A System for Disaster Response Process Management

MUHAMMAD RIZWAN SAEED

Master of Science Thesis
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:7
A System for Disaster Response Process Management

13-Jan-2012
KTH – The Royal Institute of Technology, Sweden
Muhammad Rizwan Saeed
mrsaeed@kth.se

Examiner: Vladimir Vlassov (vladv@kth.se)

Industry Supervisor: Joern Franke (joern.franke@sap.com)
Abstract

During a disaster, public safety organizations face a very dynamic and continuously changing situation with unforeseen challenges and unexpected events. The main problem in disaster response management lies at the coordination and collaboration of activities of different organizations involved, both at the inter- and intra-organization level. Current practices and process management approaches have several limitations and are not suitable for managing disaster response processes. Our developed system focuses at this issue and provides support for real time collaboration of activities and their dependencies among the participant organizations involved in disaster response management. The system is implemented as an extension to Google Wave collaboration infrastructure.

Note: This work was done at SAP Labs France within the Public Security research team. The development work presented in the document is based on the research done by Joern Franke.
# Table of Contents

## CHAPTER 1 ............................................................................................................................................................. 1

### BACKGROUND & SCOPE ........................................................................................................................................ 1

1.1 DISASTER MANAGEMENT LIFECYCLE .................................................................................................................. 2
1.2 MOTIVATION .......................................................................................................................................................... 3
1.3 CURRENT PRACTICES FOR DISASTER PROCESS MANAGEMENT ........................................................................ 3
    1.3.1 Drawbacks of Current Practices .................................................................................................................. 6
    1.3.2 Inflexibility of the Current Process Management Approaches .................................................................... 6
        1.3.2.1 Process Modeling ........................................................................................................................................ 6
        1.3.2.2 Process Execution ........................................................................................................................................ 7
        1.3.2.3 Inter-Organizational Coordination/Collaboration ....................................................................................... 7
        1.3.2.4 Monitoring .................................................................................................................................................. 8
1.4 PERSONAL MOTIVATION ....................................................................................................................................... 8
1.5 CHAPTER SUMMARY ........................................................................................................................................... 9

## CHAPTER 2 .......................................................................................................................................................... 10

### INTRODUCTION ............................................................................................................................................... 10

2.1 SYSTEM REQUIREMENTS ...................................................................................................................................... 10
2.2 TECHNOLOGY REQUIREMENTS ........................................................................................................................... 11
2.3 BASIC DISASTER SCENARIO ................................................................................................................................. 13
2.4 PRIMER OF GOOGLE WAVE .................................................................................................................................. 15
    2.4.1 Wave .............................................................................................................................................................. 15
    2.4.2 Wavelet .......................................................................................................................................................... 15
    2.4.3 Document ...................................................................................................................................................... 15
    2.4.4 Wave Extensions ........................................................................................................................................... 16
        2.4.4.1 Gadgets ..................................................................................................................................................... 16
        2.4.4.2 Robots .................................................................................................................................................... 17
2.5 USAGE OF WAVE EXTENSION MECHANISM IN THE CONTEXT OF SYSTEM ................................................. 18
2.6 WAVE SERVICE ARCHITECTURE .......................................................................................................................... 18
2.7 CHAPTER SUMMARY ........................................................................................................................................... 20

## CHAPTER 3 ......................................................................................................................................................... 21

### PROPOSED SOLUTION ..................................................................................................................................... 21

3.1 MODELING .......................................................................................................................................................... 21
    3.1.1 Activity Type .................................................................................................................................................. 22
    3.1.2 Activity .......................................................................................................................................................... 23
    3.1.3 Temporal Dependency ...................................................................................................................................... 23
3.2 VERIFICATION .................................................................................................................................................... 24
3.3 EXECUTION .......................................................................................................................................................... 30
3.4 MONITORING ...................................................................................................................................................... 34
3.5 SHARING OF ACTIVITIES ................................................................................................................................... 35
    3.5.1 Example of Sharing an Activity ...................................................................................................................... 36
3.6 CHAPTER SUMMARY ........................................................................................................................................ 40
CHAPTER 4 IMPLEMENTATION DETAILS

4.1 HOW TO DEVELOP A GOOGLE WAVE GADGET

4.1.1 What is in a Gadget?

4.1.2 Development of Wave Gadget

4.1.2.1 Programming Language

4.1.2.2 Google Web Toolkit (GWT)

4.1.2.3 Eclipse

4.1.2.4 Subclipse

4.1.2.5 Google Plug-in for Eclipse

4.1.2.6 Google App Engine

4.1.2.7 Using the CobogWave API

4.2 DEPLOYING THE GADGET

4.3 USING WAVE EXTENSIONS USING EXTENSION INSTALLER

4.3.1 Creating an Extension Installer

4.4 SYSTEM IMPLEMENTATION IN CONTEXT OF GOOGLE WAVE

4.4.1 Gadget “Activity Dashboard”

4.4.2 Gadget “Activity Specification”

4.4.3 Gadget “Activity Participant View”

4.4.4 Robot “UProMan”

4.5 MAIN SOFTWARE CONCEPTS / PROGRAMMING ABSTRACTIONS

4.6 USE CASE DIAGRAM

4.7 USE CASES

4.7.1 Use Case Id: UC1

4.7.2 Use Case Id: UC2

4.7.3 Use Case Id: UC3

4.7.4 Use Case Id: UC4

4.7.5 Use Case Id: UC5

4.7.6 Use Case Id: UC6

4.7.7 Use Case Id: UC7

4.7.8 Use Case Id: UC8

4.7.9 Use Case Id: UC9

4.8 MAIN SEQUENCE DIAGRAMS

4.8.1 SD1: Update Activity List (Activity Wave Participants Changed)

4.8.2 SD2: Update Activity List (Activity Wave Participants Changed)

4.8.3 SD3: Update Activity List (Model Wave Participants Changed)

4.8.4 SD4: Update Activity List (On Model Creation)

4.8.5 SD5: Update Activity Wave Title (On Re-Naming of Modeled Activities)

4.8.6 SD6: Update Activity Wave Title (On Re-Naming of Un-Modeled Activities)

4.9 TOOLS & TECHNOLOGIES USED

4.9.1 Google Web Toolkit (GWT)

4.9.2 GWT-Connectors

4.9.3 GWT-DND (Drag & Drop)

4.9.4 GWT-Gadget API

4.9.5 CobogWave-Gadget

4.9.6 Google Wave Robots API

4.9.7 Google App Engine SDK
Chapter 1

Background & Scope

The term disaster is generally used synonymously with the terms emergency and catastrophe, but it is important to differentiate between these terms for better understanding. In the light of definitions given by [1], we can distinguish between these terms as follows:

In an emergency, each public safety organization involved knows precisely about its tasks and how to coordinate with other organizations. It is usually a routine for them to cope with this kind of situations and the main focus is to save the lives of the affected people. Examples of emergency may include fire in a house, a traffic accident etc. The effects of the emergency are not widespread and a quick resolve of the situation can be expected.

A disaster, on the other hand is on a significantly bigger scale in which far more organizations are involved as compared to an emergency while it is not always clear what are the dependencies between the activities of different organizations. A disaster gives rise to unexpected challenges for the organizations involved and depending on the evolving situation their goals may change to handle the situation. The real challenge for an organization in a disaster is to coordinate with other organizations and to know who is doing what and how it is related to its own activities. Usually, the responsibility of activities is clear in case of emergency because of clearly defined tasks for each organization but this is not always the case for disasters. Because of highly dynamic situation, planning becomes less important in a disaster, because not everything can be planned in advance as in the case of emergency, and so new plans have to be incorporated with old plans and with the plans of other organizations. Examples of disaster can be a flood or an earthquake etc. We consider the term crisis synonym to disaster and use them interchangeably.

A catastrophe is a widespread disaster affecting the community at a large scale as well as its infrastructure. Communication and coordination becomes nearly impossible due to huge impacts on the people, organizations and infrastructure, in particular the communication infrastructure. Activities of life become paralyzed and it takes a long time to recover the situation. Examples may include nuclear bombs on Hiroshima and Nagasaki during World War II.

The focus of this project is disaster process management but can also be used to reduce and manage the risks in situations where an emergency has the potential to evolve into a disaster.
In order to understand disaster process management, it is important to have an understanding of the disaster management lifecycle, which is described below:

1.1 Disaster Management Lifecycle

Many lifecycles have been proposed for disaster management. A very generic and widely accepted lifecycle (e.g. [3]) for structuring the disaster management process consists of the following four phases:

- **Mitigation (on-going):** This phase deals with preventing or minimizing the risks that a disaster can actually happen.

- **Preparedness (on-going):** In this phase, public safety organizations perform planning and training (Intra and/or inter-organizational level) to deal with disaster response and recovery.

- **Response (usually few hours - few days):** In the response phase, different organizations with dependencies amongst them, execute plans to fight the disaster and its consequences which includes preventing complete social disruption (e.g. starvation, death, diseases etc.). We focus in this project on the processes in the response phase.

- **Recovery (usually few weeks to a few months):** Involves reconstruction of social processes (e.g. building houses, rehabilitation etc.) for the affected communities and bringing the activities of life to normal. Also includes lessons learned and discussion about the response.
1.2 Motivation

During a disaster, many organizations get involved to handle the situation. The central problem in a disaster is the coordination and collaboration of activities of different organizations involved in disaster management. The plans of the organizations can be challenged by the unexpected events and dimensions of the disaster. So the plans have to be adapted according to the circumstances. The activities of many different organizations may have temporal, spatial, resource or other dependencies.

Currently, public safety organizations manage and coordinate their activities during a disaster without or with unsophisticated ICT (Information & Communication Technology) support like phone, e-mail, fax etc. Due to several flaws, practices currently in place have been criticized by all stakeholders, also confirmed during our interviews with public safety organizations.

Recent disasters (Earthquakes, Tsunami, Floods etc) and growing awareness about sophisticated ICT support for disaster management have got much attention in research organizations of different countries as well as in the European Union. Several projects have been granted in this area within the EU FP7-framework but none of them explicitly deals with ICT support for the management and coordination of activities of different organizations involved during disaster response.

1.3 Current Practices for Disaster Process Management

As mentioned above, the main problem during a disaster is the collaboration and coordination of organizations, responsible for managing the disaster. Planning is important but sense making [4] (who is doing what and how it is related to own activities) is more important in a disaster response. In other words, we can say that decision making becomes less important than sense making because only few decisions have to be made.

In order to respond to disasters, public safety organizations establish their command centers to execute a lot of processes by initially relying on existing plans which are written documents containing the list of activities to be performed. But due to the dynamic nature of disasters, new plans have to be made and incorporated with old plans and plans of other organizations. Based on the magnitude of the disaster, one organization might establish more than one command center. There can be one or more field teams controlled by a command center. Planning or modeling is usually done using geographical maps or whiteboards. In order to execute different planned activities, command centers delegate these activities to the responsible people in the field. Field teams provide feedback to the command center about the state of their assigned activities from the field. This feedback information is presented in
different ways by different organizations for visualizing the latest situation on the ground. This may include maps, graphs, charts, matrices or any other means of information visualization. But the important point is that within an organization, usually the same tools or means are used for planning as well as monitoring the current state of activities. This is convenient as well as efficient.

