, and both serve a different purpose. The technologies utilized to develop them, and the different possibilities one have by choosing either one of them is outlined in Table 2.1. It is important to note that one in many cases needs to

Robots	Gadgets
Coded using any language, official libraries for Java and Python.	Coded using web technologies such as JavaScript, HTML and CSS.
Runs at a centralized server.	Runs locally at the client.
There can only be one single instance of a robot in one single wave, but at the same time there can be many different robots participating in a single wave.	There can be an unlimited number of instances of a gadget within one single wave.
A robot is a participant in the wave just as any other human participant, and have the same capabilities.	A gadget have limited possibilities, and can't modify a wave, it can only access it's own content. In addition it can detect when people/robots join and leave a wave.
A robot can add, remove, and modify gadgets.	A gadget has no idea whether participants of a wave is human beings or robots, and hence can't treat robots in a special way.

Table 2.1: *The key differences between the two types of Wave Extensions; Robots and Gadgets [13].*

combine them to create a solution for the problem one tries to solve.

2.5.1 The Gadget API

As mentioned in Table 2.1, a gadget is a single block of functionality that lives inside a wave, in one or multiple instances. The gadget is shown as a border-less rectangle inside a single blip to be able to blend in with the other regular content. It reacts to user input just as any other web content, since it is based on HTML and CSS. As a side note, it is also possible to embed other content here through browser plugins, just as one would on any other HTML site. The gadget also has a way to store text, and this is hidden and only available through the gadget code. In addition to this, one can also store data that are totally anonymous on

a per-user basis. Hence one can hide data and only show it to specific users, enabling gadgets for different scenarios such as anonymous voting and games [14, 13].

There are two different ways a gadget can be added to a wave:

1. A user can add it anywhere in a preexisting wave either by using a toolbar button previously added through the use of an Extension Installer, or by adding it using the universal gadget embedder combined with the URL to the XML specification of the gadget.

2. A robot can add the gadget when it is added as a participant to the wave, or at a later time as a response to a specific event.

Difference from regular Google Gadgets

Google has a framework for embedding dynamic content on websites called *Google Gadgets*. When they set out to specify how gadgets should work in Wave, they utilized this specification. But due to the nature of Wave, gadgets here needed an expanded set of capabilities. The main difference is that a gadget coded specifically for Wave has a way to persist data. This data storage is a hidden part of the wave, and is shared among all participants (or only the creator for the per-user data-store). This makes it easy for developers, since the data of their gadget is distributed to users automatically. This also makes gadgets play nice along with the playback capabilities of Wave, since it just shows a previous snapshot of the underlying data [14].

Gadgets coded specifically for Wave also have the possibility of detecting all participants of the wave, and can also call a function to find out which user is currently viewing it.

XML Format

All the code of a gadget is packed inside an XML file. The basic structure of this file is the following:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <Module>
3    <ModulePrefs title="Example gadget">
4      <Require feature="wave" />
5    </ModulePrefs>
6    <Content type="html">
7      <![CDATA[
8        HTML, CSS AND JAVASCRIPT GOES HERE
9      ]]>
10   </Content>
11 </Module>
```

This code is mostly boilerplate code, the important thing to note is that one can add additional *Require*-tags to specify features the gadget needs. All the important code of the gadget is embedded in the *CDATA*-tag.

Storing state information

The data storage capabilities of a gadget designed specifically for Wave is based around a state-object which can store key and value-pairs. This is stored in a textual way, but to simplify it is available through JavaScript-functions. To store a change in the state based on some user input, one calls a method named *submitDelta(delta)*, which takes as input an array of the key-value-pairs one wants changed. On can also call a method called *storeValue(key, value)* to store a single value. This makes it possible to specify an atomic "transaction" of all the changes one wants committed, and Wave then makes these changes.

One of the main advantages of Wave is that users can contribute on the same message at the same time. This should also be the case for

gadgets; even though multiple users interact with it at the same time, and hence changes the state simultaneously - it should work. This is mostly handled by Wave, since two state changes performed at the same time will be resolved as long as they donŠt change the same keys. But the developer also has to take this into consideration when designing the gadget, and not assume that all changes will go through.

Another important thing to note is that one should use a callback function to do the changes in the GUI based on user input that changes the state. This delta is propagated to all the participants of the wave, including the user who made the change. Hence one can combine the code that reflects changes the user does himself, and changes coming from other users.

2.5.2 The Robot API

The Wave Robot API is a collection of tools that let developers create a robot which can act as any other participant in waves. There is no difference between human participants and robots in Wave, they have the same capabilities and there is no visual distinction between them in the user interface. Since they can be deployed to any domain, there is no way to see that a participant is a robot by looking at its address. The same applies for regular users, which will have an id on the form of *username@googlewave.com* or *username@wavesandbox.com* if they are using the service provided by Google, but can have any other address if they are registered on another Wave provider.

This similarity is intentional, since a part of the vision of Wave is that humans and robots should be able to collaborate and communicate in the same manner that humans do with other humans. Robots can then aid humans in their communication, by doing routine tasks and other things that is easier and faster to do programmatically.

Communication between Wave servers and robots

While humans communicate with Wave using a Wave client, robots communicate using a protocol to interact directly with the server. This protocol is called the *Wave Robot Wire Protocol*. It is based on *JSON Message Bundles* and *JSON-RPC* (JavaScript Object Notation - Remote Procedure Calls). Events happening at the Wave server are communicated to the robot in a message bundle, and responses to these events are transferred to the server using remote procedure calls. There also exists functionality in the protocol to initiate actions without the server first communicating an event. This is called the *Active Robot API* [15].

To make it easier to develop robots, Google provides client libraries that abstracts away the protocol and instead lets you focus on the code needed to handle events. Libraries for Java and Python is provided, but since the protocol specification is open, anyone can create libraries for any language.

Java library for events

The Java Library is based on a model where there exists an abstract class containing handler methods for all the events a robot can receive. The programmer can then easily extend this class and override any methods he wants. This effectively means that the methods that get overridden represent the list of events the robot is interested in. When one deploys the robot, a file called *capabilities.xml* is created. This file specifies which events the robot is interested in, which helps reduce the load on the Wave server since it can cache this file and afterwards just send out events to the robots that are actually interested in them. The strong coupling between the method overrides and this file makes it easy to create programmatically [16].

The different events a robot can listen to is listed in Table 2.2. To be

Event	**Description**
WaveletBlipCreated	Event triggered when a new blip is created.
WaveletBlipRemoved	Event triggered when a blip is removed.
WaveletParticipantsChanged	Event triggered when participants are added and/or removed from a wavelet.
WaveletSelfAdded	Event triggered when a robot is added to a wavelet.
WaveletSelfRemoved	Event triggered when a robot is removed from a wavelet.
WaveletTitleChanged	Event triggered when the title of the wavelet has changed.
BlipContributorChanged	Event triggered when contributors are added and/or removed from a blip.
BlipSubmitted	Event triggered when a blip is submitted.
DocumentChanged	Event triggered when a blip content is changed.
FormButtonClicked	Event triggered when a form button is clicked.
GadgetStateChanged	Event triggered when the state of a gadget has changed.
AnnotatedTextChanged	Event triggered when text with an annotation has changed.

Table 2.2: *Events that can be catched using the Java Client Library method overriding. Information from [16].*

able to do something in response to the event, the robot needs data about the blip, wavelet or participant the event has happened to. A way to reduce bandwidth is to be specific about what scope you need data about. Table 2.3 summarizes the different scopes a robot can ask for when it gets an event. The default scopes that will be delivered if nothing are specified are *PARENT, CHILDREN, ROOT, SELF*.

Scope	Description
PARENT	Indicates that the event should pass the parent data. Note that PARENT makes no difference to Wavelet events.
CHILDREN	Indicates that the event should pass any children of the event's level. For Wavelets, this context passes all child Blips.
ALL	Indicates that the event passes all associated data.
SIBLINGS	Indicates that the event passes any siblings. For Blips, this context will pass data for all sibling blips within the Wavelet.
SELF	Indicates that the event only passes information pertaining to itself.
ROOT	Indicates that the event only passes information pertaining to the root blip.

Table 2.3: *The different context scopes an event can return data for. Information from [16].*

Handling of operations

In response to events, the robot can do any number of operations on the wave's content. All operations are performed on the server, so information about the desired changes have to be transferred back to the server before they are applied. This is made easy when one uses the Java client library. All the changes are performed as if they were done locally, using text editing methods. These methods don't change the wave immediately; instead they act as a gateway, translating calls to them into a JSON format which is then sent to the server for execution.

All operations are done on individual blips, and is basically simple text altering operations. There are two parts to this, first one needs to apply a *selector method* to choose a subset of the text, and then one uses an *action method* to alter the text. The following selector methods can be used:

- **all(element/text)** selects all instances of the matching text string or element (e.g. a gadget type).
- **first(criteria)** selects only the first instance of the matched criteria.
- **at(index)** selects a specific position in the text, zero-indexed.
- **range(start, end)** selects a specific range in the text, zero-indexed.

After selecting a subset of the text, one can then apply one of the following methods:

- **insert(element/text)** inserts the element or text passed at the start of the selection.
- **insert_after(element/text)** inserts the element or text passed at the end of the selection.
- **replace(element/text)** replaces the selection with the passed text/element.
- **delete()** deletes the selected text.
- **annotate(name,value)** annotates the selected text with the passed name and value-pair. This can either be a hidden meta-data about the selection, or can alter the visible appearance.
- **clear_annotation()** removes any annotation for this specific selection.
- **update_element(value)** performs an update of the element at the selection with the passed value.

As an example of how this works, these three examples shows adding a gadget at position 12, replacing the text between position 4 and 8 with *replacementText*, and last replacing all instances of the word *foo* with the word *bar*.