As shown below in Figure 2, currently this is mostly done without the support of technology and so becomes very hard to manage using whiteboards or maps on the paper because there can be cases where information may get lost or the cases where there are information overloads. But during disasters this may have serious consequences because there may be areas with double efforts or other areas with no efforts at all just because the people in the command center did not model the correct view of the situation. The support of ICT, nevertheless, is a desirable feature to manage the disaster response processes effectively and efficiently.
Figure 2: Current Practice for Managing Disaster Response Processes
1.3.1 Drawbacks of Current Practices

The drawbacks that the current practices pose, during a disaster can be summarized as follows:

- It is difficult for decision-makers and people in the field to know what is going on.
- Status about the current situation arrives usually too late.
- There is no big picture how activities are related to each other
- Activities might be “forgotten” (e.g. sandbags are transported but not needed, because area is flooded)
- Double efforts (many different organizations create meals to feed the responders)
- No efforts (nobody transports the meals to the some disaster site)

In short, we need a process management solution for disaster response. In the next section, we argue why the current process management approaches and in particular, business process management cannot be used for managing disaster response processes.

1.3.2 Inflexibility of the Current Process Management Approaches

Among other thing, business process management (BPM) [5, 6], adheres to the same issue but in a different domain. Advanced tools and technologies have also been developed in the field of BPM. So we have chosen it as a reference and based on the criteria; Process Modeling, Execution, Monitoring and Inter-organizational Coordination, we will now compare disaster process management and business process management in the following sub-sections to see how BPM is different from disaster response process management.

1.3.2.1 Process Modeling

Business process modeling languages such as EPC (Event-driven process chain) cannot be used to model parallel activities since they only support sequence of activities while in disaster response processes, many activities have to be run in parallel.

Secondly, different temporal dependencies [7] exist between the activities of a disaster response process while activities in a business process have only data dependencies between them. In other words, business process modeling languages can only model the information flow between activities but we cannot model adequately the temporal relationships, for example, two activities overlap or that activity A contains activity B.
Thirdly, using a business process modeling language, we can only model that who is responsible for executing a certain activity but disaster process management require modeling of other governance roles also associated to activities such as consulted, informed or an accountable person.

This means systems using business process models for describing and managing business processes (e.g. workflow systems) are not very suitable for managing disaster response activities with many exceptional events [9].

1.3.2.2 Process Execution

Execution of business processes is enabled using workflow systems [8], which require business processes to be defined using business process modeling language. But as mentioned in the above section, disaster response processes cannot be modeled using business process modeling languages which means we need a different approach to disaster response process execution.

Secondly, business processes are mostly static in the sense that they rarely change once they are defined. While during a disaster, the situation is very dynamic and response processes need flexibility for change and adaptation.

Thirdly, since the dependencies between activities of business processes and disaster response processes are different (former having data dependencies or information flow and later having temporal dependencies), so the semantics of execution are totally different in disaster response processes and hence need a different treatment.

Executing a disaster response process means detecting the violation of temporal dependencies between activities while business process execution (workflow) using workflow system means processing the information flow between activities which is usually done in a standardized manner because change is an exception not a rule. So it would be meaningless to execute disaster response processes in a workflow system.

1.3.2.3 Inter-Organizational Coordination/Collaboration

Usually, organizations need to coordinate or collaborate with other organizations to fulfill their tasks. This holds true for business processes as well as disaster response processes.

The difference however is that in case of business process management, organizations know each other in advance and define global inter-organizational processes (e.g. Supply Chain Management) with well defined interfaces of communication and all the collaborating
organizations agree on them. There are few interfaces of interaction, usually done at organization-to-organization level.

In case of disaster response, many organizations get involved to handle the situation and there is no global definition of processes for inter-organizational collaboration. As opposed to business process management, there are many interfaces of interaction where coordination is done based on personal contacts within organizations. Interfaces of interaction are not well designed but rather created in ad-hoc manner which are merely functional (fax, phone, e-mail etc) instead of proper communication systems.

### 1.3.2.4 Monitoring

As already mentioned, within an organization, usually the same tools or means are used for modeling as well as monitoring the current state of activities. As discussed in sections 1.3.2.1 and 1.3.2.2, business process modeling languages are not suitable for disaster response processes and thus cannot be executed in a workflow system. So monitoring of disaster response processes will be meaningless in a workflow system.

Monitoring of business processes in a workflow system means that business goals and KPI’s are met by the processed business objects by a workflow. The goals of disaster response processes on the other hand have nothing to do with business objects and so a different approach is required for monitoring of disaster response activities.

To summarize the inflexibility of current process management approaches including BPM, we can say:

- Only activity sequences are considered while other temporal dependencies are often not considered.
- Dynamic modifications are usually not allowed and complex to handle which is too constrained for disaster management.
- Pre-planned decision-making which is not applicable for dynamic collaborative domains, such as disaster management.

### 1.4 Personal Motivation

Recent disasters in Pakistan which have affected very badly the country’s infrastructure and an enormous amount of life losses motivated me to participate in this project and get an insight into the challenges faced by disaster management and public safety organizations. Examples of recent disasters with accounts of magnitude of casualties and years of happening are given in Appendix-A.
1.5 Chapter Summary

This chapter has covered background information, scope of the work, motivation for the need of a disaster response management solution by giving an account of the flaws of current practices for disaster management. We have also discussed inflexibility of current process management approaches and why business process management is not suitable for disaster process management by providing a comparison between them in terms of process modeling, execution, inter-organizational coordination and process monitoring.

From the discussion, we can conclude that business process management is substantially different from disaster process management and therefore the business process management technology such as workflow systems are not appropriate for managing disaster response processes. So we need a process management solution which not only provides adequate capabilities for modeling the disaster response processes but also covers the execution, monitoring and inter-organizational coordination aspects.
Chapter 2

Introduction

2.1 System Requirements

In the previous chapter, the need for a system to manage disaster response processes is discussed. In order to develop such a system, we have to define the requirements that the system needs to fulfill. In this section, we will present the main system requirements which have been identified based on the analysis of the problem domain:

- A disaster response process consists of several activities (defined in section 3.1.2) which may be pre-defined or created ad-hoc based on the changing situation. Transport sandbags, build dam, treat injured people etc can be examples of activities. The system should allow modeling of disaster response activities and temporal dependencies between the states of different activities. The modeling approach however should be simple, without complex constructs since in a disaster situation, a quick understanding of models is required. It is difficult for human beings to understand complex models [10] in real time.

- Due to the dynamic situation in a disaster, new activities (which have not occurred before) may be required apart from those already planned. So the system should allow ad-hoc creation of activities and dependencies by the command center or the field teams.

- Activities can be of different types e.g. decision making activities, operational activities etc and there are different management processes for them. So it should be possible to model different types of activities differently.

- The system should allow associating governance roles with activities, which means who is responsible or who should be held accountable for the execution of a certain activity. This is very important to define the governance roles constraining the execution of a disaster response activity.

- We describe change in the activity state as the execution of activities and it should be supported by the system. Temporal dependencies that exist between the states of
activities may be violated by each state change. In real disaster scenarios, many activities run concurrently which means change in the state of multiple activities at the same time. This has to be managed by the system in a flexible way where the user of the system can choose between different options e.g. only visualizing the violated dependencies and let the user fix the violation, auto-correction by the system or not allowing state changes at all which cause dependency violations.

- Public safety organizations already use some means to monitor the situation based on the feedback information received from the field. This may include an activity matrix or a map of activities. This implies that the system should provide the same means to visualize the activities and dependencies and each user should be able to visualize the response activities differently, as per convenience. This is important to be supported for users to accept the system.

- The system should allow collaboration of activities by sharing them inside and outside of the organization. Subsequent changes in the activity state should be propagated and integrated properly by the system. Surprising thing is that sharing of activities always takes place between people based on personal contacts rather than between organizations.

- Another important feature of our system is verification of the model which means to point out any inconsistencies (e.g. cycles) in the model at design time. This is useful for users to focus on their main tasks and let the system take care of it.

### 2.2 Technology Requirements

One requirement is that the system should leverage existing collaboration infrastructure rather than building everything from scratch. Another end-user requirement is that the system should be web-based.

Google Wave is an innovative and flexible, communication and collaboration platform based on open standards such as XMPP (Extensible Messaging and Presence Protocol) [13, 14]. Enabling seamless real-time communication between participants from same or different wave providers, Google Wave apart from its default features, lets developers to extend and enhance the functionality using its extension mechanisms.

In a disaster, people from many different organizations get involved and need to collaborate in order to handle the situation. Every organization typically has its own server and Google Wave
provides support for a decentralized infrastructure using its Federation protocol for real time communication and collaboration between participants from different organizations (i.e. servers). This capability also provides a solution for the requirement of inter-organizational sharing of activities.

Any internet enabled device equipped with some standard compliant web browser can access the web-based client of Google Wave from anywhere. Since Google Wave is based on open standards and protocols, any organization can become a wave provider by implementing those protocols.

Google Wave is discussed in more detail in section 2.4.
2.3 Basic Disaster Scenario

*Inter- and Intra-Organizational Activity Coordination during Flood:*

Here we present a disaster scenario of a flood, based on the interviews with public safety organizations to demonstrate what kind of support our system needs to provide for coordination of disaster response activities at inter- and intra-organizational level.

In Figure 3, we have modeled the activities using circles and their dependencies with dotted arrows. Each organization has a command center and field teams. Most of the strategic level decision making about different activities take place in the command centers while field teams are responsible for executing the activities. The three organizations involved in managing the flood disaster response activities include Military, Civil Administration and Irrigation & Power Department.

![Figure 3: A Simple Disaster Response Scenario](image)

The civil administration is rescuing the flood affectees by searching the trapped people, transporting people to the relief camps and treating injured people as well as treating injuries of the workers of Irrigation & Power department during their work. The Irrigation & Power department is responsible for protecting the crop fields and power installations from flood by building a dam and changing the flow of river. They are getting sandbags required to build the dam from Military, which is protecting a nuclear plant by building another dam.
It should be noted that all the activities are running concurrently and different organizations need to coordinate their activities at intra- and inter-organizational level for managing disaster response processes.

Several use cases presented in section 4.7 have been derived from the scenario.
2.4 Primer of Google Wave

Announced on May 27, 2009 at Google I/O conference, Google wave is an innovative online communication and collaboration platform. It has presented a new perspective of the web applications, designed to merge instant messaging, e-mail, social networking and wikis. Developed in Java using OpenJDK, Google wave provides seamless real time communication enabling people to collaborate in more effective ways using rich text, photos, videos, maps, and more. The underlying network protocol for communication is the “Google Wave Federation Protocol over XMPP” which is an extension to XMPP (Extensible Messaging and Presence Protocol) core protocol [RFC3920]. Below we will describe the main concepts of Google Wave platform which are also relevant for the implementation of our system:

2.4.1 Wave A wave comprises a set of concurrently editable structured documents and supports real-time sharing between multiple participants [13]. In simple terms, a wave represents a conversation, with many participants collaborating in this conversation by editing its content at any time, anywhere within the conversation. Participants which may include both human participants and robots (explained below) can be added to the wave at anytime. Essentially a wave itself is simply a container of one or more wavelets which collectively makeup the wave and where the actual conversation takes place.

2.4.2 Wavelet can be defined as a threaded conversation which is spawned from inside a wave, with its own set of participants which is a subset of the participants of the wave. A wavelet is the main mechanism of data access control inside a wave. Only a participant of the wavelet can view and modify the content inside the wavelet. Each wavelet within a wave can have a different set of participants which means that each participant of the wave will get a different view of the wave because he/she may or may not have access to every single wavelet within a wave. The content may include rich text messages, videos, photos, or even gadgets (explained below) like Google maps or other third-party/self-developed gadgets. The content of the wavelet is stored as a set of documents which are described below. In simple words, a wavelet is a collection of participants and a collection of documents that those participants have access to.