```
1 blip.at(12).insert(elements.Gadget(URLTOGADGET));
2 blip.range(4,8).replace("replacementText");
3 blip.all("foo").replace("bar");
```

The Active Robot API

The first version of the Robot API was only capable of responding to events. With version 2.0 of the API, it's also possible to initiate actions based on events happening outside of the Wave platform. This new technology is called the Active Robot API. To be able to do this securely, one need to register the Robot with the Wave server to be able to exchange cryptographic keys. This makes the server able to identify the robot, and then only accept change requests from the trusted robot server. When authenticated, the robot then has the same capabilities as when it responds to a specific event. In addition, it can also create entirely new waves.

Hosting robots on a server

There are two possibilities for hosting robots. The first is to host the robot on your own server. By doing this, you can use any language to code the robot, by manually handle the protocol. You can also download either the Java or Python Robot Library, and deploy that on the server.

Another possibility is to host the robot on Google's infrastructure. This solution is called Google App Engine. Developers of robots sign up for this service, and will be able to register up to 10 different domains for free, each which can run one single robot. The URL's are in the form *selectedName.appspot.com*, and the created robot then gets the address *selectedName@appspot.com*. This environment is free for simple uses, but for large scale robots, one needs to buy a plan that gives additional bandwidth. The tools provided integrate well with Eclipse, and deployment of new code is as easy as a single click.

Chapter 3

User service composition in Wave

This chapter starts off by looking at the general idea of user service composition in Sect. 3.1. We then move on to look at how this principle can be applied to the field of communication and collaboration in Sect. 3.2. In Sect. 3.3 we take a look at how one can find existing components to utilize in waves. Lastly, we create two examples of compositions users easily can make using Wave: an example of collaboratively proposing and deciding between selectable objects in Sect. 3.4, and a exploration of the different compositions developers can create to utilize Wave to plan software development in Sect. 3.5.

3.1 User service composition

We use the term "user service composition" as a way to describe systems that let users set up the system in different ways according to their needs. Instead of designing a "one suit fits all"-system that are minimalistic and just suits the basic needs of the user, or doing the opposite and designing a comprehensive solution that tries to solve everything, a better approach is to design a user configurable system. This makes the solution easy

for new users to grasp, since the basic functionality will be fairly simple. At the same time, it makes the system a good solution for power users, since they will find it easy to add additional functionality as their needs demands it. User service composition also makes the solution grow with the users, hence as novice users grow into power users, the service grows with them, and they can stay with the solution instead of switching to gradually more complex solutions.

Wave is based on the idea that it should be fairly simple and minimalistic, but as previously discussed in Chapter 2, there are different types of APIs and extension points that lets developers create plug-ins to the system. Since these possibilities exists, one can look at Wave as a system that lets users compose services to suit their needs. One can look at Wave as the culmination of user service composition-systems, since Wave lets each user compose each new wave in an unlimited number of ways. Hence there are almost an endless number of possible tasks Wave can be used for.

3.2 Using service composition in communication

Today, online communication and collaboration are based around a lot of different systems that each has their different strengths and weaknesses. The most prominent examples are e-mail, which provides a very simple text-communication, and instant messaging that provides real-time textual communication, often only between two parties. There are also a lot of different systems that are based more around the idea of collaboration instead of communication, like online wiki systems, document collaboration, and various other systems specifically tailored to suit different business needs.

Wave is an attempt to unify these different types of systems into one core communication and collaboration platform. The only sensible way

to do this was to make the core product fairly simple, but at the same time create many points where developers can add extra functionality that enriches the platforms capabilities. The way this was done was to let the actual users select which components they want to add.

Users of Wave can install extensions into their Wave account, making them easier to access later. The actual usage of the different extensions happens in a ”per wave”-fashion. This means that the user service composition is done every time the user needs to initiate communication and collaboration with some other people. The user who initiates the communication starts the wave, adds the desired components needed for the communication, and invite other users to participate in the wave. If another participant has write-rights to the wave, he can also add additional components to the wave. This makes Wave a truly multi-user service composition system.

To automate the process even further, there is possible to define templates one can easily use to initiate a specific type of communication or collaboration. This makes it easy to define which user selectable components one needs one time, and then having the whole service composition process automated every other time the user needs this set of tools in the future.

Another way to automate the process is to include robots that can do some of the repetitive tasks of configuring the service composition. These robots can be configured to respond to certain events, and can hence improve the communication and collaboration by automatically adding service components into the wave when it deems them necessary.

3.3 Existing solutions: The Extension gallery

As mentioned in Sect. 2.3, Wave is largely based on open source solutions. This also applies to the extensions developers can create for user service

composition. There are no formal requirements extensions have to comply to before they can be used inside Wave, the only requirement is that they resist on a server that are accessible by Wave. This means that developers can host their gadgets on their private server, and just provide users with an URL to the extension. The same applies for robots.

Even though the ecosystem surrounding extensions are free and open, Google have provided a solution for developers who want to spread their extensions to end users. This solution is called the Extension Gallery. Basically, this is just an online directory of extensions that developers have submitted which is then checked and approved by Google. By browsing through this gallery, one can find extensions for many different purposes. This makes it easier for users to be able to perform service composition, since they just have to find existing extensions that suits their needs, and by doing this don't have to create them themselves.

3.3.1 Different ways to access the content in the Extension gallery

As of today, the Extension gallery provided by Google is mostly meant for developer use. The reason for this is that Wave and Google Wave still is in the early phases of development. The version of Wave users can use is just a preview-edition, and hence the whole ecosystem surrounding the platform is not ready. For these reasons, the Extension gallery is mostly focused on the idea that developers should use it to share ideas and techniques. As Wave matures, so will the Extension gallery, and features targeted at end users will be developed [24].

This means that there today are quite a few different ways one can access the content in the Extension gallery. Not all extensions have all the possibilities, and as the platform matures some of these possibilities will most likely disappear for some extensions. The different ways to access extensions are:

- **Textual description, video preview and user comments:** Each extension have its own page that provides the necessary information about the extension.

- **Live demo:** This provides a link to a public wave containing the robot or gadgets that the extension consists of. This lets you test out the extension directly.

- **Source code:** For gadgets, the source code always have to be available, since gadgets are based on technology that runs on the end users computer. For robots, the inclusion of the source code is not necessary since it runs on a server, but many extensions in the gallery provides it nevertheless.

- **Gadget XML URL / Robot Address:** For gadgets, the user needs the URL to the XML describing the gadget to be able to install it manually. Robots use an address resembling an e-mail address to add them to a wave.

- **Extension installer:** Many extensions also provides an Extension installer through the Extension gallery. This is just a link to a special form of wave, that lets the user install custom icons, templates and contacts to their Wave account. This is often the preferred way to add an extension for most users, since it is most user friendly and also plays nicely with the integrated extension management solution in Wave.

By using the different ways to add extensions one finds in the Extension gallery, one can combine the work of many different developers into a totally user composable service solution.

3.4 Example composition 1: Collaborative movie night voting

Consider the scenario that a group of people should all attend an event where one has to select between multiple possible activities. One simple example of this is a ”movie night”. Everyone wants to go see a movie, but you need to decide which film you are going to see. For this purpose, online communication is good. But by using a system that lets you compose the service you need to make the process specifically tailored to your need, you can make the process easier.

By doing this in Wave, there are a few pre-existing components you can include:

- **The IMDB Robot:** This is a robots that automatically transforms any IMDB-link (The Internet Movie Database) into a nice short summary of the movie, containing the poster, rating and prominent actors. By integrating this information, people can make their choice just by glancing at the wave. And if they need more information about the movie, they can just follow the link.

- **The Rating Robot:** This robot automatically adds a system that lets the participants of the wave easily vote on the different wavelets in the wave, either by pressing a +1 or a -1 button.

By including these two robots, and then write a short text instructing the participants to paste links to movies they propose in new wavelets, you have composed a complete system for deciding between movies. The result of a wave like this is shown in Fig. 3.1

This solution is also fairly customizable. You can use the same approach to decide between any type of items, just by including the rating robot and then instructing the participants to post each proposal as a

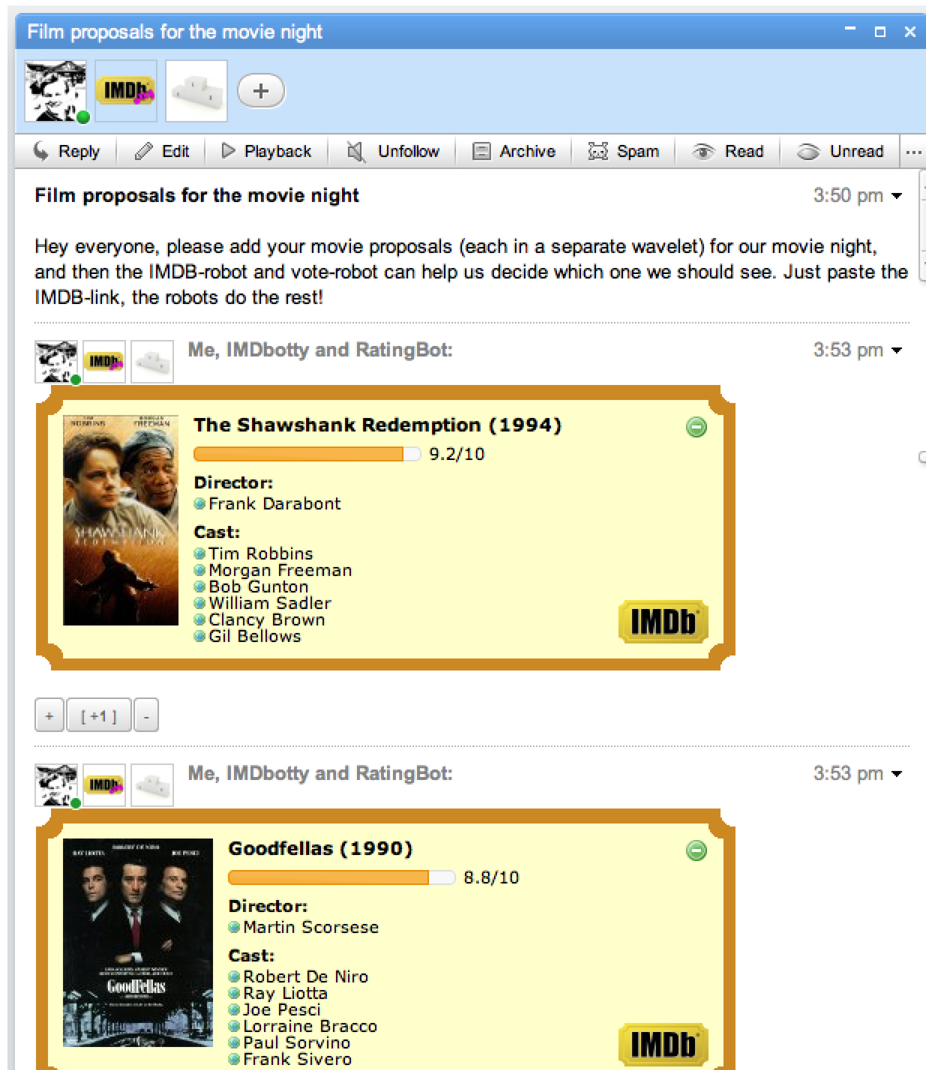

Figure 3.1: *Screenshot of Wave user composition Use Case: Wave used for selecting films for a movienight.*

single wavelet. Either one can just use textual proposals, or one can incorporate robots that fetch information from other external sources to help the participants take informed choices.

3.5 Example composition 2: Wave as an aid for developers

Developers use a wide variety of tools to keep track of development, discuss potential new features, and keeping track of who does what and when the various tasks should be finished. Bug tracking systems, physical and digital "scrum-boards", meetings to discuss new features, interacting with users using a variety of channels, and systems for monitoring progress is just a few examples of the different needs they have to find a solution to.

A user composable system would be ideal for developers. For once, developers are the pickiest about the software solutions they use - they won't put up with badly working solutions. Also, developers are the most suited group to pioneer this sort of solutions, since they are the most technically competent users out there, and hence is a good group to beta-test a new system for user composition.

By using a variety of extensions from the Extension gallery, and probably also implementing a few themselves to suit their exact needs, developers can gain a lot from using Wave. Shown in Fig. 3.2 are just a few possible solutions that can be integrated into one single wave to help out in the development of a single new feature for a system. The gadgets shown are:

- **A digital scrum board:** This gadget lets the participants add user stories to the backlog, and also add different sub-tasks to each user story. The tasks can be assigned to different user, and moved along the scrum-board by using drag and drop. This gadget fills most of

the needs of developers using the scrum process [31].

- **A progress indicator:** To track progress of the feature, a progress indicator can be added, making it easy to change the status using only a single click. Creative developers can extend this gadget to suit their needs better, for example by making it track the progress automatically based on some sort of input, either from another gadget in the wave or from an external system using the Active Robot API.

- **Polling gadget:** When the team needs to decide something, they can include a gadget that facilitates the decision making. There are many different solutions for this in the Extension gallery, an example is anonymous polling.

Not shown in Fig. 3.2 is the following extensions types that also can aid in a developer wave:

- **Translation robot:** By utilizing a robot that automatically translates the blips of different users into a common language, developers from multiple countries can all work together without having to use external translation services before they can communicate. Some robots even include real time translation features, meaning that users who speak other languages can participate in the instant communication Wave provides.

- **Integration with bug tracking systems:** There are different active robots in the Extension gallery that automatically creates new waves as bugs are created in the external bug tracking system. This makes it easy to use Wave as a communication platform for the bugs, but at the same time maintaining the legacy system.

Figure 3.2: *Screenshot of a wave used by developers to collaborate on a new feature in a software system.*

- **Integration with e-mail:** Software companies often have existing solutions for dealing with user requests. By integrating Wave with e-mail, one can automatically get mails users send into Wave, to make it easier for the developers and support staff to discuss and collaboratively come up with a suitable response. Sending out the response can also be done automatically from inside Wave.

As these examples show, the possibilities of Wave are almost endless when it comes to simplifying a developer's workflow. And by having the developers using the system, the system just gets better, since this is this group that is capable of extending the system and hence make it even more suitable for normal users.

Chapter 4

Organizing meetings collaboratively

In this chapter, we tackles the problem of collaboratively scheduling, planning and holding physical meetings. We explain the problem in greater detail in Sect. 4.1, and look at existing solutions in Sect. 4.2. Next, we look at some of the advantages can gain by using a highly collaborative approach in Sect. 4.3. In Sect. 4.4 and Sect. 4.5, we explain the process of developing two gadgets for the Wave platform that can aid in the process of solving the problem, and in Sect. 4.6 we explain how we integrate these gadgets with the Wave platform.

4.1 What do we want to solve

In all kind of collaborative work, having physical meetings between different people on the teams are of crucial importance. Even though it now exists ways to meet virtually, the old form of a physical meeting is still used in a large fashion. And even when the meetings takes place using

chat or video conferencing tools, many of the old necessaries still exists.

There are essentially three phases of a meeting: the preparation before the meeting, the actual meeting, and the aftermath of the meeting.

4.1.1 The planning phase

To be able to hold a successful meeting, one needs to do some preparation. The first and foremost problem is determining who will attend the meeting, and then sending out invites. In doing this, one can either take an approach where one determines a date and time beforehand, and then makes the people who have this time slot free attend. Another approach is to discuss with the different participants who are attending the meeting to find a suitable time where most people can attend.

In addition to this scheduling problem, one also needs to plan what one should talk about in the meeting. Often, it is the meeting scheduler who decides on the agenda for the meeting, and if others need to come with proposals for agenda items, they have to coordinate this with the scheduler in some way.

4.1.2 The meeting phase

When it is time to hold the meeting, there are a few requirements that often is put in place for the meeting to be successful. The first is that the schedule should be presented to all the participants in some way. Often, this schedule is sent out beforehand as well, but it also needs to be present at the meeting.

During the meeting, it is often preferable that someone takes notes. This is useful both for reviewing the meeting afterwards, to be able to have a way to remember everything that took place in the meeting, but also serves as a way to let people who could not attend the meeting get up to speed on the matters discussed.

It would also be ideal to have a way to note down the progression of the discussion. By doing this, it is easier to grasp not only the end result, but also the progress that made the group take that decision. Sometimes, it would also be useful to have systems in place that facilitates the meeting. Examples of this is solutions that lets the participants vote on ideas (open or anonymously), and tools that facilitates brainstorming.

4.1.3 The review phase

After the meeting, one need to distribute the notes to all parts who should have access to them, and might also need to review the results and have a way to store them for later retrieval. Some of the more formal meetings might also need to have some kind of documentation system put in place, that ensures all parties that votes and decisions which happened during the meeting were conducted in the right way.

4.2 Existing solutions

The problem of handling meetings in an efficient manner has been tackled by a lot of different vendors. There are different existing solutions for the various phases outlined in Sect. 4.1. To be able to determine what to look into in our solution, we will analyze the different solutions to find their strengths and weaknesses.

4.2.1 Clientside software solutions

There exists a wide variety of solutions for scheduling meetings based on e-mail. These are often integrated in the client-software that handles the e-mail. Microsoft Exchange (with the client software Microsoft Outlook) and Lotus Domino (with the client software Lotus Notes) are examples of these types of solutions. These systems is largely based around the idea

that one person should be responsible for handling the practical details around the meeting, and should take care of the different aspects of the organization manually [28, 27].

Systems like these are often well suited for use within one organization, since the details of all employees' calendars and contact details lie inside one system. The organizer can then use tools to compare the calendars of all the desired participants to determine which time slots is available in each person's calendar. By manually determining a time where most of the participants are able to attend, the scheduler can then pick a time and send out an invitation. These sorts of systems then rely on e-mail-replies to let the attendees RSVP and the system will parse these e-mails in a special way to keep the attendee list updated with the statuses.

To handle the agenda, the person responsible for the meeting needs to get proposed topics from the participants either by e-mail or by some other manual fashion. Before the meeting, this one person needs to take all the different proposals and combine them to create the agenda.

After the meeting, this solution is also mostly based on the e-mail format. A common practice is for one person to be responsible for taking notes, and later distributing these to the other participants.

4.2.2 Web-based solutions

Quite a few vendors have created different web solutions to solve the problem of successfully plan and hold meetings. Some of these solutions are specifically targeted at one of the aspects of the process, while others try to be more of a universal solution.

Doodle is an example of a free web based product that tries to solve the scheduling problem. It lets the creator propose a lot of different time slots, and then provides a link which can be sent to all the proposed participants. When they go to the site, they will be presented with the different options, and can pick the ones that is available to them. After

all participants have gone through this procedure, the person responsible for the meeting can see the answers, and pick the time where most of the participants can join. The rest of the planning is done manually [9].

There exist quite a few different tools that can be used to collaboratively create agendas and notes. Google Docs is an example of this, and with the recent updates it lets participants concurrently work on the same documents without big problems with syncing of the different changes [6]. The process of sharing the documents and keeping all the different resources created for a single meeting organized can become a burden, since Google Docs in mostly focused on creating documents, and not maintaining a set of resources for a single event.

4.3 Advantages of using a collaborative approach

We believe the task of scheduling and executing meetings can be greatly improved by using a highly collaborative platform like Wave. The core idea of having a meeting is to get multiple people together to collaborate on some task. To not let all of the participants collaborate in the same manner on the digital platform hinders the whole idea of collaboration.

The solutions one have today makes the usage of digital tools very uncollaborative, and hence it has become a norm to not use them during the meeting, since this have a tendency to destroy some of the collaborative nature of meetings. But by making the tools highly collaborative, this trend can be turned. The goal has to be to make the tools so good that they instead enrich the meeting.

There are some few key areas where today's tools need improvements. These are:

1. **Collaborative scheduling:** If one can make the scheduling solution simpler than the old process of manually picking a time, this solution will be preferred. We want a single solution for picking the

attendees, and then have the scheduling solution set up automatically, with the invites automatically distributed, and the solution for responding also automatically integrated.

2. **Let the user be in control, but aid in the process:** The process of deciding a date after getting all the responses should still be done manually to let the user feel in control and give room for unplanned changes, but the task of picking the time and informing all participants should still be as easy as a click on a button.

3. **Collaborative agenda within the solution:** By making it easy for participants to give inputs to the agenda, and also collaboratively decide on what should be included, they will feel more of an ownership in the meeting. This makes the meeting seem more like a group-work, rather than something that one person has organized.

4. **Make it easy to take notes:** Instead of designating one person to take notes of the meeting, one should instead distribute this work among the participants. The solution should be able to automatically create an outline based on the agenda, and make it easy for each and every participant to add their bits to the notes. By making the solution intelligent and adding in features that drags in relevant data from external sources, the notes gets better and the meeting can also progress faster since one immediately gets access to additional data. The solution can also aid in different cases, for example by providing a way to hold anonymous votes right inside the notes.

5. **Make the distribution of material automatic:** By integrating all the functionality inside one solution, one greatly eases the task of distributing material. When one has picked the participants in the beginning, the solution should automatically distribute everything to everyone.

6. **Keep record of everything:** Since everything is integrated in one solution, there should be fairly simple so retain the information for later reference.

4.4 Gadget for time scheduling

The solution we end up creating is based on Wave. To be able to easily schedule the meeting, we create a new gadget that can be included in a wave to facilitate time scheduling. As described in Sect. 4.2.2, the website doodle.com has come up with a great user interface for voting on different time slots. We want to use some of the lessons learned from studying this solution when we create the gadget in Wave.

First and foremost, we can see that a square representing each person for a single timeslot is a good user interface paradigm. By having the different time slots along the horizontal axis, the participants along the vertical axis, and having each person-timeslot relation in the space between these two axises, we manage to condense a lot of complex information into a small and easily graspable space.

We also want to use the universally acceptable colors red and green, where red is negative and hence means that the person can't attend, and green means that they can. By sticking to these conventions, we can make the user interface uncluttered by not having to include that much help-text and descriptions. We also include a gray square which just means that the user hasn't made a choice yet. This color is very modest, and just serves as a way to keep the grid intact even before all users have decided upon all timeslots.

One problem with Doodle is that there is no concise way to see who haven't answered the poll yet. Their solution shows just those who have answered. This limitation is mostly based on the fact that the actual sending of invitations is handled outside of the system. In the scheduling

gadget we develop, we definitely want to tackle this problem and create a solution that lets you easily grasp who have and who haven't answered. This means that we have to grab the list of persons from the wave participant list, and show them all, even if they haven't replied yet.

One important aspect of Wave is that everything can be changed at any time, and that all changes are kept track of in a history that anyone can look through at a later time. Hence we want our gadget to be able to let users change their status for the different timeslots at any time. If they can't take a decision on one timeslot when they receive the invitation to the meeting, it's better to let them make their decision on all other timeslots and still let them be able to keep that timeslot undecided. One potential problem of letting all users change their status at any time is that they can change their minds after a decision have been made regarding the time of the meeting. But by having a tight integration with the history feature of Wave, one can just go back in time and look at the history to fix any misunderstandings. This problem can also be easily solved using some lines of JavaScript.

4.4.1 Underlying data structure

To be able to store the data about the choices users makes regarding the different time slots, we need a data structure. As outlined in Sect. 2.5.1, Wave provides a simple solution to store state information from a gadget. This information is hidden from the users, and can only be accessed by the Javascript-code of the gadget. The way one stores data is to create a structure consisting of key-value pairs, and then calling the *submitDelta*-function to store them in the gadgets data store. The following example shows how one sets (or updates) the key *attending*.

```
1 var stateChange = {'attending': 'no'};
2 wave.getState().submitDelta(stateChange);
```

If you just need to change one value in the state store, you can use a simpler function where you don't have to initialize a stateChange-object:

```
1 wave.getState().submitValue('attending', 'no');
```

One major limitation of Wave's current state-storage mechanism is that it only supports strings as values. This means that there is no simple way to store a structured data set, like an array or an object. We have to find some way to serialize the data structure we create to be able to store it in a string.

As we discussed in Sect. 2.5.1, Wave provides good concurrency handling, meaning that many users can interact with the gadget at the same time as long as they don't change the same keys in the underlaying data structure at the same time. This means that if we store all information in one complex data structure, and then serialize this and assigns it to one key, we break this concurrency handling. We want to avoid this, and hence have to take this into consideration when we design the data structure.

The ideal solution would be to have one key for each user; then we can guarantee that the system will work with any number of concurrent users, as long as each user is just logged in from one location at any given time. We can indeed achieve this in this case, by storing each user's attendee statuses in separate keys. The code to create and store the data structure is the following:

```
1 var currentUserID = wave.getViewer().getId();
2 var userState = {};
3
4 //Add info about each timeslot to userState
5
6 var serializedUserState= JSON.stringify(userState);
7 wave.getState().submitValue(currentUserID, serializedUserState)
    ;
```

In the code above, we first find a suitable key for each user's state store. We just use the user ID fetched from wave, since this will be unique for each user. We then create an object that can hold the key-value pairs consisting of keys for each timeslot and values representing the attendee-status of the user for that timeslot. Here we don't assign any statuses, meaning that each timeslot will be shown with gray. This is the initial state of each user before they have made any decisions.

We then serialize the object to be able to store it as a string. We use a serialization that converts the object to a JSON string. This is a standard serialization, specified in IEEE RFC 4627 [8]. As of today, most browsers have support for serialization and de-serialization of JSON through native JavaScript functions. We could have used a fall-back technique to support browsers that doesn't include this, but since the current Wave client doesn't support these browsers anyway, this is not necessary. Lastly, we store this string in the key for the user.

The data structure for the different timeslots will have to be accessible to all participants of the wave. But since this is not changed nearly as frequently, this is an acceptable trade-off. To initialize this structure and store it, we use the following code:

```
1 var newTimeslotTable = [];
2 var serializedTable = JSON.stringify(newTimeslotTable);
3 wave.getState().submitValue("timeslots", serializedTable);
```

This code is mostly a simpler version of the code for the user state storage, only that we here uses a table instead of an object, since we only have to store single values, not key-value pairs. And we store it in the *timeslots* key instead of a dynamic key that changes according to which user is logged in. When a user adds a timeslot, we will go through the following procedure:

```
1 //Find the timeslot to add
2 var form = document.getElementById("newTimeForm");
```

```
3  var newTimeSlot = form.newTime.value;
4
5  var serializedTable = wave.getState().get('timeslots');
6  var timeslotTable = JSON.parse(serializedTable);
7
8  timeslotTable.push(newTimeSlot);
9
10 serializedTable = JSON.stringify(timeslotTable)
11 wave.getState().submitValue('timeslots', serializedTable);
```

Here we see the process one have to go through each time one wants to change something in a serialized data structure. One have to find what to change in the data structure, then fetch the serialized version from the key and de-serialize it. Then one does the desired changes to the data, and lastly serializes it and overwrites the old value of the key. The process of changing a user attendee state is mostly the same, just with a bit of complexity added to find the current user.

4.4.2 Calculating the number of possible attendees

A needed feature is to be able to easily see how many users can attend the meeting at each proposed timeslot. One would think that this should be a fairly trivial calculation to do, but since we have spread the data out in many different keys to achieve great concurrency handling, we have to do a fair amount of data fetching to calculate this number. The following code does the task:

```
1 function countParticipants(timeSlots) {
2     var participants = wave.getParticipants();
3     for ( var t in timeSlots ) {
4         var count = 0;
5         for (var p in participants) {
6             currentParticipantId = participants[p].getId();
7             var userState = JSON.parse(wave.getState().get(
                  currentParticipantId));
```

```
8             if(userState[timeSlots[t]] == 1) {
9                 count++;
10            }
11        }
12        //Output or store the count for this timeslot
13    }
14 }
```

This code loops through all the timeslots to calculate how many users have marked this as green (meaning they can attend). For each timeslot, we have to loop through all the participants to check if they have marked it green, and then increase the count by one. Lastly we need to reflect this in the GUI, this code is omitted since it is mostly trivial HTML generation.

4.4.3 The final gadget

When one has the underlying data structure ready, most of the remaining work is to create a GUI that can be used to access and change the data. All the details of this are too much to include in this thesis, but most of is fairly standard HTML and CSS, plus a fair bit of JavaScript to make it as dynamic as possible. The result can be seen in Figure. 4.1, and by installing the complete work of this thesis as explained later, you can also test it out in Wave.

One important thing to note about the way the GUI is created is the decision made about when one updates the GUI to reflect user actions. There are two times the GUI needs to be updated: either when the user himself does some changes it needs to react to, or as a result of someone else also participating in the same wave doing some changes. Wave is meant to be highly cooperative, and results of other usersŠ actions should immediately be reflected in the GUI. Hence we have chosen to do all GUI updates as a result of changes coming from the Wave server, and none as a result of local events.

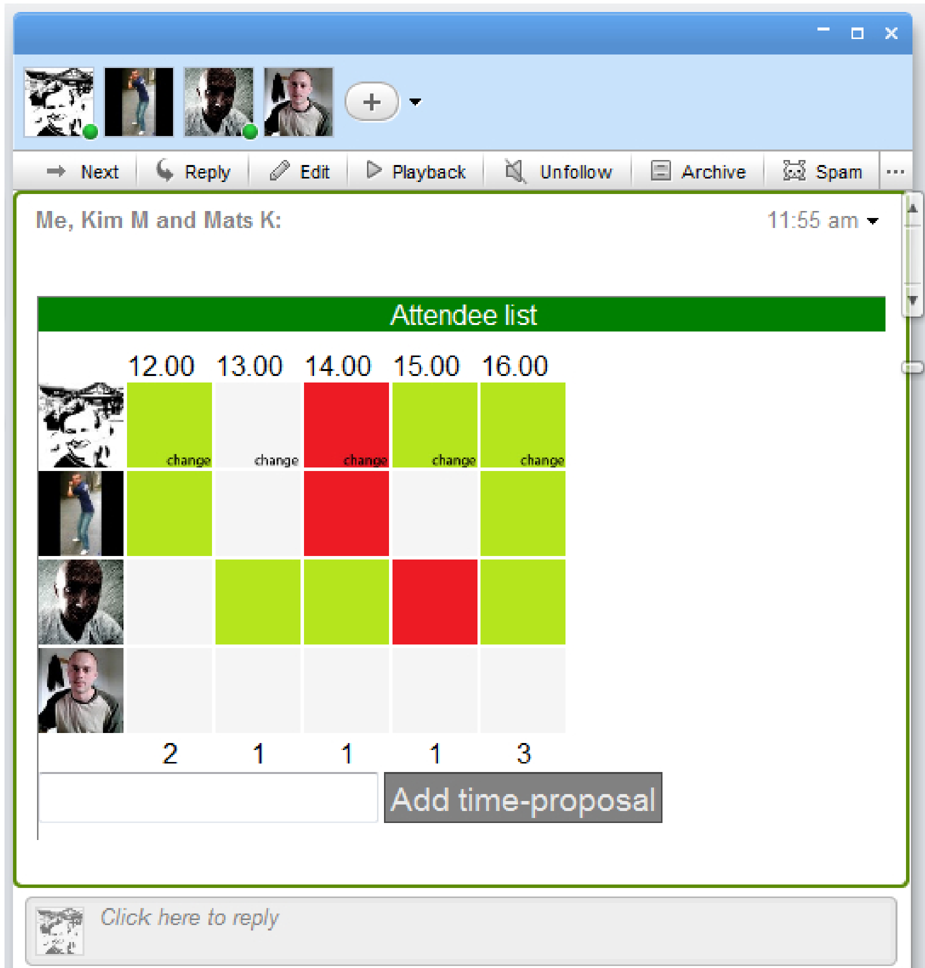

Figure 4.1: *The final gadget for doing time scheduling*

What this decision means in practice is that we only have to have one set of functions that updates the GUI. When a user does something that causes some changes to content of the underlying data structure, the action handler ships these changes to the Wave server, but doesn't touch the GUI. These changes are then propagated to all participants of the wave, including the user who caused them. Then we can have a single point of interaction with the GUI. It doesn't matter if you are the user who caused the change or if you just receive them, both users will see the changes reflected at the same time (the network speed is the only difference).

What this all means it that we have to write the code that sends the changes to the server in a way that makes this process fast. One would think that this roundtrip to the server causes the GUI to be sluggish and unresponsive, but in practice this is not happening as long as one writes fast and concise code.

4.5 Gadget for a collaborative agenda

As we have discussed earlier in this chapter, the approach taken by many meeting organizer tools today is that one person have to be responsible for the agenda, and have all other participants send him or her suggestions, which is then incorporated into the agenda by this one person.

We want to take another approach, where the agenda is created collaboratively by all the participants of the meeting. All participants should be able to see the agenda as it is created, and be able to add new items to it as they see fit. Since Wave has powerful features for recapping the history of a wave, we should be able to give people the power to do just about anything to the schedule such as deleting items others have added, with the confidence that one always is able to revert to an earlier state should someone misuse their rights or accidentally delete something.

Our solution will consist of a gadget that is fairly simple. It will list all the items proposed by the different participants, and also allow anyone to add new items to the list. It should also be possible to delete items from the list, and anyone should be able to do this, regardless of who originally added the proposal.

The user interface of the gadget should be fairly simple to not overwhelm the user with too much needless information. In its simplest form, it should just list the ideas, and hide all the details. These details can be stored for later use, but should not be shown here.

As with the scheduling gadget, we have to be careful when designing the backend of the gadget, to make sure it conforms to the architectural principles of Wave regarding concurrent interaction with it by multiple users and the gadgets ability to integrate well with the history-showing replay-feature of Wave.

4.5.1 Underlying data structure

For the agenda gadget, we create a fairly simple data structure to hold the agenda. We just need a few pieces of information about each agenda proposals, namely the text of the proposals itself, and some metadata about it, such as who created it and when it was created.

We will extend this gadget in later chapters when we introduce a robot into the solution, so we need to keep that in mind when designing the initial simple data structure. By using JavaScript objects for the storage of each proposal, we are free to extend the data structure object as we see fit at a later time.

For this gadget, the best solution is to let a single key hold the serialized array of the different agenda items. The decision to do this has some advantages, namely making it much simpler to do all the operations on this list. One trade-off is that we lose some of the automatic concurrency handling Wave provides. But since there is no way to store arrays in the

underlying gadget data structure, this is a trade-off we have to live with.

When we create a new agenda item object as a response to a user writing in the title and pushing a button, we do the following to fetch all the relevant data:

```
1 var agendaItem = {};
2 agendaItem.agendaItemName = form.newSak.value;
3 agendaItem.userName = wave.getViewer().getDisplayName();
4 agendaItem.userThumbnail = wave.getViewer().getThumbnailUrl();
5 agendaItem.timeSubmitted = wave.getTime();
```

Here, we first create a new empty object to hold all the data about the agenda item. We then populate it with data, first the value the user entered into the text field representing the description of the agenda item, then the username and thumbnail picture of the user, and lastly the current time.

As we discussed in Sect. 4.4.1, there are not much support for data structures in the underlying storage model for gadgets in Wave. We hence have to use a serialization method to be able to represent our data structures as strings in the key-value data store for the gadget. Refer to Sect. 4.4.1 for all the details about this procedure. As explained there, we fetch the existing data and deserializes it, adds the new data to this, and then serializes it again and stores it back into the same variable. The code for this procedure is the following:

```
1 var existingItems = JSON.parse(wave.getState().get('agenda'));
2 existingItems.push(agendaItem);
3 wave.getState().submitDelta({'agenda': JSON.stringify(
      existingItems)});
```

4.5.2 The final gadget

With the underlying data structure in place, we can easily create a GUI to present the agenda to the user. We simply loop through all the agenda

items in the data structure, and show them in the gadget. The code for this is the following:

```
var items = JSON.parse(wave.getState().get('agenda', '0'));
for ( var i in items ) {
   //Show this agenda item in the GUI
}
```

As we explained in Sect. 4.4.3, all the code for updating the GUI is located in the method that gets called when the Wave client receives an update from the server. Hence we can write one single GUI code that responds to both other users' actions and our own commands to the GUI. This also means that we can be certain that the changes we did were propagated to the server when we see the change in the GUI.

The final gadget is shown in Fig. 4.2. Here we see the simple view the user sees when he isn't interacting with the wave. As we can see, all the details about each agenda item is hidden, and only the most important thing, namely the description of each item, is shown.

4.6 Making it all come together

To make it simpler to utilize these features, we need to make it possible to "install" them into the Wave client. The solution to this is to make an Extension installer. As discussed in Sect. 2.5, this is an XML file that specifies what GUI-elements to add in the Wave client, and how these interact with the back-end. This XML file is then used as input for a specific wave template, which creates a wave containing the description, instruction and an install link for the extension package.

The specification for the extension installer XML file format includes a lot of different possibilities regarding how one can extend the Wave client. We will only be using one of these possibilities for these gadgets, namely the function that lets you add gadget-inserting toolbar icons in

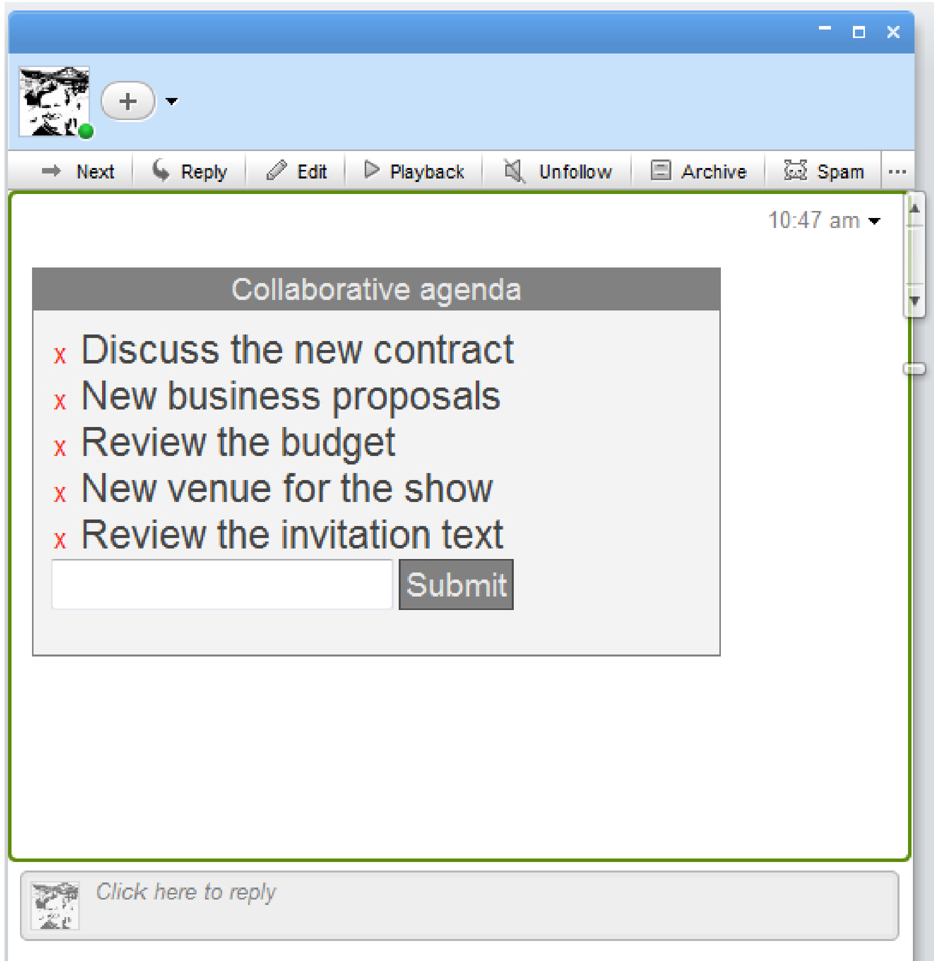

Figure 4.2: *The final gadget for collaborative agenda creation.*

the compose window. The complete XML for the extension installer is the following.