2.4.3 Document is the basic unit of content within a wavelet, composed of an XML document and a set of annotations. The documents can be Conversational document (of which the basic one is known as blip) or Data documents. In theory, both are document types but in practice they are treated quite differently, serving different purpose.

- **Blip**: A blip is a conversational document representing a single rich-text message which appears within a wavelet and is the basic unit of threaded conversation. One blip may
have other blips as its children, making a blip hierarchy. There is always at least one root blip in every wavelet.

- **Data documents**: Blips are the documents which are visible to the wavelet participants but data documents on the other hand contain data pertaining to the wavelet but not visible to the user. Data documents are usually used as wavelet’s internal data store. This provides a mechanism for Google Wave Extensions (Gadget, Robot) to place their intermediary data for later usage. For example, the spell checker gadget may store spelling suggestions within data documents. The format of the data is typically key-value pairs.

![Figure 4: Google Wave Entities](image)

2.4.4 Wave Extensions

Google Wave provides two mechanisms for developers to extend the functionality of waves and Google wave client by authoring mini applications; Gadgets and Robots. They can be used in combination and serve different purposes.

2.4.4.1 Gadgets Wave gadgets are one of the two extension mechanisms, used to create small collaborative add-on applications which can be embedded into a wave, with all the wave participants sharing the same gadget state. Gadgets are equipped with two internal state objects; shared and private; which are basically maps of key-value strings. The difference between the two is that shared state is accessible to all the wave participants while the private state object can only be accessed by an individual participant to keep some private information.
not accessible to any other participant of the wave. Developers can use the shared state object to synchronize the view of the gadget for all the participants and enable collaboration based on state changes events of the shared state object. An example of gadget is the Map gadget provided as part of the Google Wave client, to embed Google Maps inside a wave, also shown in Figure 5 below:

Figure 5: Map Gadget inserted into a Wave

2.4.4.2 Robots Robots are developer applications which can serve the role of automated participants of the waves, automating certain tasks in response to particular events. Generally, robots can be more powerful participants as compared to human participants of the wave, in terms of their capabilities assigned to them by their developers. A robot based on its capabilities can create read/modify contents of the wave, read/modify the shared state of a gadget, create new wavelets and blips as well as add/remove participants of the wave. There can be only one instance of a robot per wave but there can be more than one robots participating in a wave.
2.5 Usage of Wave Extension Mechanism in the Context of System

Currently the system contains three gadgets ("Activity Dashboard", "Activity Specification" and "Activity Participant View") and a robot ("UProMan"). The details of these gadgets and robot are described in the Chapter 4 but a quick overview is given as follows:

The main purpose of gadget “Activity Dashboard” is to allow modeling the activities and dependencies as well as for monitoring execution of activities.

The gadget “Activity Specification” allows to specify different details related to the activity (name, location, execution time, governance roles etc).

The “Activity Participant View” enables to view the current state of an activity on activity dashboards of different organizations wherever this activity has been modeled.

The robot “UProMan” (Abbreviated form of Unstructured Process Management) is used for multiple automated tasks (sharing the activities, execution of activities, verification of the model for inconsistencies etc).

2.6 Wave Service Architecture

Since Google Wave Federation Protocol is an open protocol for contributions and improvements, it enables any organization or individual to become a wave provider, sharing waves with others. For example an ISP can provide the wave service on one or more networked servers as a supplement to e-mail or instant messaging services for its users. A wave provider is identified by its Internet domain name. Wave users’ addresses have the same form as an e-mail address i.e. username@domain, where domain represents the wave provider’s internet domain name. Access to all the waves by the wave users is done through their wave providers. It may happen that a wave has participants from different wave providers. In this case wave providers of all the participants on the wave maintain a copy of the wave to serve it to their users. All the updates to the wave are shared using Google Wave Federation Protocol.

The main components of the wave service architecture include wave store and the wave server. The wave store is used to store the wavelet operations submitted by various participants while the wave server communicates with the wave store to resolve wavelet operations by performing operational transformations [18].

Typically a wave is comprised of more than one wavelet. Each of these wavelets can be created by participants belonging to same or different wave providers. Participants belonging to the same domain as the wave provider are called local participants of that wave provider while the
participants belonging to a different domain are called the remote participants. When some participant belonging to a certain wave provider creates a new wavelet, the wave server of that wave provider hosts this wavelet. Hosting the wavelet means that all the operations submitted to this wavelet (by local or remote participants) are processed by the wave server of hosting wave provider using operational transformation. Thus in a certain wave, different wavelets can be hosted by wave servers belonging to different wave providers depending on which participant created the wavelet. The wavelets hosted by a wave server are called local wavelets relative to that wave server while the wavelets hosted by the wave servers belonging to other wave providers are called remote wavelets.

A certain wave server is responsible to store local wavelets and a cached copy of all the remote wavelets to serve waves to its local participants. The advantage of storing the cached copies of remote wavelets is two-fold. For read access, the wave server serves the remote wavelets to its local participants by using the cached copies and thus avoiding a round trip to the hosting wave provider. The second purpose of cached copy is that the wavelet operations submitted by local participants on a remote wavelet are forwarded to the hosting wave server which returns the transformed operations back. The transformed operations are then applied to the cached copy of the remote wavelet and thus the state of a particular wavelet is synchronized across multiple wave providers. The wave servers communicate with each other using Google Wave Federation Protocol. When a wave participant connects via Google wave client, the participant gets a view of a certain wave from its wave server which retrieves local wavelets and cached copies of remote wavelets from its wave store.

Figure 6: Wave Service Architecture
2.7 Chapter Summary

In this chapter, an introduction to the system has been presented by identifying system requirements based on the analysis of problem domain, both from the perspective of end-users as well as the technology requirements. A disaster scenario depicting the response to a flood situation by public safety organizations is presented to give an understanding of the problem domain. The scenario is also used to derive the use cases for the system. Section 2.4 presents an introduction to the Google Wave by describing its different components relevant to our system. Google Wave extension mechanisms including gadgets and robots have also been discussed which play an important role in the development of the system. Section 2.6 provides an overview of the wave service architecture and communication between different wave providers using Google Wave Federation Protocol. In the next chapter we will discuss our proposed solution.
Chapter 3
Proposed Solution

Based on the system and technology requirements described in previous chapter, we will present in this section, our system for managing disaster response activities which includes support for modeling the activities with temporal dependencies between their states, as well as execution, monitoring and inter- and intra-organizational coordination of activities. The system also supports detection of inconsistencies in the disaster response process models, which we call as verification of model. We will describe all these features in their respective sections but the system offers these features in an integrated manner, all at the same time and there are no separate modeling, execution, monitoring and verification phases. We start with the modeling of disaster response processes:

3.1 Modeling

We have already described in chapter 1 that business process modeling languages such as EPC, BPMN etc are not suitable to model disaster response processes. So this means that another modeling approach is required which supports modeling of activities and temporal dependencies between the states of activities. The proposed modeling language [2] will be described here. For clarification, we will present the meta-model of this modeling language, supported by an example.

The meta-model distinguishes between three modeling elements: Activity Type, Activity and Temporal Dependency. It enables modeling of disaster response activities and the various dependencies between their states, as well as supports the execution of activities.

Each activity has an activity type where the activity type allows to define the various states (Plan, Execute, Finish etc) of activity management lifecycle and governance roles (responsible, accountable, consulted etc) that can be associated to an activity. This is because different types of activities (e.g. Strategic, Operational) can have different states in their management lifecycle.

Temporal dependencies (Precedes, Contains, Overlaps etc) exist between the states of different activities which control the execution of activities. We use the temporal relationships proposed by Allen [7]. In contrast to other modeling approaches, we do not make use of gateways (XOR,
OR, AND) for multiple execution paths, to avoid unnecessary complexity; rather it is achieved using activity type.

3.1.1 Activity Type represents the management lifecycle of an activity by describing the activity states and governance rules. Activity types help the process modelers to intuitively plan the response processes and cater for the dynamic events in advance. Activity types differ from each other by their lifecycle and governance rules.

It is defined as: \( AT = (S, f, G) \), where

\( S \) is the finite set of activity states including the start state ‘ss’ and end state ‘se’. An end state means that no further transition is possible. It should be noted that \( ss \neq se \).

\( f: S \rightarrow S \) is the transition function describing possible state transitions. There must not be any strongly connected components (cycles) in the lifecycle because it can create ambiguities. One example of this ambiguity may be that some user might think that the activity is being re-executed even though it has finished already.

\( G \) extends the specification of activity type by describing the governance rules (i.e. who is allowed to perform state transitions). Four governance roles have been defined:

- **Accountable**: Who decides ultimately on the activity.
- **Responsible**: Who will execute the activity.
- **Consulted**: Who should be consulted before making a state change.
- **Informed**: Who should be informed after a state change.

![Diagram](image.png)

Figure 7: Example Activity type without governance rules
3.1.2 Activity An activity is defined as \( A = (\text{id}, \text{name}, \text{type}, \text{cs}, \text{GA}) \), where

- \text{id} is the unique identifier of an activity,
- \text{name} represents the activity name,
- \text{type} is one of the activity type from the pool of existing activity types,
- \text{cs} \in S; describes the current state of the activity. When an activity is created \( \text{cs} = \text{ss} \),
- \text{GA} = P \times G defines the assignment of governance roles to the participants (e.g. commander of the field team).

3.1.3 Temporal Dependency is established between two states of two different activities. These dependencies are based on the time interval relationships defined by Allen [7], also shown in Figure 8. Although thirteen relationships have been defined by Allen, but we use only seven out of them shown in Figure 9, because the other six are just inverse of the others.

Temporal dependency is defined as: \( \text{TD} = (\text{type}, \text{a}_s, \text{s}_s, \text{a}_d, \text{s}_d) \), where

- \text{type} is one of the temporal dependency types,
- \text{a}_s is the source activity of the dependency,
- \text{s}_s is the state of the source activity,
- \text{a}_d is the destination activity of the dependency,
- \text{s}_d is the state of the destination activity.

![Figure 8: Allen's proposed 13 time-interval relationships](image-url)
3.2 Verification

Another important feature of our system is verification of the model. The main purpose of verification is to point out any inconsistencies in the model, by taking into account the currently modeled activities and dependencies between activity states. This is also useful because even in simple models, with few activities and dependencies it may not be trivial to find out inconsistencies. In the context of disaster process management, this feature gets more highlighted for the people involved, to focus only on modeling and coordination of activities rather than spending time searching the model for inconsistencies, each time a dependency is added/removed. This is explained with a simple example followed a more complex example below.

Let say, we have three activities A, B and C. The dependencies established between the activities are shown in Figure 10. We assume that all the dependencies are between state “execute” of three activities. At this moment, the model does not have any inconsistency and it just describes that activity A should be executed before starting execution of activity B, while activity B should be executed before executing activity C. It makes sense because we have a clear order in which the activities have to be executed.
Now suppose that we introduce another dependency “precedes” between states execute of activity C and A. This is shown in Figure 11.

Now this new dependency between activity C and A makes the model inconsistent, in the sense that we do not know in which order the activities have to be executed. In other words, this model demonstrates that every activity precedes itself.