```
1 <extension
2  name="Meeting Extension"
3  thumbnailUrl="http://folk.ntnu.no/espenher/wave/All-mail.png"
4  description="Lets you schedule, plan and have meetings using
       Wave as the collaboration tool.">
5      <author name="Espen Herseth Halvorsen"/>
6      <menuHook location="toolbar" text="Add scheduling gadget"
7       iconUrl="http://folk.ntnu.no/espenher/wave/calendar.png">
8          <insertGadget url="http://folk.ntnu.no/espenher/wave/
               doodleGadget.xml"/>
9      </menuHook>
10     <menuHook location="toolbar" text="Add agenda gadget"
11      iconUrl="http://folk.ntnu.no/espenher/wave/clipboard.png">
12         <insertGadget url="http://folk.ntnu.no/espenher/wave/
               saksliste.xml"/>
13     </menuHook>
14 </extension>
```

This XML file first defines the extension, and gives it a name, thumbnail picture and description. This meta-data will be used by the Wave client to display the extension nicely in the GUI. Then we define the extension author. The next two parts are the menuHooks, i.e. what the extension adds to the Wave client. In this case, we can see that we add two new items to the menu. We specify the location to be the toolbar, which is the set of icons over the wave when it is in edit mode. We specify an icon, a tooltip text and the actual address to the XML file of the gadget.

Next, we use a specific template to input this XML, and this creates a nicely formatted Extension installer wave as shown in Fig. 4.3.

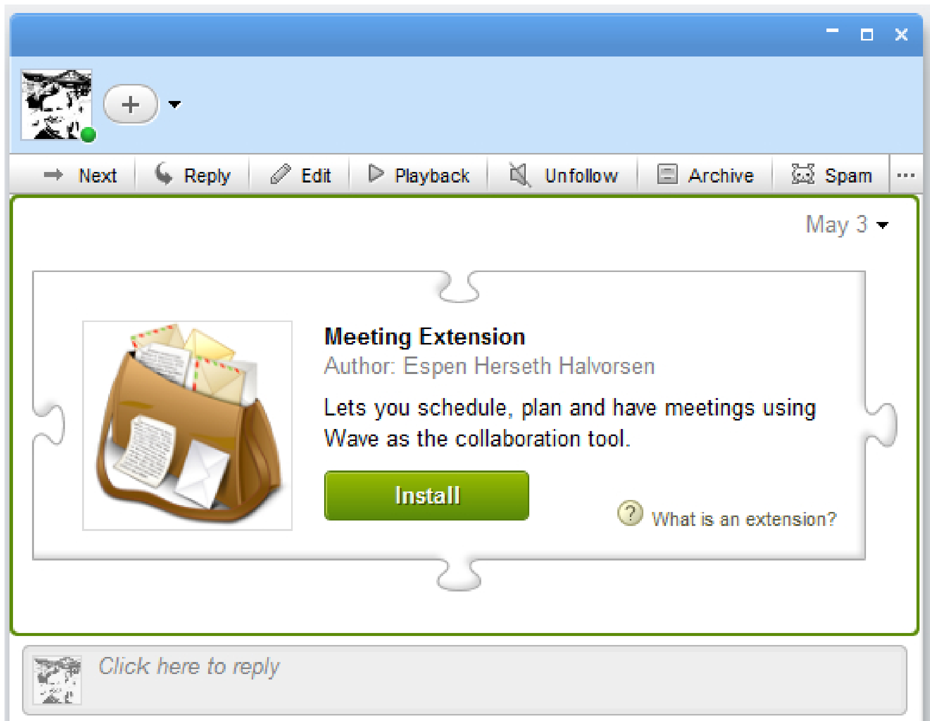

Figure 4.3: *The extension installer as shown in Google Wave*

Chapter 5

Enhancing communication using Wave Robots

In this chapter we extends the solution we created in Chap. 4 to include the usage of Wave Robots. Since this might have some implications for how users interact with the solution, we begin by looking at the psychological side of having robots as equal participants in Sect. 5.1. Next, we move on to describe how we extend the previously created gadgets functionality in Sect. 5.2. We look at a way to automatically initialize a new wave in Sect. 5.2.1, a way to automatically find the best time to hold the meeting in Sect. 5.2.2 and a way to automatically decide upon a fitting meeting schedule in Sect. 5.2.3. We end this chapter by looking at the final solution, explaining and illustrating each step of the process of planning and holding a meeting by using Wave in Sect. 5.3.

5.1 Robots as equal participants

As we have seen through the example in Chap. 4, one can greatly improve communication and collaboration by using gadgets in Wave. This allowed us to create a comprehensive solution that made the process of setting up meetings and collaboratively decide upon time and topics easier.

Even though this solution works great just using the different gadgets, we can do better. As we have seen from the example compositions described in Sect. 3.4 and 3.5, we can enhance a wave even further by using robots to assist users in tasks that are either repetitive or can be done better by an automated system. There we saw the robots helping users by bringing in additional external data that enriched the communication. Robots also could help in decision making, and assist users in the communication easing the process for all parties.

To be able to fully understand the implication of allowing a robot to interact in a conversation, we have to look at the psychological effects. As we have discussed earlier, one of the main design principles of Wave is that there should be no strict difference between a human user and a robot. They are equal participants in the wave when you look at them from a permission-viewpoint. They also appear similar in the user interface - aside from the name, there are no way to distinguish between a robot and a human user. This means that the users participating in a wave where there is one or more robots actively participating have to be comfortable with this situation.

As we develop robots for Wave that are smarter than robots that just bring in information, for example robots that can look at input from real users and actually make decisions for the group based on this data, the psychological problem becomes more important. Not all users will be comfortable having a computer making decisions for them. Hence we have to be very open about which factors the robot takes into consideration

when making decisions, making the skeptical users able to find the result of the work done by the robot by manual computation, and compare the result to the one generated by the robot. By having a way to show the users the exact mathematical algorithms the robot uses for its composition, we can reduce the likelihood of users feeling a loss of control.

The skepticism against robots will most likely become smaller as these users get used to robots participating in the communication, and hence we can make the robots smarter over time without running into the risk of making the users feel that they lose control.

5.2 Enhancing the previous example using robots

In Chap. 4, we created two gadgets that aids in the cooperative scheduling of meetings. By adding a robot with some extra capabilities to the wave, we can greatly expand their functionalities, making them even simpler to use. The processes that had to be done manually based on just the two gadgets, can now be automated and done by the robot. We will look into a few examples of things the robot can automate for the participants, beginning with the creation of the wave.

5.2.1 Setting up the wave

As we saw in Sect. 4.6, we can make it easier to add gadgets to a wave by installing buttons for each gadget to the toolbar of the Wave client. There is a similar functionality that lets you create new waves where a set of participants gets added automatically.

We want to have a new option in the "new wave"-menu for creating a new meeting, where we automatically add a robot that will aid in the process. To do this, we extend our already existing Extension installer which we created and explained in Sect. 4.6, adding this part inside the extension-tag:

```
1 <menuHook location="newwavemenu" text="New Meeting"
2   iconUrl="http://folk.ntnu.no/espenher/wave/users.png">
3       <createNewWave>
4           <participant id="wave20apitest@appspot.com" />
5       </createNewWave>
6 </menuHook>
```

As we see, the syntax of this is almost the same as we saw in Sect. 4.6. We have a *menuHook*-tag which specifies that we want to extend the menu of the Wave client. We specify the desired text and icon we want to appear. As for now, Google's Wave client doesn't use the icon, but this is an example of an area where the different Wave clients can implement things differently. Hence it is smart to include it anyway, to be future proof regarding new clients that will appear.

Inside the *menuHook*-tag, we specify what should happen when the user selects our new menu item. We want to create a new Wave, hence we specifies the *createNewWave*-tag. Inside this tag, we add a *participant*-tag which specifies the address to out newly created robot. The result of this is that a new wave will be created, containing the user currently logged in as well as our robot as participants.

Automatically adding the gadgets to the wave

Now that we have a robot as a participant in the wave, we can begin to think of different procedures that are tedious and repeating, and hence can be automated. As we explained in Sect. 5.1, robots participate in a wave in the same way as humans, and can perform all the same tasks as humans. This means that we can delegate things we don't want to have to do manually to the robot.

The process of adding the two gadgets to a new wave for planning a meeting is perhaps the most repeated task, it is the first thing you do each time you want to plan a meeting. Hence it is a great first task for

the robot.

As we explained in Sect. 2.5.2, the robot can subscribe to different events that happens in a wave. We want to add the gadgets to the wave as soon as it is created. As we see from Table 2.2, the suitable event to listen to is the *WaveletSelfAdded*-event. We don't have an event for the creation of the wave, but this event is triggered when the robot is added to the wave, and since the robot is added at the creation of the wave, this is actually the first time the robot can interact with the wave.

We use the official Wave Java Robot Library to specify the desired actions when this event happens. Using the library, the only thing we have to worry about is overriding the method corresponding to this event, the framework will set up everything else for us - namely creating a file called *capabilities.xml* which specifies which events the Wave server should notify the robot about.

The code for the method override is the following:

```
1 @Override
2 public void onWaveletSelfAdded(WaveletSelfAddedEvent event) {
3     Blip rootBlip = event.getBlip();
4     rootBlip.append(new Gadget("URL_TO_SCHEDULING_GADGET"));
5     rootBlip.append(new Gadget("URL_TO_AGENDA_GADGET"));
6 }
```

Here, we first find the blip corresponding to the events. (As explained in the terminology section [2.3.1], a blip is the single message inside a wave) Since the context of this event is the main wavelet of the wave, the blip we get will be the root blip, e.g. the first message of the wave.

Next, we use the *append*-function on the blip, adding a *BlipContent*-object to the blip. In this case, these *BlipContent*-objects is Gadget-objects instantiated with the URL of the two gadgets we want to add.

We don't have to do anything more after this, the Wave Java Robot Framework will take care of replicating the changes we did to the blip

when we return from the method.

The result of all this can be seen in Fig. 5.1, where we first show the new item added to the menu, and then the new wave which gets created when a user selects this menu item. The gadgets will be added as soon as the response from the robot returns, which in most cases happens within a few seconds.

5.2.2 Automatically decide a suitable time

In the previous chapter, we created a gadget that lets each participant vote on different timeslots for the meeting by indicating whether they can participate at that timeslot or not. This provides us with a good solution for simply seeing which timeslots are the most popular ones, but there are still some manual work left that could be automated by a robot.

One of the first things that comes to mind when thinking about meetings is that the different participants might be of various importance. Some of the people invited to the meeting might be crucial for the success of the meeting, in such a way that a meeting without them cannot be held. Other persons might just be ”nice to have” spectators, their attendance is not required but it might still be nice to have them attend. And others again might just be regular participants, in which one should try to find a time where they can attend since they might have valuable input in the meeting, but the meeting can still be held even if a few of these people are unable to attend.

With all this extra data in play, it becomes a bit harder to see which timeslot is the most suited. Even though a timeslot have a lot of participants who can attend, it doesn't matter if all of these are in the unimportant category and the important participants can't join. Hence we need some sort of automated procedure which can help us pick between the timeslots.

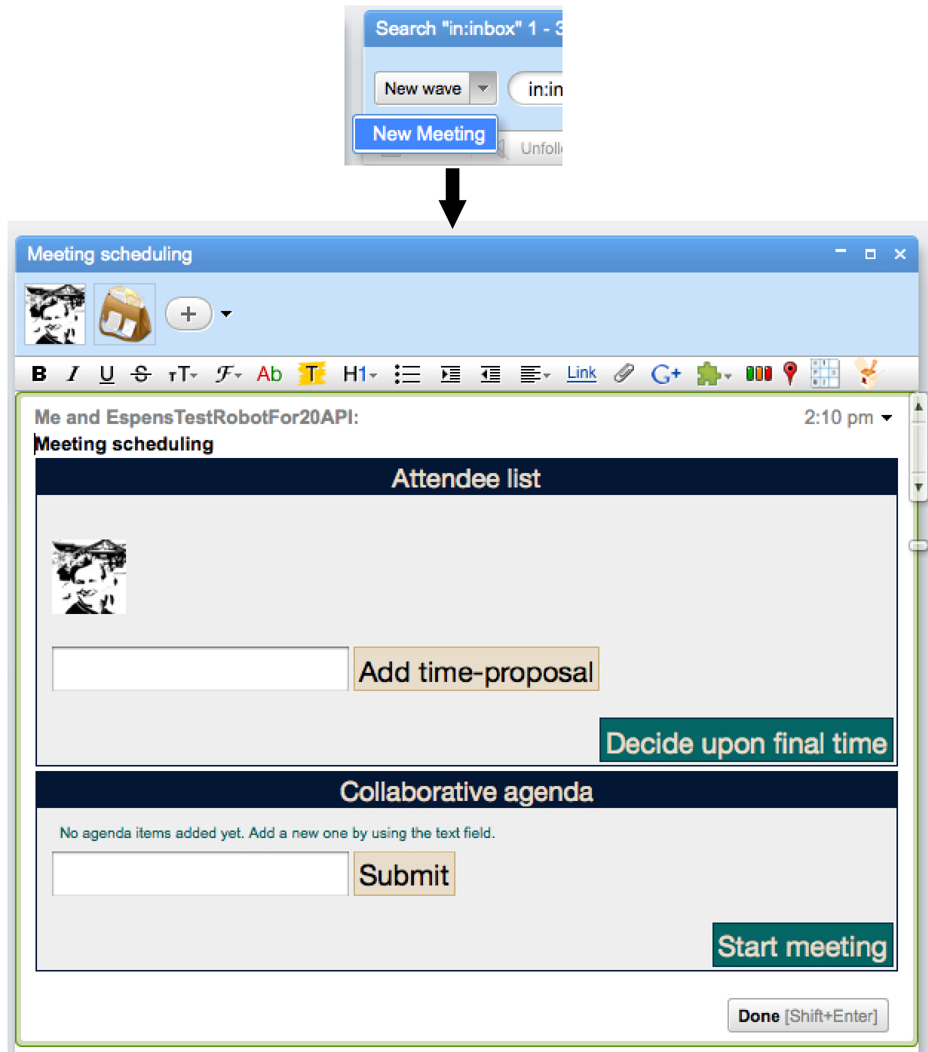

Figure 5.1: *Top: the new menu item created by the extension installer. Bottom: the resulting wave containing the two gadgets added by the robot.*

Assigning priorities to the different participants

To be able to automate this process, we first need a way to assign the priority of the different participants. How many different priorities one should have is a matter of discussion, for the gadget we are developing we will go with three different classes of users:

1. **Normal priority:** This is the default priority assigned to all new participants.
2. **Low priority:** This is the participants who are just ”nice to have” attending the meeting.
3. **High priority:** This is the participants which are crucial for the meeting.

Now that we have decided upon these three classes, we need a way to represent it in the graphical user interface, and also a way to change the priority. We developed a way to show and assign attendee status in Sect. 4.4 which also have three different statuses: unknown, attending and not attending. The changing of status was done by a simple click on the block showing the status.

We can easily reuse that principle in this case. As seen in Fig. 4.1, each participant is shown using their profile picture. Since there is a convention to click on the squares to change the status, we can use the same principle here: click on the profile picture to change the priority of the user. This is however not immediately recognizable as a feature, hence we have to explain it somewhere. We explained the feature to change the attendee status using text on the boxes for the current user, but this can’t directly be transferred to the profile picture since we don’t want to put text over a picture. Hence we instead explain it textually beneath the function.

We also need a way to display the assigned priority. We will use the same principle as the other boxes: namely using colors. But now we only

use a small stroke, since we want the profile pictures to be the main item. We assign a gray color to normal priority participants, a yellow color to low priority participants, and a green color to high priority participants.

The code for changing and showing the priorities is JavaScript code, since this happens only in the gadget - we haven't touched the robot yet. We use the same principle as earlier, with the response to an interaction with the GUI causing a new variable to be stored in the underlying data structure of the gadget. The following method will change the priority of a user, and is called when a user click on any participants profile picture:

```
1 function changeUserPriority(newUserStatus, userId) {
2     var userPriorities = JSON.parse(wave.getState().get('
          userPriorities'));
3     if(userPriorities == null) {
4         userPriorities = new Object();
5     }
6     userPriorities[userId] = newUserStatus;
7     wave.getState().submitValue('userPriorities', JSON.stringify
          (userPriorities));
8 }
```

The structure of this code is fairly similar to the previously shown code snippets for updating the underlying data structure. First, we fetch the existing priorities-object. If this is not already set, we create an empty object. Next, we assign the priority value in this object for the user which profile picture was clicked. Then, we store the object back into the underlying data structure.

Next, we have to alter the code which shows the profile pictures of each user to add the color stroke around the pictures based on the assigned priority. The following code is used to decide upon the color of the picture:

```
1 var userPriorities = JSON.parse(wave.getState().get('
      userPriorities'));
2 if( userPriorities == null ||
3     userPriorities[participants[i].getId()] == null ||
```

```
4       userPriorities[participants[i].getId()] == 0) {
5         // Output picture with gray stroke
6   } else if (userPriorities[participants[i].getId()] == 1) {
7         // Output picture with green stroke
8   } else if (userPriorities[participants[i].getId()] == 2) {
9         // Output picture with yellow stroke
10  }
```

This code is fairly self explanatory. The only tricky part is the first if statement, since we have to take into consideration the case where no user priorities have been set, and the case where the user priority for the user in question hasn't been set. In both these cases, we assume that the priority of the uses is "normal".

The algorithm for ranking timeslots

Before we create the robot that will automatically rank the different timeslots, we have to develop an algorithm that it will use. Based on the earlier work, we see that we can use the different priorities assigned to people to rank the different timeslots based on the attendee responses of the participants.

To be able to easily rank the different timeslots, we will simply calculate a numerical value for each timeslot and show this to the user. Then the user can make an informed decision about the timeslots. In most cases there will be one clear winning timeslot, but since we want the user to feel that he or she remains in control, we will leave the ultimate choice to the user.

The pseudo code for the algorithm will be the following:

```
1 loop through all the timeslots:
2     loop through all the participants
3         add or subtract to the timeslot's score based on priority
              and attendee-status of user.
```

User priority	Color representation	If user are attending	If user are not attending
Normal	Gray	+3 points	-1 points
High	Green	+5 points	-3 points
Low	Yellow	+1 points	0 points

Table 5.1: *The default values for the adding or substraction of points for the timeslot based on user priorities.*

We need to have some default values for the algorithm. The default values chosen are summarized in Table 5.1.

Using a robot to implement the algorithm

Now that we have a ranking system of the participants in place, and a algorithm for ranking the different timeslots based both user priorities and their attendee status on the different timeslots, we can begin implementing the code for the robot that should do all the calculation.

First, we need a way to be able to tell the robot to start the calculation. Ideally, we would have developed the solution in a way that allowed us to show the rankings at all times, and also let the robot update the rankings each time a user changes a priority or attendee status. But due to a bug in the current implementation of the Google Wave Robot API (Issue 679 in the Google Wave API Bug Tracker [17]), we have no way to communicate a result directly back into the gadget. Instead, we chose to implement the solution in a way that lets the user push a button to calculate the result. The Robot will communicate the result to the user through a textual response beneath the gadget.

We create a button in the GUI of the gadget, and when this is pushed, we store a value in the underlying data structure which the robot can then query to see if it should calculate the result. We store the value using this code:

```
function scheduleMeeting() {
    wave.getState().submitValue("decideTime", "true");
}
```

This makes it possible for the robot to query the *decideTime*-property, and if it is set to *true*, it can start the calculation.

The robot is coded in Java using the Wave Robot Java Client Library, which as explained in Sect. 2.5.2 lets us respond to changes in a wave by simply overwriting methods. As we can see from Table 2.2, the event we want to respond to is the *GadgetStateChanged*-event. This event will be triggered each time a gadget's underlying data structure is changed. Since we are just interested in the scheduling gadget, we need to filter on this. The following code will filter out the scheduling gadget, and then see if the property *decideTime* is set to true.

```
@Override
public void onGadgetStateChanged(GadgetStateChangedEvent event)
    {
    Blip mainBlip = event.getBlip();
    Gadget gadget = Gadget.class.cast(mainBlip.at(event.getIndex
        ()).value());
    if(gadget.getUrl().startsWith(URL_TO_SCHEDULING_GADGET)) {
        if(gadget.getProperty("decideTime").startsWith("true")) {
            //The code for the algorithm goes here.
        }
    }
}
```

Here, we first fetch the blip containing the gadget, and then finds the gadget that received a change in that blip and casts it to a Gadget-object. We check if the URL of the gadget is matching the URL of the scheduling gadget, and then uses the function *getProperty(String propertyName)* to get the *decideTime*-property to check if it is set to true.

Next, we need to fetch the data from the underlaying data structure

of the Gadget to be able to have some data for the algorithm. We use the same method for this. The following code will fetch all the data we need:

```
1 JSONObject userStatuses = new JSONObject(gadget.getProperty("
      userStatus", "{}"));
2 JSONArray timeSlots = new JSONArray(gadget.getProperty("
      timeslots", "[]"));
3 Participants participants=event.getWavelet().getParticipants();
```

As we explained in Sect. 4.4.1, we needed to serialize our data structures using JSON. Since Java doesn't have a native JSON parser, we need to use a third party library to de-serialize our JSON strings. We have chosen an open source library provided by json.org [21], which lets us create JSON objects and JSON arrays from serialized strings, and later on query these objects to get the desired data from our data structures. This library's de-serializing functions works by simply inputting the serialized strings into the JSONObject and JSONArray constructors, based on what our serialized string is. We also need the participants of the wave, and gets this by using the *getParticipants()* function on the wavelet.

Now that we have all this, we can loop through each timeslot, and then loop through all the participants, look at their attendee status for that timeslot and add to or subtract to the score based on the values in Table 5.1. The code is structured according to the pseudo code developed in Sect. 5.2.2, but is fairly complicated due to all the different checks one needs to put in place since both the attendee status and priority of users can be unknown if the users have yet to make a choice in the GUI.