The inconsistency can be figured out easily in the above example, but there can be complex models where it may become very hard to determine if there are inconsistencies. For example, the model shown in Figure 12 below demonstrates a little bit more complex situation in which the inconsistency may not be very easily visible.
In this model, again all the dependencies are between the activity states “execute”. As we can see that execution of activities B, C and D cannot start before starting the execution of activity A. Also the execution of activities B, C and D has to be done before completing the execution of activity A. This is because of the temporal dependency “contains” between activity A and activities B, C and D. Similar is the case between activity E and F.

The dependencies “precedes” between activity D and F as well as between E and A suggest that execution of activity D and E has to be completed before starting execution of activity F and A respectively.

Now after carefully looking at the model, we can see the problem.

- Activity A cannot move to state “execute” until completion of execution of activity E.
- While activity E cannot move to state “execute” until the completion of execution of activity F which on the other hand cannot be executed until activity D completes its execution.
- But activity D cannot start its execution until the start of execution of activity A which cannot move to state “execute” until activity E completes its execution.

To avoid these situations and detect inconsistencies in the model, we use the path consistency algorithm proposed in [7]. Although this algorithm was devised for reasoning about network of interval relationships, which means that from a given temporal constraint network it was able to infer all the other possible temporal constraints. But this algorithm can also be used to detect inconsistencies in a network of temporal constraints.

We make use of Allen’s algorithm [7] after transforming our model of activities and temporal dependencies into a directed graph with activity states representing the nodes of the graph and
dependencies represented as edges between the nodes. An extra step involved in the transformation of model into graph before giving it as input to the algorithm is to connect the states of same activity using the “meets” constraint. The transformation of model to create a graph before using it in the algorithm is shown in Figure 14 below with a simple example involving only two activities and one temporal dependency “overlaps” between the states execute of both activities.

![Figure 13: Model to be transformed into graph used as input to path consistency algorithm](image)

![Figure 14: Graph produced after transformation of model shown in Figure 13](image)

So each time a dependency is added/removed, the system performs the following steps to verify the model for consistency:

- State of the gadget is updated on addition/removal of dependency.
• Robot detects this and loads the model into the activity management engine.

• Activity management engine generates a graph of nodes and temporal constraints as mentioned above, based on the model loaded by the robot.

• Activity management engine processes the graph using path consistency algorithm which basically tries to infer other constraints based on the existing constraints using transitivity table (Table 1) below.

• The path consistency algorithm takes one constraint at one time to infer other constraints. If it cannot infer new constraints or if the inferred constraints do not match the constraint between the nodes, it means the temporal constraint network model is inconsistent and in turn, the model is inconsistent.

• Activity management engine gives feedback if the model is consistent or not.

• The robot provides the gadget with all dependencies which have been considered for verification (because in the meantime the model could have been changed again).

• The gadget evaluates the feedback and if it still applies to the model (i.e. the same dependencies considered for verification as displayed in the model), then the results of the verification are displayed, otherwise the feedback of the verification process is ignored since the model is different than the one passed to the activity management engine for verification.
<table>
<thead>
<tr>
<th></th>
<th>B r2 C</th>
<th>&lt;</th>
<th>&gt;</th>
<th>d</th>
<th>di</th>
<th>o</th>
<th>oi</th>
<th>m</th>
<th>mi</th>
<th>s</th>
<th>si</th>
<th>f</th>
<th>fi</th>
</tr>
</thead>
<tbody>
<tr>
<td>A r1 B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;before&quot;</td>
<td>&lt;</td>
<td>no</td>
<td>info</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
</tr>
<tr>
<td>&quot;after&quot;</td>
<td>&gt;</td>
<td>no</td>
<td>info</td>
<td>m</td>
<td>di</td>
<td>d</td>
<td>f</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
</tr>
<tr>
<td>&quot;during&quot;</td>
<td>d</td>
<td>no</td>
<td>info</td>
<td>d</td>
<td>no</td>
<td>info</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
<td>d</td>
</tr>
<tr>
<td>&quot;contains&quot;</td>
<td>di</td>
<td>o</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>m</td>
<td>d</td>
</tr>
<tr>
<td>&quot;overlaps&quot;</td>
<td>o</td>
<td>d</td>
<td>o</td>
<td>d</td>
<td>f</td>
<td>o</td>
<td>d</td>
<td>f</td>
<td>o</td>
<td>d</td>
<td>f</td>
<td>o</td>
<td>d</td>
</tr>
<tr>
<td>&quot;overlapped-by&quot;</td>
<td>o</td>
<td>f</td>
<td>di</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>&quot;meets&quot;</td>
<td>m</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
</tr>
<tr>
<td>&quot;met-by&quot;</td>
<td>m</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
<td>o</td>
<td>d</td>
<td>s</td>
</tr>
<tr>
<td>&quot;starts&quot;</td>
<td>s</td>
<td>d</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
</tr>
<tr>
<td>&quot;started by&quot;</td>
<td>si</td>
<td>d</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
</tr>
<tr>
<td>&quot;finishes&quot;</td>
<td>f</td>
<td>d</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
</tr>
<tr>
<td>&quot;finished by&quot;</td>
<td>fi</td>
<td>d</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
<td>di</td>
<td>o</td>
<td>m</td>
</tr>
</tbody>
</table>

Table 1: Transitivity table for temporal relations [7]
3.3 Execution

As an integral part of the system, execution means changing the activity states and checking if these state changes cause any violation of temporal dependencies between the states of different activities. The system supports changing in the state of multiple activities at the same time because in real disaster situations many activities are executed concurrently. Temporal dependencies that exist between the states of activities may be violated by each state change.

The system provides the flexibility to handle these dependency violations in two possible ways:

- **Enforce**: If the user changes the state of an activity which violates any temporal dependency, then the system does not allow this state change.
- **Support (Default)**: Make any violated dependencies visible to user (e.g. marking them with red color). In this case, the users will have to resolve the conflicts outside the system and after solving it, the violated dependencies can be set to resolved, which means the system will not display them as violated dependencies anymore.

The system enables the users to choose any of these choices along with modeling the dependency. If the user does not specify the strategy for the treatment of a dependency violation, then the system defaults to ‘Support’.

The purpose of providing these choices is to make the system flexible and leave the decision to the users, who may select from the choices depending on the situation and type of activity. The system maintains a log of all the state changes of the activities as well as the users who made these changes, for future reference.

The pre-requisite to both the strategies (Enforce, Support) for managing violated dependencies is to detect whether the current change in the activity state is contradictory to dependencies connected to the activity. This mechanism is implemented by representing each temporal dependency as a finite state machine (FSM), as will be explained shortly.

Algorithm for detecting dependency violations caused by changes in the activity states is presented below. It takes as input a list of all the state changes performed by the user and after checking each dependency associated with all the activities performing state changes, the algorithm returns the set of dependencies which are contradictory to the state changes. As already mentioned, each temporal dependency has a corresponding finite state machine. The figure below illustrates the finite state machine of the temporal dependency “overlaps”. On the basis of temporal dependency given as parameter, the procedure CheckViolation() selects the respective finite state machine and provides the states of the two connected activities to the
finite state machine as input. On the basis of input, the finite state machine goes into the “neutral” or “violated” state.

**Algorithm:** Detecting Dependency Violations during Execution of Activities

```
input: List M of all the state changes performed by the user
output: Set X containing all the violated dependencies

dependencylist ← GetAssociatedDependencies (M);
counter ← 0;
While (counter < dependencylist.size) do
    state ← CheckViolation(dependencylist[counter], M);
    If (state is violated) then
        X.add(dependencylist[counter]);
    end
    counter++;
end
```

Figure 15 represents finite state machine for the temporal dependency “overlaps”. We suppose that there are two activities A and B and there is a dependency ‘overlaps’ between activity A and B in states ‘sa’ and ‘sb’ respectively. The notation (A:sa) represents that activity A goes into state ‘sa’ while ~(A:sa) denotes activity A changing its state to any successor state of ‘sa’.

![Figure 15: FSM representing temporal dependency ‘Overlaps’](image-url)
Temporal dependency ‘overlaps’ defines that activity A should go first into the state ‘sa’ and then activity B should go into state ‘sb’ while activity A is still in the state ‘sa’. Secondly the state of activity A should be changed first to any successor state of ‘sa’ while activity B remains in the state ‘sb’. The finite state machine is in the state ‘neutral’ in the beginning.

Now let suppose the user changes the state of the activity B to state ‘sb’ which mean the finite state machine goes into the ‘violated’ state because this is contradictory to the dependency ‘overlaps’ between activities A and B. So this is how the system makes use of finite state machines (different for each temporal dependency) to detect violations of temporal dependencies after state changes of activities.

We thought of another strategy apart from Enforce and Support, which can be named as ‘Automate’ which means that if a certain state change violates any temporal dependency, then the system triggers state changes of other activities to avoid violation of the dependencies. But it has not been implemented as part of the system due to the complexities associated to it (e.g. state of some activities may not be changed to desired state). But can be done in the future by devising a protocol for it.

**Example:** Here we present an example demonstrating execution of activities. There are three activities modeled on the activity dashboard of the fire fighters as shown in Figure 16. There is a temporal dependency “Contains” between the state ‘Execute’ of activity

![Figure 16: Example Demonstrating Execution of Activities](image-url)
“Protect Residential Area from Flood” and the state ‘Execute’ of the activity “Transport Sandbags”. This means that the activity “Protect Residential Area from Flood” has to be in state ‘Execute’ before the activity “Transport Sandbags” moves to state ‘Execute’. The other condition of the dependency “Contains” is that the activity “Transport Sandbags” has to move to subsequent state of ‘Execute’ while the activity “Protect Residential Area from Flood” is in the state ‘Execute’.

Now let say the activity “Transport Sandbags” moves into state ‘Execute’ while the activity “Protect Residential Area from Flood” is still in the state ‘Plan’. Surely this violates the temporal dependency “Contains” between the two activities. The system highlights the violated temporal dependency with red color and displays a warning message saying that temporal dependency is violated. This is shown in Figure 17.

![Figure 17: State changes causing violation of temporal dependencies](image-url)
3.4 Monitoring

During a disaster situation, it is very important for organizations involved to be aware of current state of activities and to know if there is any violation of dependencies. This is the essence of monitoring and it is very important for command centers to manage the situation properly. Our system provides an integrated approach for modeling, execution, verification and monitoring of activities and dependencies between them. What this means is that the gadget “Activity Dashboard” in the Model Wave allows modeling of activities and temporal dependencies between the states of different activities. In the same gadget, the current state of an activity is shown and any violations of dependencies during execution of activities are also shown in the gadget “Activity Dashboard”.

Furthermore, by taking advantage of Google Wave’s extension mechanism, the system can also be easily extended to support different kinds of visualization as per requirement. Google Wave enables to run customized and self-developed or third-party mini application within Google Wave. Google map is one such example which has been provided as default gadgets in Google Wave client. For example, Figure 18 demonstrates different activities on the map inserted into a wave, as a way to visualize the current situation.

![Figure 18: An example of visualizing the situation using Google map gadget inserted into a wave. The flooded area is marked by blue pentagon with high tides marked inside the flooded area. The activity transport sandbag is failed because the truck is broken down.](image-url)
3.5 Sharing of Activities

In order to be aware of activities of other organizations, we allow sharing activities between them. Sharing of activities is the main mechanism of coordination and collaboration between different public safety organizations involved to handle disasters. Sharing of activities is required because some of the activities of one organization may depend on the activities of other organizations. Using our system, an activity modeled in the Activity dashboard of one organization can be shared with participants of the activity dashboard in another organization. The robot “UProMan” is used to perform the backend functionality of sharing activities between different Activity dashboards. This functionality is described in the section regarding details of the robot “UProMan” but the basic idea will be described here.

As described before, each activity is represented by an Activity wave where the specification of the activity can be defined. So whenever a participant ‘P’ from an organization ‘A’ wants to share an activity with the participant ‘Q’ of another organization ‘B’, participant ‘P’ navigates to the activity wave corresponding to the activity to be shared, either by clicking the link available on the activity or directly clicking the Activity wave in Google Wave client. In the activity wave, participant P adds participant Q to the activity wave using participant Q’s wave address (i.e. <username>@<domain>). The robot “UProMan” is by-default a participant of all the activity and model waves because it is added whenever a new model wave or an activity wave is created. So as soon as a new participant is added to the activity wave, the robot “UProMan” detects it by receiving an event, and so it makes the activity shared by participant ‘P’ to the activity dashboard of participant ‘Q’, available for modeling. Now when participant ‘Q’ models the shared activity on its dashboard, this activity becomes available to all the participants of the activity dashboard of organization B.

After the shared activity is modeled on the activity dashboard of organization B, the participants of the activity dashboard can change the state of the shared activity and create dependencies with their own activities, like any other non-shared activity. The system, however, differentiates the appearance of shared activities from non-shared activities just to give an indication if a modeled activity is exchanged by the activity dashboard of another organization.

There are two important points to consider here for shared activities:

1) A shared activity is modeled on different activity dashboards, so the change in state of the activity on one dashboard is propagated to all the other dashboards where the activity is modeled.
2) A propagated change in the state of a shared activity in one dashboard may violate its temporal dependencies in another activity dashboard, so these dependency violations have to be resolved similar to the violations caused by execution of non-shared activities e.g. remove the dependency, communicate with participants of other activity dashboards or change the states of activities such that the violations are resolved. The system, however, highlights the violated dependencies.

3.5.1 Example of Sharing an Activity
We explain step-by-step and illustrate using a scenario how sharing of an activity is done in the system.

**Step 1:** The military commander ‘Bob’ wants to share the activity “Transport Sandbags” with the fire fighters, so that they are always up-to-date about the state of this activity. Bob enters the activity wave of the activity “Transport Sandbags”.

Figure 19: Military commander navigating to the activity wave of the activity Transport Sandbags by clicking the icon displayed on hovering over the activity.
Step 2: Bob adds John, the fire fighter commander, to activity wave of Transport Sandbags. This means the activity information is shared between them.

Figure 20: The Military commander adds Fire fighter commander to the activity wave of the activity “Transport Sandbags”
Step 3: The robot “UProMan” detects the event of adding the Fire fighter commander to the activity wave and it makes the activity transport sandbags available for modeling in the activity dashboard of the fire fighter commander.

Figure 21: Activity Transport Sandbags becomes available in the activity dashboard of Fire fighter commander
Step 4: The fire fighter commander integrates the activity Transport Sandbags into its activity dashboard and creates dependencies with other activities.

Figure 22: Shared Activity Integrated and Dependency Created
3.6 Chapter Summary

This chapter describes our proposed solution for managing disaster response processes. The solution provides an approach for modeling, execution, monitoring and sharing of activities at inter- and intra-organizational level.

The main constructs of modeling language include Activity Type, Activity and Temporal Dependency. The system also includes a very useful feature of verification of the model for inconsistencies at design time by checking currently modeled activities and dependencies between their states.

The next chapter describes the implementation details of the system from a software development perspective.
Chapter 4

Implementation Details

This chapter will present inner implementation details of the system by first describing how to develop, deploy and use Google Wave extensions using different API's. Then the functionality and roles of the three developed gadgets and the robot are discussed. Also presented is the system behavior using UML models and diagrams. Static behavior is modeled using Class diagram and Use case diagram while the dynamic behavior is depicted using Sequence diagrams. Details of programming environment, tools, technologies used to develop and deploy the system will also be provided in this chapter. Let's start with the development of Google Wave Gadget.

4.1 How to develop a Google Wave Gadget

In this section, we will describe the process of developing a Google Wave Gadget. From development point of view, among our three gadgets “Activity Dashboard”, “Activity Specification” & “Activity Participant View”, the gadget “Activity Dashboard” involved more complexities due to its GUI which involved functionalities such as drag-n-drop, dynamic connectors between different activities, displaying dialogue boxes on different events, displaying warning/error messages etc. So we will describe the process of developing Google wave gadgets in the perspective of the gadget “Activity Dashboard”. Other two gadgets were developed in the similar manner.

4.1.1 What is in a Gadget?

The building blocks for developing gadgets require an understanding of XML, HTML and JavaScript.

XML is used to write gadget specifications. In simple terms, we can say that a gadget is an XML file containing gadget specifications as well as the instructions on how to process and render the gadget, and is deployed on an application server.

HTML is used to include static content, while JavaScript adds dynamic behavior to the gadgets.
Although gadgets technology can be used to embed third party applications in various Google products such as Gmail, iGoogle, Calendar, Orkut etc but our focus is on gadgets for Google Wave, also known as Wave gadgets. The main purpose of wave gadgets is to enable creation of collaborative applications inside a wave.

The difference between Wave gadgets and gadgets developed for non-wave containers is that non-wave gadgets cannot take advantage of live, collaborative and multi-user environment of Google Wave while a wave gadget lives in a wave and can communicate or access different properties of the containing wave. Although non-wave gadgets are deprived of the benefits of Google Wave, however most of the non-wave gadgets can still run in Wave. The added advantages Gadgets written for Google Wave have over non-wave gadgets include access to more granular state management, determining the current viewer and all other participants of the wave, using the wave playback mechanism etc.

Another difference is that the gadgets running in Google Wave belong to the containing wave while non-wave gadgets sit on a profile page belonging to the owner of the page. So unlike non-wave gadgets which store information on per-user basis, wave gadgets store any gadget related information on per-wave basis. This gadget related information and gadget state are shared among all participants of the wave, enabling live collaboration for wave participants.

4.1.2 Development of Wave Gadget

As mentioned earlier, the ingredients of the wave are XML, HTML and Javascript. But this also makes the development a little bit difficult especially coding with Javascript. So we found a library named ‘cobogwave’ which provides a GWT wrapper for Google Wave Gadget API. This library made our life easier because it enabled to develop gadget using Java language. Of course behind the scenes, the Java code was compiled to produce the required Javascript, HTML.

Details of the programming environment are given below:

4.1.2.1 Programming Language: Java was used as the language for development.

4.1.2.2 Google Web Toolkit (GWT): It is a toolkit for developing and optimizing complex web applications. Its goal is to enhance developer productivity of browser-based applications without worrying about cross-browser compatibility issues and having to be an expert of browser quirks and JavaScript. It enables the development in pure Java language and after compilation; the code is translated to cross-browser optimized HTML and JavaScript. Another nice feature of GWT is the inclusion of development mode server, which is a local server used for development and debugging the application before deploying the application for production. So developers can debug their code by using breakpoints and watches which was
not very easy before, for the web applications. Another benefit of development mode is to save the development time considerably by not having to compile and deploy the application each time a change is made in the code. By using development mode, developer just has to save the modified Java files; press the refresh button in the browser and all the changes will be reflected automatically.

4.1.2.3 Eclipse: Used as an IDE for programming. One reason for choosing eclipse as an IDE was the availability of Google plugin for eclipse.

4.1.2.4 Subclipse: Subclipse is an Eclipse plug-in providing support for Subversion within the Eclipse IDE.

4.1.2.5 Google Plug-in for Eclipse: This plug-in facilitates the development of Web application projects using GWT and deployment on Google App Engine. It provides a set of software development tools that enable Java developers to design, build and deploy cloud based applications very efficiently. These powerful tools enable the developer to focus on application logic and are designed specifically for Eclipse Java developers to quickly develop high quality applications for the Google cloud.

After installing the plug-in, a new wizard becomes available for creating Web Application project using GWT SDK and/or Google App Engine SDK as shown in Figures 23 and 24:

![Image of wizard for Web Application Project after Installing Google Plug-in for Eclipse]
Apart from this wizard, three buttons are also added to the main menu for creating New Web Application Project, Compiling GWT Project and Deploying the project to Google App Engine. This is highlighted in Figure 25 below:

Figure 24: Web Application Project Specifications

Figure 25: Menu for creating, compiling and deploying Web Application Project
4.1.2.6 Google App Engine: App Engine is Google’s cloud environment to host web applications. We used it to deploy our developed gadgets and robot, which were then served to be part of the containing wave. App Engine offers simple administration, easy deployment and effortless scalability with the growing needs.

App Engine currently supports applications developed in Java, Python and Go. This means that App Engine is empowered with Java, Python and Go run time environments. Since we developed our gadgets and robot in Java, so we will focus on Java technology. Google App Engine’s Java run-time environment includes support for developing applications using standard Java web development tools and API standards. The application communicates with the environment using Java Servlet standard. The App Engine Java SDK supports application development using Java 5 or 6. The application accesses most App Engine services using standard Java APIs. One such service includes App Engine data store, for which implementations of Java Persistence API (JPA) and Java Data Objects (JDO) have been included in the Java SDK.

With App Engine, you only pay for what you use but the usage of 500 MB data storage and CPU and bandwidth to enable an application for efficiently serving about 5 million views a month are absolutely free of cost.

For brevity, some of the salient features of Google App Engine are listed below:

- Run-time environment support for applications written in Java, Python and Go.
- Persistent data storage allowing queries, sorting and transactions.
- Applications running reliably supporting automatic load balancing under heavy load and scalability with large amounts of data.
- Memcache service for achieving high performance with in-memory key value cache. It can be useful for storing intermediate data that does not need persistence and transaction features but fulfils high speed data access requirement for the applications.
- APIs for user authentication and using Google Accounts for sending emails.
- Scheduler for triggering events at specified times and intervals.
- Support for deployment of multiple versions of an application.
- Web-based administration console to manage your applications running on App Engine.

4.1.2.7 Using the CobogWave API:

As mentioned earlier, that the installation of Google plug-in for Eclipse will make three buttons available in the main menu. Developers can either create a Google Web Application project by clicking the New Project option available under the File menu or the first of the three buttons
available in the main menu. After creation of the project, include WaveGadget.gwt.xml by adding the following line to your project’s gwt.xml:

```xml
<inherits name='org.cobogw.gwt.waveapi.gadget.WaveGadget'/>
```

Next, create a class that extends WaveGadget class as follows:

```java
@Gadget.ModulePrefs(title = "Sample Gadget")
public class SampleGadget extends WaveGadget<UserPreferences> {

    public SampleGadget() {
    }

    protected void init(UserPreferences preferences) {
        getWave().addParticipantUpdateEventHandler(new ParticipantUpdateEventHandler() {
            public void onUpdate(ParticipantUpdateEvent event) {
                //handle participants added, also called when wave with gadget is opened.
            }
        });
        getWave().addStateUpdateEventHandler(new StateUpdateEventHandler() {
            public void onUpdate(StateUpdateEvent event) {
                //handle state changes.
            }
        });
        ...init your widgets...
        RootPanel.get().add(<you widget>);
    }
}
```

WaveFeature class provides abstraction to the wave hosting this gadget. Four events are available to act to state changes on the wave. These include StateUpdateEvent, PrivateStateUpdateEvent, ParticipantUpdateEvent and ModeChangeEvent. Event handlers can be registered to act on changes using methods of the WaveFeature class. This makes the foundation for developing a gadget using Cobog-Wave API. Rest depends upon the logic of the application.