```
1 for (int i = 0; i < timeSlots.length(); i++) {
2     String timeSlot = timeSlots.getString(i);
3     Participants participants = event.getWavelet().
          getParticipants();
4     for (Iterator iterator = participants.iterator(); iterator.
          hasNext();) {
5         String userID = (String) iterator.next();
```

```
 6          String userParticipatingStatusJSON = gadget.getProperty(
                userID);
 7          if(userParticipatingStatusJSON != null) {
 8              JSONObject userParticipatingStatus = new JSONObject(
                    userParticipatingStatusJSON);
 9              if(userParticipatingStatus.has(timeSlot) &&
                    userParticipatingStatus.getInt(timeSlot)==1) {
10                  if(!userStatuses.has(userID)) {
11                      timeslotScores[i] += 3;
12                  } else if (userStatuses.getInt(userID) == 0) {
13                      timeslotScores[i] += 3;
14                  } else if (userStatuses.getInt(userID) == 1) {
15                      timeslotScores[i] += 5;
16                  } else if (userStatuses.getInt(userID) == 2) {
17                      timeslotScores[i] += 1;
18                  }
19              } else if (userParticipatingStatus.has(timeSlot) &&
                    userParticipatingStatus.getInt(timeSlot)==0) {
20                  if(!userStatuses.has(userID)) {
21                      //No alteration to the score
22                  } else if (userStatuses.getInt(userID) == 0) {
23                      timeslotScores[i] -= 1;
24                  } else if (userStatuses.getInt(userID) == 1) {
25                      timeslotScores[i] -= 3;
26                  } else if (userStatuses.getInt(userID) == 2) {
27                      //No alteration to the score
28                  }
29              }
30          }
31      }
32  }
```

This code is provided without further explanation, since it shows just the actual Java implementation of the pseudo code provided in Sect. 5.2.2. The two *for*-loops refer to the for-loops in the pseudo code, and everything inside these implements the actual logic for assigning scores according to

the values in Table 5.1.

Lastly, we need to transfer the results back to the wave. For now, we will simply put it in textual form. When the bug that hinders us from communication the result back to the gadget is fixed, some future work could be to incorporate this directly into the gadget and have it update in real-time as participants interact with the gadget.

The code we use for appending the result to the end of the blip is the following:

```
1 for (int i = 0; i < timeslotScores.length; i++) {
2    mainBlip.append( "Score for timeslot starting " + timeSlots.
         getString(i) + " : " + timeslotScores[i] + " points\n" )
         ;
3 }
```

The final gadget and robot result

Based on all the changes described in this section, the gadget looks a bit different. The final result can be seen in Fig. 5.2. Beneath the gadget, the results given by the robot can be seen in textual form.

5.2.3 Automatically decide a possible schedule based on user input

In Chap. 4, we also created a gadget that enables users to collaboratively create an agenda by adding agenda items to a list. We also added the possibility to delete agenda items. But apart from this, the gadget was fairly simple, and didn't offer any features to let the participants easily filter the agenda items and decide upon which of them should be included in the actual meeting.

We want to include a robot in the back-end to help the participants intelligently choose between the proposed agenda items. To do this, we

Meeting scheduling

Next Reply Edit Playback Unfollow Archive Spam Read

Me, EspensTestRobotFor20API and Mats K: 4:26 pm

Meeting scheduling

Attendee list

	12.00	13.00	14.00	15.00	16.00
	change	change	change	change	change
	1	2	0	0	1

Add time-proposal

Click on the participant-pictures to change their importance.
Gray users are normal participants.
Green users HAVE to attend at the meeting.
Yellow users are just "nice to have" spectators.

Decide upon final time

Scores for the different timeslots:

Based on which participants is most important and when the different participants can attend, the robot have given a score to each time slot. You are of course free to pick anyone you like, but based on the algorithm the timeslot with the highest score is the best.

- **Score for timeslot starting 12.00** : -2 points
- **Score for timeslot starting 13.00** : 6 points
- **Score for timeslot starting 14.00** : -3 points
- **Score for timeslot starting 15.00** : 0 points
- **Score for timeslot starting 16.00** : 5 points

Figure 5.2: *The Gadget for time scheduling, with the scores for the different time slots calculated by the robot shown beneath.*

need a way to let the participants rate the agenda items in a meaningful way. By simply letting the participants vote on the different agenda items, we could easily implement an algorithm in the robot to rate and sort the agenda items based on popularity.

To automatically select the agenda items that fit into the meeting, we also need a way to know how long each agenda item will take to discuss, and also the total duration of the meeting. Hence we need to have an extra field to let the users input the assumed time each agenda item will take to discuss. In addition, we need a way to specify how long the meeting will be.

Based on all this, we need to make some changes to the Gadget before we can develop the Robot.

Adding time and rating features to the gadget

The first thing we have to do is to add a field to let the participants specify how long they think each agenda item they proposes will take during the meeting. We add this field directly beside the field for inputting the description of the agenda item.

After the user pushes the button to add the agenda item, we execute the previously explained function to store the agenda item in the underlying data structure of the gadget. We add the following line to this function to also store the assumed time:

```
1 newAgendaItem.agendaItemDuration = form.duration.value;
```

We also update the code generating the list of proposed agenda items to include this new information behind the description of the agenda item. This lets the participants take a more informed choice regarding the voting on the different agenda items.

Next, we have to create a way for users to be able to vote on the different agenda items. We do this by adding a check mark in front of

each agenda item in the list. When the user clicks on this check mark, we call a function that will store the fact that the user have voted on this item in the underlying data structure. The code for this function is the following:

```
1 function voteOnAgendaItem(state, agendaItemID) {
2     var thisUsersVoteKey = 'votesBy' + wave.getViewer().getId();
3     var votes = JSON.parse(wave.getState().get(thisUsersVoteKey)
          );
4     if(votes == null) {
5         votes = new Array();
6     }
7     votes[agendaItemID] = state;
8     wave.getState().submitValue(thisUsersVoteKey, JSON.stringify
          (votes));
9 }
```

As we see from this code, we keep a separate array for each participant's votes. We need to do this to be able to show each user which items they have voted on in the GUI, and to let them undo the vote at a later time. By keeping each single vote in the underlying data structure, instead of just a count of the total number of votes for each agenda item, we also have more control over the consistency of the data. And lastly, we gets all the concurrency features offered by Wave since each user only is able to change the value of one key in the underlying state storage system.

The rest of this code is based around the same principles as all the other code we have developed to respond to user input in the GUI. First, we fetch the existing data from the underlying data structure. If it doesn't exist, we create an empty array. We alter the data adding the new information, and then submit it back into the underlying data structure.

The updating of the GUI happens when the altered data structure returns from the server. Here, we need to check if the user viewing the gadget has voted on any of the agenda items. If he or she has, we mark

these items with a green color to indicate that they have voted on them. We also alter the way the *voteOnAgendaItem()*-function is called, making it cancel the vote if executed on an agenda item that is voted on.

A robot that selects agenda items

Now that we have some data to use for deciding between the different proposed agenda items, we can implement a robot that does this automatically. To start this process, we use the same principle as in Sect. 5.2.2. We create a button to start the selection of agenda items, which calls the following method that sets a variable the robot later can query.

```
1 function startMeeting() {
2   var meetingLength = prompt('Please specify the length of the
        meeting.')
3   if (meetingLength != null) {
4     wave.getState().submitValue("startMeeting", meetingLength)
          ;
5   }
6 }
```

Note that we here also prompt the user and ask about the desired length of the meeting. This will be used by the robot to select the correct number of agenda items.

Now we can extend the method in the Wave Java Robot Client Library that responds to the *GadgetStateChanged*-event, and then filter out events that happens on the agenda gadget. We do this using the following code, which is fairly similar to the one provided in Sect. 5.2.2.

```
1 @Override
2 public void onGadgetStateChanged(GadgetStateChangedEvent event)
      {
3   Blip mainBlip = event.getBlip();
4   Gadget gadget = Gadget.class.cast(mainBlip.at(event.getIndex
        ()).value());
```

```
5       if(gadget.getUrl().startsWith(URL_TO_AGENDA_GADGET)) {
6           String meetingLength = gadget.getProperty("startMeeting")
                ;
7           if(meetingLength != null) {
8               //The code for selecting items goes here
9           }
10      }
11  }
```

The check here is a bit different, since we doesn't check if the property startMeeting is set to *true*, instead we just checks if it is set at all, and if it is, the value is the meeting length specified by the user who pushed the button to select agenda items.

Inside this code, we will do the calculation of the score for each agenda item. We will simply count a vote from any user as a single point in the score. One could alternatively develop a more sophisticated algorithm, taking into account the different priorities of the participants, and whether they are attending the meeting at the selected meeting time or not. For now, this functionality is left out to keep the two gadgets fairly separate regarding shared code, but as a future work this seems like a nice optional feature for the gadget.

Since we have kept a separate data structure for each userŠs votes, we now need to combine them all to count the total votes for each agenda item. We do this using Java code matching the following pseudo code:

```
1 Create object to hold the vote count for each agenda item
2 Loop through all the participants
3     Loop through participant's votes
4         Add one vote to the agenda item the user voted for
```

To implement this pseudo code, we first create a new Class that can hold an agenda item and its vote count using this code:

```
1  class Items implements Comparable<Items> {
2      int id;
3      int count;
4      public Items(int id, int count){
5          this.id=id;
6          this.count=count;
7      }
8      @Override
9      public int compareTo(Items other){
10         if(this.count == other.count) {
11             return 0;
12         } else if (this.count < other.count) {
13             return -1;
14         } else {
15             return 1;
16         }
17     }
18 }
```

Note that this Class implements the Comparable interface, meaning that we later on can use the *Collections*-package in the Java library to sort and reverse sort the list we creates containing the instances of these objects. We create this List using the following code:

```
1  JSONArray agendaItems = new JSONArray(gadget.getProperty("
       agenda", "[]"));
2  ArrayList<Items> agendaItemsVotes = new ArrayList<Items>();
3  for (int i = 0; i < agendaItems.length(); i++) {
4      agendaItemsVotes.add(new Items(i,0));
5  }
```

Here we first fetch all the agenda items from the gadget. We then creates an empty ArrayList, then loops through the agenda items and adds one instance of the object we created and sets the id to match the id of the agenda item. The count we obviously initializes to 0.

The next phase is to loop through all the users, fetch all their votes, and increase the total vote count for each agenda item accordingly. The following code implements the pseudo code we specified earlier:

```
1 Participants participants=event.getWavelet().getParticipants();
2 for (Iterator iterator = participants.iterator(); iterator.
      hasNext();) {
3     String participantID = (String) iterator.next();
4     JSONArray userVotes = new JSONArray(gadget.getProperty("
          votesBy" + participantID, "[]"));
5     for (int i = 0; i < agendaItems.length(); i++) {
6         if(userVotes.optBoolean(i)) {
7             agendaItemsVotes.get(i).count++;
8         }
9     }
10 }
```

This code is fairly self explained, as it uses the same principles as the code explained in Sect. 5.2.2. Note that according to the JavaScript code of the gadget, we used the key *votesBy+PARTICIPANTID* for the voting tables, hence we use the same here to fetch the value of the property. We specify an empty Array as the default value, since this property can be null if the user hasn't voted on any agenda items. This lets us use the method *optBoolean(index)* to fetch the vote, as this method returns false for all items other than "true", even though the array value is unset. This spares us from writing a lot of checks to see if variables are assigned or not.

Lastly, we need to sort the *ArrayList* containing the count of all the votes. Since we implemented the *Comparable* interface on the Class we stored the votes in, we can use the *Collections* class in Java to do this. The following code will first sort the ArrayList, and then reverse it. This gives us a list of all the agenda items, sorted reversely on the number of votes, which means that the agenda items with the most votes comes first in the list.

```
1 Collections.sort(agendaItemsVotes);
2 Collections.reverse(agendaItemsVotes);
```

Creating a template for meeting notes

We need a way to communicate back to the participants which agenda items were selected. As we discussed in Chap. 4, we also need to provide a way for the participants to take notes during the meeting. These two things can easily be combined, meaning that the template for the meeting notes will provide the participants with the necessary information regarding the priority of the different agenda items.

Since we have an ArrayList of all the agenda items sorted by votes, it is fairly easy to generate this template. Wave already provides a great solution for collaboratively editing text; hence we don't need to create a new gadget for taking notes. Instead, we just uses the agenda description, number of votes and the time as a heading, and provides a pre-made list beneath each item for the participants to take notes collaboratively.