For the gadget “Activity Dashboard”, we needed to allow the user to model activities and make connections between the states of activities using connectors. So for this purpose, we also made use of another third party API named “gwt-connectors” which provided dynamic connector capability to our application.

### 4.2 Deploying the Gadget

Once the application is ready, it can be deployed to Google App Engine. But App Engine requires an Application Id for each application to be deployed. For this purpose, go to [http://appengine.google.com](http://appengine.google.com) and login with your Google account. After that click on the ‘Create Application’ button which will open a form requiring Application Id, Application title and other optional inputs. Maximum 10 applications can be registered per developer account (i.e. Google account). After filling the form for creating application with the specified application id, this application will appear under the heading “My Applications”. By clicking on the application title (displayed as hyperlink) of an application will open the administration console for that
application where different parameters can be specified. After the application is created on Google App Engine with the specified application id, developer can use the button provided by Google plug-in for Eclipse to deploy application on the Google App Engine. When this button is pressed, a dialogue box will be displayed as shown in Figure 26.

![Figure 26: Dialog for deploying Web Application Project to Google App Engine](image)

Browse the project containing the application, enter your Google account email address and password same as the one used for logging into Google App Engine. After filling in this information, click on the ‘App Engine Project Settings’ link. It will open another dialogue box as shown below in Figure 27:

![Figure 27: Dialog for Google App Engine Project Settings](image)
In this dialogue, enter the Application ID specified during creation of application on the App Engine. You can also specify the version of the application.

After this information is filled, the application is ready for deployment on the App Engine. Clicking Ok will start the process of deployment after which the application (in our case gadgets and robot) will be available. Since we have developed gadgets and robot and these are not standalone applications but rather these are contained within a wave, so our applications are not similar to ones accessible by typing address in the browser.

So for the gadgets to run, log on to Google Wave using Google account user name and password. The following screen will appear.

Create a new wave by clicking on the icon ‘Blank Wave’. This will create a blank wave with only one participant as shown below in Figure 29:
Clicking on the green icon to add gadgets by URL (highlighted above in Figure 29) will ask for the URL of the deployed gadget. Google wave requires the URL of the gadget specification XML file which is generated when the project is compiled. This file can be found under the war directory of the project as shown below in Figure 30:

But it can be cumbersome for users to add gadget by URL each time they want to add one. So to ease this process, we have the option to create an extension (gadget or robot) installer.
Details of how to create extension installers can be found at [37] but we will write the main points here.

### 4.3 Using Wave Extensions using Extension Installer

#### 4.3.1 Creating an Extension Installer

An extension installer is a package consisting of three parts which are extension metadata, one or more hooks and actions.

Extension metadata describes the extension such as its name, what is does, the author of extension and its current version.

Hooks describe the interfacing of extensions with the Google Wave client, telling where and how the extension works within the client. One example is

```xml
<menuHook location="NewWaveMenu" text="New Activity Dashboard">
```

which describes that there will be a new option “New Activity Dashboard” under the New Wave menu.

What happens when a hook is triggered is described by action(s). Example of an action can be to insert a gadget into the wave upon triggering of the corresponding hook.

An extension installer is referenced by an Extension Manifest, which is an XML file containing all the three parts of the installer i.e. extension metadata, hooks and actions.

Below is the manifest file of a very simple extension installer that we created to add a “New Activity Dashboard” option in the NewWave menu. When this menu option will be clicked, it will create a new blank wave and add robot ‘UProMan’ as its default participant. The robot (as per its functionality) in turn will automatically insert the gadget ‘Activity Dashboard’ on this newly created wave.

```xml
<extension
    name="Unstructure Processmanagement"
    description="Useful for coordinating and synchronizing your activities on the intra-and interorganizational level"
    thumbnailUrl="http://upromanrobotlocal.appspot.com/public/installericon.png"
    version="6">

    <author name="SAP"/>

    <menuHook location="NewWaveMenu" text="New Activity Dashboard">
        <createNewWave>
```
Now the next step is to insert this extension installer in the wave. But for this the wave needs to have an installer element and there is a special extension “Extension Installer Creator” which makes it convenient to insert installer elements in the waves. This special extension can be installed from [https://wave.google.com/a/wavesandbox.com/waveref/wavesandbox.com/w+PgRT7264Y](https://wave.google.com/a/wavesandbox.com/waveref/wavesandbox.com/w+PgRT7264Y) for Wave Sandbox and from [https://wave.google.com/wave/waveref/googlewave.com/w+GhUsKuWfC](https://wave.google.com/wave/waveref/googlewave.com/w+GhUsKuWfC) for Wave Preview.

After the “Extension Installer Creator” is installed, there will be a new icon visible in the editing toolbar. To insert your created extension installer, click on this new icon and paste the URL of the Extension Manifest XML file in the text box. This is shown in Figure 31 below.

![Figure 31: Using Extension Installer Creator](image)
After installing your extension installer by giving the URL of the extension installer manifest file shown above; will install the extension installer with the specified metadata, hooks and actions. This means that a new sub-option “New Activity Dashboard” will be available under the “New Wave” menu. This is shown in the Figure 32 below. As the installer XML file describes, clicking on the “New Activity Dashboard” option will create a new wave with robot “UProMan” as its initial participant.

Figure 32: Google Wave Client with Extension Installer enabling to create New Activity Dashboard and Robot as the default initial participant

4.4 System Implementation in Context of Google Wave

Three gadgets have been developed namely “Activity Dashboard”, “Activity Specification” and “Activity Participant View”). The system also contains a robot named “UProMan” for automation of different tasks. The purpose and functionality of each one of them is described below.

The gadget “Activity Dashboard” is used for modeling the activities and creating dependencies between the states of different activities as well as for execution of activities. Warnings and error messages during execution of activities and verification of the model for inconsistencies are also displayed by the gadget “Activity Dashboard”. Since this gadget presents the current
view of the model of disaster response processes planned by a public safety organization, so we call the wave containing this gadget as “Model Wave”. “Model Wave” by default only contains the gadget “Activity Dashboard”. In a typical setting, each organization will model its processes on a separate Model Wave via gadget “Activity Dashboard” with participants of the Model Wave belonging to that organization. So the model of an organization will only be accessible to the participants of the model wave (who have been added explicitly to the model wave) and in turn to the designated members of that organization. This implicitly provides a solution to the privacy issues.

The detailed specification of each activity modeled in the Activity Dashboard is represented by another wave, which we call as “Activity Wave”. This means that there will be as many Activity Waves as there are distinct activities modeled in the Activity Dashboard. Activity Wave also provides a solution for sharing the activities at an intra- and inter-organizational level. As already mentioned, sharing is done at people-to-people level. So in order to share an activity of organization ‘A’ with a member of another organization ‘B’, a member of organization ‘A’ being already a participant of the Activity Wave adds the member of organization ‘B’ as a participant to Activity Wave corresponding to the activity being shared. When an activity is shared with another organization, this activity becomes available for modeling in the Activity dashboard of that organization. This means one activity can be modeled on Activity dashboards of multiple organizations. But the important point to note is that even if an activity is modeled in multiple Activity Dashboards, it is represented by only one Activity Wave.

Activity Wave contains gadget “Activity Specification” and initially one instance of “Activity Participant View”. As the name suggests, the gadget “Activity Specification” allows to specify different details related to the activity which currently includes name of the activity, the responsible person for execution of the activity, when will the activity be executed, the location of execution, governance roles associated to the activity and assignment of participants to governance roles. Other details can be added in the future as per requirement. The other gadget on the Activity Wave is “Activity Participant View” which enables to view the current state of an activity on activity dashboards of different organizations wherever this activity has been modeled. So an Activity wave can contains as many instances of the gadget “Activity Participant View” as many Activity dashboards this activity has been modeled in, each representing the current state of activity in that Activity Dashboard.

The robot “UProMan” (Abbreviated form of Unstructured Process Management) is used for multiple automated tasks. All the Activity and Model Waves include Robot “UProMan” as a default participant, which mainly enables to execute the protocol for sharing the activities, communication with Activity Engine for facilitating execution of activities and verification of the model for inconsistencies. The robot “UProMan” performs other tasks as well which will be
described in detail afterwards. Figure 33 below provides an overview of system design, describing the relationship between the three gadgets.

![Diagram: System Design in the Context of Google Wave](image)

**Figure 33: System Design in the Context of Google Wave**

Next sections will describe the detailed functionality of gadgets “Activity Dashboard”, “Activity Specification”, “Activity Participant View” and robot “UProMan”.
4.4.1 Gadget "Activity Dashboard"

As the name suggests, this gadget plays the role of a dashboard for activities which enable the people involved in disaster response management to model and monitor activities in an integrated manner. The gadget Activity Dashboard is inserted automatically by the robot “UProMan” on creation of a new model wave. This gadget serves the purpose to:

a) Collaboratively Plan the response processes by modeling activities and creating dependencies between the states of different activities.
b) Integrate activities shared by participants of the activity dashboards of other organizations.
c) Support execution of activities and detecting dependency violations.
d) Support verification of the model for inconsistencies.
e) Display warnings and errors during execution of activities and verification of models.
f) Monitor the current states of different activities and their dependencies.
g) Navigate to the activity wave corresponding to the activity.

The gadget uses its shared state object to store and load state as a HashMap of <key,value> pairs, described in the table below.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity_A_waveid</td>
<td>waveid</td>
</tr>
<tr>
<td>activity_A_coordinates</td>
<td>x,y</td>
</tr>
<tr>
<td>activity_A_name</td>
<td>Name of the activity. Also used by the robot “UProMan” to give title of the activity wave corresponding to the activity. Where A is same number as in activity_A_coordinates and activity_A_waveid</td>
</tr>
<tr>
<td>activity_A_state</td>
<td>Current State of the Activity (e.g. Plan, Execute, Finish,</td>
</tr>
<tr>
<td><strong>activity_A_operation</strong></td>
<td>‘createWave’</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>

This <key,value> pair is inserted temporarily into the gadget state to give a signal to robot that a new activity is created and in response, the robot saves this <key,value> pair in the App Engine datastore if not saved already. The case where the robot does not find an entry in the datastore with the same activity id, it means this activity is newly created and not the shared one and thus a new activity wave corresponding to the activity has to be created. Saving this entry in the datastore ensures that if this activity is modeled in another activity dashboard, then it is linked with the existing activity wave and not creating a new one.

<table>
<thead>
<tr>
<th><strong>activitylist_A</strong></th>
<th>Activity X Data, Activity Y Data</th>
</tr>
</thead>
</table>

This <key,value> pair is used to display activities in the shared activity list panel. These activities are those which have been shared to the participant A but which have not yet been modeled in the current activity dashboard. where A is participant id and the value of this entry contains a comma separated list of activity data, each of which has the format “activityId|activityName|activityWaveId|activityWaveletId”

<table>
<thead>
<tr>
<th><strong>dependency_A_connection</strong></th>
<th>activtiy_X_connectionPoint, activity_Y_connectionPoint</th>
</tr>
</thead>
</table>

where A is a random id (long). and connectionPoint is the point at which the dependency line is connected to the activity widget. activtiy_X and activity_Y are the activities which are connected by the dependency line.

<table>
<thead>
<tr>
<th><strong>dependency_A_coordinates</strong></th>
<th>corner points</th>
</tr>
</thead>
</table>

where A is a random id (long) each corner point represents the x,y coordinates of different corners of dependency line => important to load and save the position of the dependency connector): point1_left,point1_top; point2_left,point2_top; Whereby left and top are ints and different points are separated by an ; (semicolon)

<table>
<thead>
<tr>
<th><strong>dependency_A_finalpoints</strong></th>
<th>startLeft,startTop;endLeft,endTop</th>
</tr>
</thead>
</table>

where A is a random id (long) And finalPoints are the starting and ending points of the dependency connector separated by an ; (semicolon)
dependency_A_type | dependencyType

where A is a random id (long)
and dependencyType can be one of the value from the list: {Precedes, Meets, Overlaps, Contains, Starts, Equals, Finishes}

dependency_A_data | Activity_X,State_M;Activity_Y,State_N

A dependency is established between two states of two activities (e.g. Activity “Build Dam” in state “Fail” STARTS with activity “Warn People” in state “Execute”).

where X is the activity id of the first activity, M is the state id of the source activity X, Y is the activity id of the second activity and N is the state id of the destination activity Y

### 4.4.2 Gadget “Activity Specification”

The gadget “Activity Specification” is inserted automatically by the robot “UProMan” on creation of a new activity and in turn a corresponding new activity wave. Each activity wave contains one instance of the gadget “Activity Specification”.

The gadget “Activity Specification” enables participants to define activity specification details. In other words, it allows to:

a) Set/Change the name of the activity, time and location for execution of activity.
b) Define the management lifecycle of an activity by specifying different states and transitions, depending on the type of the activity.
c) Assign governance roles (accountable, responsible, informed and consulted) to the state transitions as well as assigning participants to these governance roles.

The gadget uses its shared state object to store and load state as a HashMap of <key,value> pairs, in the table below.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity_data</td>
<td>Name;Time;Location</td>
</tr>
<tr>
<td>activity_state_A</td>
<td>StateName</td>
</tr>
</tbody>
</table>

Represents the activity name, time and location of its execution.
where A is a random id (long)

This entry represents different states of the activity management lifecycle.

<table>
<thead>
<tr>
<th>transition_A_connection</th>
<th>SourceStateId, DestinationStateId</th>
</tr>
</thead>
<tbody>
<tr>
<td>where A is a random id (long)</td>
<td></td>
</tr>
</tbody>
</table>

Defines the possible transitions between the states of the activity.

<table>
<thead>
<tr>
<th>transition_A_roleassignment</th>
<th>Role1, Role2, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>where A is a random id (long)</td>
<td></td>
</tr>
</tbody>
</table>

Role1 and Role2 are the different roles (Responsible, Informed etc) assigned to state transitions. Multiple roles are separated by commas.

<table>
<thead>
<tr>
<th>role_A_assignment</th>
<th>Participant1, Participant2, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>where A is a random id (long)</td>
<td></td>
</tr>
</tbody>
</table>

Participant1 and Participant2 are the different participants assigned to governance roles. Multiple participants are separated by commas.

4.4.3 Gadget “Activity Participant View”

Activity Wave initially contains one instance of “Activity Participant View”. Each instance of this gadget provides a read-only view of the current state of an activity on activity dashboards of different organizations wherever this activity has been modeled. Whenever an activity is modeled on an activity dashboard, a new gadget “activity participant view” is inserted in the activity wave corresponding to the activity. The gadget helps the participants of the activity to resolve state conflicts by looking at current state of an activity on different activity dashboards.

This gadget is inserted by the robot “UProMan” into activity wave for each activity dashboard where the activity corresponding to the activity wave is modeled. This means an activity wave will contain as many “Activity Participant View” gadget instances as the number of activity dashboards where the corresponding activity is modeled. This is in contrast to the gadget “activity_specification” which is inserted only once for an activity. The data model of the gadget is given in the table below.
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>model_waveid</td>
<td>Wave id the model wave whose Activity Dashboard has the activity modeled with this current state.</td>
</tr>
<tr>
<td>model_wave_title</td>
<td>Title of the model wave containing activity dashboard</td>
</tr>
<tr>
<td>current_activity_state</td>
<td>Current state of the activity (e.g. Plan, Execute, Fail, Finish, Cancel etc)</td>
</tr>
</tbody>
</table>

### 4.4.4 Robot “UProMan”

In this project we have implemented the robot “UProMan” (Abbreviated form of Unstructured Process Management) using Google Wave Robots API to provide the backend/automated functionality required by the system. A robot is an automated participant on a wave with many capabilities like any other human participant such as reading the contents of a wave in which it is participating, inserting/modifying/deleting the contents of the wave, adding or removing wave participants, creating new waves and blips etc. All the Activity and Model Waves include Robot “UProMan” as a default participant. UProMan in this project enables to perform the following tasks:

- Inserting gadget “Activity Dashboard” on creation of a model wave.
- Inserting one instance of gadgets “Activity Specification” and “Activity Participant View” in the newly created Activity Waves on creation of a new activity and inserting subsequent instances of gadget “Activity Participant View” on modeling of an activity in other activity dashboards.
- Read/Insert/Update/Delete entries (<key, value> pair) in the gadgets’ shared state objects.
- Adding participants to the model waves and activity waves on certain events.
- Storing and loading data to/from the Google AppEngine data store.
- Communicating with the activity management engine to support verification of model for inconsistencies and execution of activities.
- Supporting navigation to the activity wave corresponding to the activity.

The Google Wave Robot API provides functionality to respond to different events generated in the wave where the robot is a participant. Following is the functionality performed by the robot “UProMan” on generation of various events:
Each time a model wave is created:

- Store the wave id of the wave containing activity dashboard and wave participants in the App Engine DataStore.

Each time an activity is created using New Activity button:

- Create an activity wave, with all the participants of the model wave added as the participants of activity wave.
- Insert gadget “Activity Specification” in the activity wave.
- Insert gadget “Activity Participant View” in the activity wave.
  - Store in the state of the gadget “Activity Participant View”:
    - Title of the model wave containing activity dashboard.
    - Current State of activity in the activity dashboard.
    - Wave-id of the model wave containing activity dashboard to navigate from activity wave back to the activity dashboard.
- Add the activity related data (activity id, name and activity wave id) to the state of gadget activity dashboard for the participant who created this activity in order to make this activity available for modeling to this participant in other activity dashboard where he/she is participating.
- Store the activity wave id and its participants in the App Engine datastore.

Each Time a participant is added to the Model Wave:

- Add the participant to all the activity waves of all the activities currently modeled in the activity dashboard.
- Add the participant to the participant list against the wave id of the model wave in the App Engine datastore.
- Search for all the activity waves where this participant is participating.
- From the activity waves searched in the above step, make all the corresponding activities available to the participant for modeling which have not yet been modeled in this activity dashboard.

Each Time a participant is added to an activity wave (Sharing an Activity):

- Search all the model waves where this participant is participating.
- Make the activity available for modeling to the participant in all the activity dashboards searched in the above step except those where this activity is already modeled.
- Add the participant to the participant list against the activity wave id in the App Engine datastore.
Each time an activity is modeled using the button to move an activity from the Available Activities List:

- Remove the activity id from the activity list in the gadget state for the participant who moved the activity from available activities list to the model.

Each time the title of an activity wave is changed:

- Search in App Engine datastore for all the activity dashboards where the corresponding activity is modeled or available for modeling.
  - Change the name of activity in all the activity dashboards.
  - Change the name of the activity in all the available activities Lists.

Each time the title of model wave is changed:

- Get all the activity waves of the current activities in the model
- Change the name in the gadget “Activity Participant View” in all the activity waves searched in the above step.

The robot UProMan implements a very basic type of cache to optimize the performance of GUI. Technically speaking the robot developed using Google Wave Robots API is actually an HTTP servlet having its own context, allowing to store data as <key,value> pairs. Our robot needs to store some information mentioned in the following table to the persistent storage for performing its required functionality. Currently we are using Google App Engine datastore to store this information. But since communicating with datastore is an expensive operation, so in order to avoid delays in rendering the updated state of gadgets the robot saves this information first in the ServletContext and after gadgets state is updated, the robot saves the information stored in the context to the datastore for later usage. Following information is stored in the ServletContext and subsequently to the Google App Engine datastore.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity_A_name</td>
<td>Title of the activity wave</td>
</tr>
<tr>
<td>where A is activity id which is a random number (long).</td>
<td></td>
</tr>
<tr>
<td>activitywave</td>
<td>A</td>
</tr>
<tr>
<td>In the key ‘activitywave’ and ‘participants’ are fixed text while A is the activity id of the corresponding activity. waveld &amp; waveletId represent the string values of wave id and root wavelet id of the activity wave respectively. The value contains the comma separated list of</td>
<td></td>
</tr>
</tbody>
</table>
id’s of activity wave participants.

<table>
<thead>
<tr>
<th>modelwave</th>
<th>waveId</th>
<th>waveletId</th>
<th>blipId</th>
<th>participants</th>
<th>Participant X Id, Participant Y Id</th>
</tr>
</thead>
</table>

In the key ‘modelwave’ and ‘participants’ are fixed text while waveId, waveletId and blipId represent the string values of wave id, root wavelet id and blip id containing the gadget activity dashboard. The value contains the comma separated list of id’s of model wave participants.

| activity_A_operation | ‘createWave’ |

This <key,value> pair is inserted temporarily by the gadget activity dashboard into its state to give a signal to robot that a new activity is created and in response the robot saves this <key,value> pair in the context/datastore if not saved already. The case where the robot does not find an entry in the context with the same activity id, it means this activity is newly created and a new activity wave corresponding to the activity has to be created. Saving this entry in the context/datastore ensures that if this activity is modeled in another activity dashboard, then it is linked with the existing activity wave and not creating a new one.

<table>
<thead>
<tr>
<th>Wave</th>
<th>CurrentView</th>
<th>A</th>
<th>domain</th>
<th>waveId</th>
<th>waveletId</th>
<th>blipId</th>
<th>waveId</th>
<th>waveletId</th>
</tr>
</thead>
</table>

In the key ‘Wave|CurrentView’ is fixed text while A is activity id and domain, waveId, waveletId and blipId represent the string values of domain of the wave provider (e.g. sandbox, googlewave or third party), wave id, root wavelet id and blip id of the model wave containing the activity dashboard where the activity represented by activity id is modeled. The value contains the string values of wave Id and wavelet id of the activity wave corresponding to the activity. The purpose of this entry is to know the activity wave id and activity wavelet id of the activity wave corresponding to each activity modeled in some activity dashboard.

<table>
<thead>
<tr>
<th>activity_A_waveid</th>
<th>waveId</th>
<th>waveletId</th>
</tr>
</thead>
</table>

In the key A is the activity id and ‘activity’ and ‘waveid’ are fixed texts. The key contains the string values of wave Id and root wavelet id of activity wave corresponding to the activity.
4.5 Main Software Concepts / Programming Abstractions

Figure 34: Main Software Abstractions and Relationships between them
4.6 Use Case Diagram

![Use Case Diagram]

Figure 35: Use Case Diagram
4.7 Use Cases

4.7.1 Use Case Id: UC1

Use Case Name: Create Activity Dashboard

Initiator: Participant via Google Wave Client

Goal: To enable the people involved in disaster response management to create a new activity dashboard as a platform to model activities and dependencies as well as coordinate activities on the intra and inter organization level.

Pre-Conditions:

1. The participant has an account on Google Wave
2. The participant has installed the extension installer “Unstructured Process Management”.