```
1  int meetingTimeRemaining = Integer.parseInt(meetingLength);
2  for (int i = 0; i < agendaItemsVotes.size(); i++) {
3      meetingTimeRemaining = meetingTimeRemaining - agendaItems.
           getJSONObject(agendaItemsVotes.get(i).id).getInt("
           agendaItemDuration");
4      String optional = "";
5      if(meetingTimeRemaining<0) {
6          optional = "(optional) ";
7      }
8      mainBlip.append(heading2Style);
9      mainBlip.append(optional + agendaItems.getJSONObject(
           agendaItemsVotes.get(i).id).getString("agendaItemName")
           + " (" + agendaItems.getJSONObject(agendaItemsVotes.get(
           i).id).getString("agendaItemDuration") + " min)" + " ["
           + agendaItemsVotes.get(i).count + " votes]");
10     mainBlip.append(bulletedStyle);
```

```
11     mainBlip.append("[insert your notes here]");
12 }
```

Here, we have also included a counter for the meeting time. We subtract the time taken for each item we picks, and when we have gone over the designated time for the meeting, we add the prefix "(optional)" in front of the agenda items. This allows the participants to be more flexible, allowing them to incorporate more agenda items into the meeting if they have misrepresented the time for the first agenda items.

The final gadget, with the generated agenda complete with votes can be seen in Fig. 5.3.

Figure 5.3: *The agenda gadget, with voting and time specification. Beneath the gadget is the note taking template generated by the robot.*

5.3 The finished solution

Now that we have used a robot to extend the functionality of our solution, the process of arranging a meeting has been greatly simplified. The process is as follows, and each step is illustrated in Fig. 5.4 showing the upper part of the wave, and Fig. 5.5 showing the bottom part of the wave.

1. Select ”new meeting” from the ”new wave”-menu. The robot is added as a participant automatically, and then the robot adds the two necessary gadgets.

2. Add the desired meeting participants as receivers of this wave.

3. Propose a few timeslots, or let the other participants do this. Wait until the participants have indicated their attendee-status for the different timeslots.

4. At the same time, collaboratively propose agenda items, and vote on the proposals.

5. When you want to decide upon a time, push the button in the scheduling gadget, and let the robot rank the timeslots. Pick the one you want, based on the information you are given.

6. When you are ready to start the meeting, push the button in the agenda gadget, specify the meeting-length, and a template for the meeting notes with the selected agenda items are created by the robot.

7. Collaboratively take notes during the meeting, and have this automatically saved for later reference.

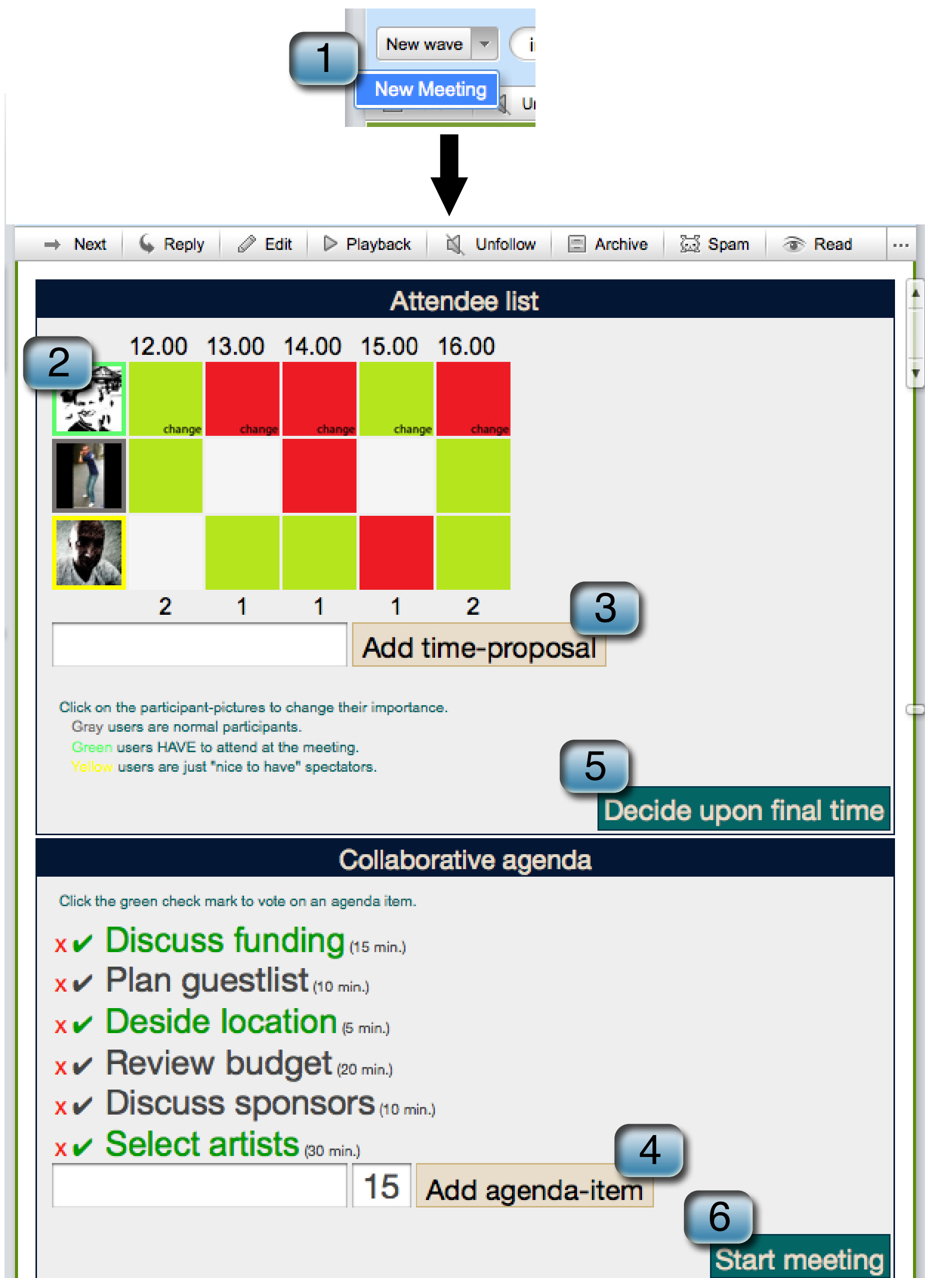

Figure 5.4: *A wave showing the finished gadgets and results of the robots calculations, upper part*

15 Add agenda-item

Start meeting

Scores for the different timeslots: 5

Based on which participants is most important and when the diffe[...]cipants can attend, the robot have given a score to each time slot. You are of course free to pick anyone you like, but based on the algorithm the timeslot with the highest score is the best.

Score for timeslot starting 12.00 : 8 points
Score for timeslot starting 13.00 : -2 points
Score for timeslot starting 14.00 : -3 points
Score for timeslot starting 15.00 : 5 points
Score for timeslot starting 16.00 : 1 points

Meeting notes: 6

Below you can take notes duri[...]eeting. Please do this collaboratively, you can see each other write in real time and also see who is writing what part both in real time and by using the playback functionality to replace the events of the meeting after you have finished. Good luck :)

Select artists (30 min) [3 votes]

- We want high profile artists!

Deside location (5 min) [3 votes]

- How about the town hall?
- Other cheaper alternatives?
- We should Mats K 7

Discuss funding [...]in) [2 votes]

- [insert your notes here]

Discuss sponsors (10 min) [1 votes]

- [insert your notes here]

(optional) Review budget (20 min) [1 votes]

- [insert your notes here]

(optional) Plan guestlist (10 min) [0 votes]

- [insert your notes here]

Done [Shift+Enter]

Figure 5.5: *A wave showing the finished gadgets and results of the robots calculations, bottom part*

Chapter 6

Conclusion and Future work

Through this thesis, we have explored Wave, a new communication and collaboration platform developed primarily by Google and released under an open source license. By analyzing Wave in the perspective of end user service composition, we have identified different use cases where such a highly cooperative platform can help users perform everyday tasks in a more efficient manner.

We have also looked into the different ways one can extend the functionality of Wave, and explained the details of the frameworks that let developers create robots and gadgets which can inter-operate with Wave. The possibility of reusing existing components created by other developers to perform service composition has been thoroughly explored, and two different example compositions have been explained.

The last part of this thesis focuses on creating examples of Wave extensions that aims to solve the problem of organizing meetings. We have created two different gadgets which solves respectively the problem of scheduling a meeting and the problem of collaboratively creating a meeting schedule.

In addition, we have also developed a robot designed to automate some of the more complex tasks which is suited to solve algorithmically. This involves automatically picking suitable time slots for the meeting based on the input of the participants, and to pick agenda items based on the time limit of the meeting and the votes on the different items and utilize these to create a template which can be used for collaborative note-taking during the meeting.

6.1 Future work

Based on the results of this thesis, two different areas worth further examination is proposed. Each of these areas could be the basis of a future work.

6.1.1 Trust and security in Wave

Due to the highly cooperative nature of Wave, there are a lot of issues related to trust and security that should be looked into. The work done in this thesis has been limited regarding this area, but this doesn't mean that it isn't important.

Google has put a lot of thought into the he specification of the Wave framework regarding security and trust[22, 32], hence one gets a lot "for free" just by using the frameworks provided. A future work could consist of looking into all the detailed specification of access control in Wave, examining whether the solutions given in the specification provides enough security for such a platform. Additional security measures, one example being obtaining a higher level of trust by storing encryption keys of users in their private data store provided by gadgets, should also be proposed and detailed.

6.1.2 Combining Wave and Arctis

Arctis is a tool for creating reactive systems, developed at the Department of Telematics at NTNU [23]. The tool is based around the principle that small building blocks can be created and shared among developers, which in turn can combine them to form larger blocks. By repeating this process, one will in the end have a complete system consisting of different reusable parts.

Throughout this thesis, we have explained how Wave can be used as a system for end user service composition. As we can see, these two ideas are in principal quite similar. Hence one can imagine that it could be possible to integrate the Arctis framework with Wave, hereby making it possible to develop extensions for Wave using the Arctis toolset.

Based on these ideas, a future work could consist of looking into whether these two technologies can be combined, and doing some measurements to see if combining them actually makes the development process easier. If the result is that it is worth doing this fusion, a code generator for the Arctis framework could be developed, that would translate Arctis building blocks into Java code for Wave Robots, and JavaScript, HTML and CSS for Wave Gadgets.

Bibliography

[1] R.M. Baecker. Readings in human-computer interaction: toward the year 2000, 1995.

[2] Anthony Baxter, Jochen Bekmann, Daniel Berlin, Soren Lassen, and Sam Thorogood. Google wave federation protocol over xmpp, 2009.

[3] Jochen Bekmann. Updates from google wave federation day, 2009.

[4] Jochen Bekmann, Michael Lancaster, Soren Lassen, and David Wang. Google wave data model and client-server protocol, 2009.

[5] Jochen Bekmann and Sam Thorogood. Google wave federation protocol and open source updates, 2009.

[6] Google Docs Blog. A rebuilt, more real time google documents, 2010.

[7] Uwe M. Borghoff and Johann H. Schlichter. Computer supported cooperative work: Introduction to distributed applications, 2000.

[8] Douglas Crockford et al. Json specification, request for comments: 4627, internet engineering task force, 2006.

[9] Doodle. The doodle documentation, 2010.

[10] The Apache Foundation. Apache license, version 2.0, 2004.

[11] XMPP Standards Foundation. The extensible messaging and presence protocol, 1999.

[12] Adam Pash Gina Trapani. The complete guide to google wave, 2010.

[13] Google. Wave extensions api overview, 2009.

[14] Google. Wave gadgets api, 2009.

[15] Google. Wave robot wire protocol, 2009.

[16] Google. Wave robots api, 2009.

[17] Google. google-wave-resources - bug tracker for the google wave apis, 2010.

[18] J. Grudin. Why cscw applications fail: problems in the design and evaluation of organization of organizational interfaces, 1988.

[19] J. Grudin. Computer-supported cooperative work: history and focus. *Computer*, 27(5):19 –26, may 1994.

[20] Stephanie Hannon. The google wave blog: Google wave available for everyone, 2010.

[21] json.org. Json java library, 2010.

[22] Lea Kissner and Ben Laurie. General verifiable federation, 2009.

[23] Frank Alexander Kraemer. Arctis and Ramses: Tool Suites for Rapid Service Engineering. In *Proceedings of NIK 2007 (Norsk informatikkonferanse), Oslo, Norway.* Tapir Akademisk Forlag, November 2007.

[24] Google Labs. The google wave extension samples gallery, 2010.

[25] Michael Lancaster. Google wave attachments, 2009.

[26] Soren Lassen and Sam Thorogood. Google wave federation architecture whitepaper, 2009.

[27] Lotus. Lotus domino and notes documentation, 2010.

[28] Microsoft. Microsoft exchange and outlook documentation, 2010.

[29] Douwe Osinga. Introducing the google wave apis: what can you build?, 2009.

[30] Lars Rasmussen. Went walkabout. brought back google wave. (introduction of google wave), 2009.

[31] Linda Rising and AG Communication Systems Norman S. Janoff. The scrum software development process for small teams, 2000.

[32] Jon Tirsen. Access control in google wave, 2009.

[33] David Wang and Alex Mah. Google wave operational transformation, 2009.

[34] P. Wilson. Computer supported cooperative work: An introduction, 1991.