Post-Conditions:

1. New Activity Dashboard is created with the initiating wave participant and the robot “UProMan” as the initial participants.

Main Success Scenario:

1. The participant clicks on the menu item “New Activity Dashboard” available under the “New Wave” menu item.
2. The system adds a new Activity Dashboard gadget with buttons to create new activities and dependencies.
3. The system searches for any other activities that the initiating participant is part of and if it finds any activities, it displays them on the activity dashboard that enables modeling of existing activities (created in other activity dashboards) on this activity dashboard.
4. The participant views the new Activity Dashboard with the options to create new activities and dependencies or model existing activities.
4.7.2 Use Case Id: UC2

Use Case Name: Create Activity

Initiator: Participant via Google Wave Client

Goal: To enable people involved in disaster response management to create a new activity in the model and to collaborate with other participants.

Pre-Conditions:

1. The participant has created a new Activity Dashboard or is already a participant of an existing Activity Dashboard.

Post-Conditions:

1. New Activity is created on the Activity Dashboard.
2. Activity Wave corresponding to the new activity is created with all the participants of the model wave.

Main Success Scenario:

1. The participant clicks on the button to create a new activity.
2. The system displays a dialogue box to enter the name and initial state of the new activity.
3. The participant enters the activity name and selects the initial activity state from the list.
4. The system creates a new activity on the Activity Dashboard with the specified activity name and activity state.
5. The system creates a new activity wave corresponding to the newly created activity with all the participants of the model wave, enabling the participants to specify the details of the newly created activity.
6. The participant views the new Activity on the Activity Dashboard with the specified activity name, state and a link to navigate to the corresponding activity wave.
4.7.3 Use Case Id: UC3

**Use Case Name:** Share an Activity

**Initiator:** Participant via Google Wave Client

**Goal:** To enable people involved in disaster response management to share an activity to other participants at the intra or inter-organizational level.

**Pre-Conditions:**

1. The activity to share and its corresponding activity wave have been created by one of the participants.
2. The initiator participant is part of the activity wave corresponding to the activity to be shared.

**Post-Conditions:**

1. The activity is available for modeling in the Activity Dashboard of the participant with whom the initiator participant has shared the activity.
2. The participant with whom the activity is shared is a participant of activity wave corresponding to the shared activity.

**Main Success Scenario:**

1. The initiator participant adds the participant (with whom the activity has to be shared) to the activity wave corresponding to the shared activity.
2. The system searches for all the Activity Dashboards in which the newly added participant is part of.
3. The system makes the shared activity available for modeling to all the Activity Dashboards of the new participant.
4. The new participant views the shared activity in the list of activities which have been shared to this participant and which have not yet been modeled in the Activity Dashboards.
4.7.4 Use Case Id: UC4

Use Case Name: Model a Shared Activity

Initiator: Participant via Google Wave Client

Goal: To enable a participant involved in disaster response management to model an activity in its Activity Dashboard, which is actually created in another Activity Dashboard and has been shared with this participant. By modeling the shared activity, in turn, shares this activity with other participants on this Activity Dashboard.

Pre-Conditions:

1. The activity to be modeled is available to the initiator participant’s Activity Dashboard in the available activities list.

Post-Conditions:

1. The shared activity is modeled on the Activity Dashboard and all the participants of the Activity Dashboard can view the modeled activity, same as the initiator participant.
2. All the participants of the model wave containing current Activity Dashboard have been added as participants of the activity wave corresponding to the modeled activity.

Main Success Scenario:

1. The initiator participant selects and moves an activity from the list of available activities which have been shared to the initiator participant and which have not yet been modeled in the current Activity Dashboard.
2. The system created the selected activity on the Activity Dashboard.
3. The system adds all the participants of the model wave to the activity wave corresponding to the modeled activity.
4. The system links the modeled activity to the same activity wave which was created after the activity was initially created in some other Activity Dashboard.
5. All the participants on the Activity Dashboard view the shared activity modeled on their Activity Dashboard in a similar way as a new activity is created in use case UC2.
4.7.5 Use Case Id: UC5

Use Case Name: Change Activity Name

Initiator: Participant via Google Wave Client

Goal: To enable a participant involved in disaster response management to change the name of an activity for some reason and get this change propagated in all the Activity Dashboards, where this activity is modeled or available for modeling.

Pre-Conditions:

1. The activity whose name is to be changed and its corresponding activity wave have been created by one of the participants.
2. The initiator participant is part of the activity wave corresponding to the activity whose name is to be changed.

Post-Conditions:

1. The name of the activity is changed in all the Activity Dashboards, where this activity is modeled or available for modeling.
2. The title of the activity wave corresponding to the activity is changed.

Main Success Scenario:

1. The initiator participant navigates to the activity wave corresponding to the activity by clicking on the button available in the hover toolbar of the activity.
2. The system navigates to the corresponding activity wave containing the specification details about the activity.
3. The initiator participant changes the title of the activity wave as required.
4. The system searches for all the model waves where the activity corresponding to the activity wave is already modeled or available in the list of activities to be modeled.
5. The system gets the updated title of the activity wave and uses it as the new name of the activity in all the Activity Dashboards returned as a search in step 4.
6. All the participants who could view the activity in their Activity Dashboards (modeled/available for modeling), now view the activity with the new name.
4.7.6 Use Case Id: UC6

Use Case Name: Add Participant to Activity Dashboard

Initiator: Participant via Google Wave Client

Goal: To enable a participant involved in disaster response management to add the participant to the Activity Dashboard and in turn, make all the currently modeled activities in the Activity Dashboard available to new participant.

Pre-Conditions:

1. The model wave containing Activity Dashboard in which the new participant is to be added has already been created by some participant.
2. The initiator participant is already a part of the model wave containing Activity Dashboard in which the new participant is to be added.

Post-Conditions:

1. The new participant is added to the model wave containing Activity Dashboard as well as to all the activity waves corresponding to the activities currently modeled in the Activity Dashboard.
2. All the activities currently modeled in the Activity Dashboard are visible to the new participant.
3. All the activities, the new participant is part of, and not modeled in the participant’s Activity Dashboard are available to the new participant for modeling.

Main Success Scenario:

1. The initiator participant invites the new participant to the model wave containing Activity Dashboard by clicking on the button “Add Participant”

2. The system searches for all the activity waves corresponding to the activities currently modeled in the Activity Dashboard and add the new participant to all those activity waves.
3. The system makes those activities available in the new participant’s activity dashboard for modeling, which have not yet been modeled.
4. The new participant views all the activities modeled in the current Activity Dashboard as well as the list of activities available for modeling in the participant’s other activity dashboards.
4.7.7 Use Case Id: UC7

**Use Case Name:** Create Dependency

**Initiator:** Participant via Google Wave Client

**Goal:** To enable a participant involved in disaster response management to create a dependency between the states of different activities.

**Pre-Conditions:**

1. Atleast two activities are modeled on the Activity Dashboard.
2. The initiator participant is a participant of the Activity Dashboard

**Post-Conditions:**

1. A dependency exists between the two states of two different activities.
2. All the participants on the Activity Dashboard can view the newly created dependency.

**Main Success Scenario:**

1. The initiator participant clicks on the button to create a new dependency.
2. The system asks to select the source activity and the destination activity.
3. After selection of source and destination activity, the system displays a dialogue box to select the state of the source and the destination activity and the type of the dependency to be created between them.
4. The initiator participant selects the state of source activity and the destination activity and the type of dependency to be established between the two states.
5. The system creates a new dependency of the required dependency type between the selected states of two activities.
6. All the participants on the Activity Dashboard can view the newly established dependency in the form of an arrow with dependency type written on it.
7. The system verifies the model for any inconsistencies introduced by this newly created dependency.
4.7.8 Use Case Id: UC8

**Use Case Name:** Change Activity States

**Initiator:** Participant via Google Wave Client

**Goal:** To enable participants involved in disaster response management to execute activities by changing their states.

**Pre-Conditions:**

1. The activities whose states have to be changed are visible to the initiator participant in the Activity Dashboard.

**Post-Conditions:**

1. The current states of selected activities are changed and visible to all the participants in the Activity Dashboard.

**Main Success Scenario:**

1. The initiator participant clicks on the button to change activity states.
2. The system enables a drop down list on all the currently modeled activities to select their current states.
3. The initiator participant selects the desired current state of different activities.
4. The system detects if the states selected in step 3 cause any violation of temporal dependencies.
5. If the system finds dependency violations, then depending on the strategy for managing dependency violations (i.e. enforce or support), the system either does not allow changing the states of activities by displaying a warning message or highlights the dependency violations that may be caused by states changed of step 3.
6. In case of support strategy or if the state changes done in step 3 do not cause any dependency violation then changes in the current states of activities are stored by the system.
7. The system propagates the state changes of the shared activities to all the activity dashbaords.
8. All the participants on the Activity Dashboard can view the new current states of the activities.
4.7.9 Use Case Id: UC9

**Use Case Name:** Remove Dependency

**Initiator:** Participant via Google Wave Client

**Goal:** To enable a participant involved in disaster response management to remove an existing dependency between the two states of two different activities modeled on the Activity Dashboard, allowing change in the plans depending on the situation or removing any inconsistencies or dependency violations.

**Pre-Conditions:**

1. The dependency and the activities between which the dependency exists are visible to the initiator participant in the Activity Dashboard.

**Post-Conditions:**

1. The dependency is removed between the states of two activities and the change is visible to all the participants in the Activity Dashboard.

**Main Success Scenario:**

1. The initiator participant selects the dependency to be removed between the states of two activities and presses ‘Del’ key.
2. The system deletes all the data related to the selected dependency from the gadget state and removes the dependency (connecting two activities) from the Activity Dashboard.
3. The system verifies the model for inconsistencies and checks for violation of temporal dependencies.
4. The system propagates the change to all the participants of the Activity Dashboard.
5. All the participants on the Activity Dashboard can view the dependency removed from the model.
4.8 Main Sequence Diagrams

4.8.1 SD1: Create Activity
4.8.2 SD2: Update Activity List (Activity Wave Participants Changed)
4.8.3 SD3: Update Activity List (Model Wave Participants Changed)

```plaintext
GetParticipatsActivityWaves:

Check in Context if AttributeName statsWith "activityWaves" && endsWith "[participants"
{
    for all participants check
        if AttributeValue contains participant && activity is not modeled
            activityId = get ActivityId from context AttributeName
            activityName = get Name of activity from context
            activityWaveId = get ActivityWaveId from context AttributeName
            activityWaveListId = get ActivityWaveId from context AttributeName
            Save activityId[activityName, activityWaveId, activityWaveListId in List
        }
    return list;

Note:
For knowing if activity is modeled or not, we can check one of the keys we put in the
gadget state when an activity is created, e.g. here we check if gadget state contains the
key activity_A_name or not.
```
4.8.4 SD4: Update Activity List (On Model Creation)

GetParticipantsActivityWaves:

Check in Context if AttributeName startsWith "activity/" & endsWith "participants"
{
  for all participants check
    if attributeValue contains participant & activity is not modeled
      activityId = get ActivityId from context AttributeName
      activityName = get name of activity from context;
      activityWaveId = get ActivityWaveId from context AttributeName
      activityWaveId = get ActivityWaveId from context AttributeName
      Save activityId|activityName|activityWaveId|activityWaveId in List
  }
return list;

Note:
For knowing if activity is modeled or not, we can check any of the keys we put in the
gadget state when an activity is created, e.g. because we check if gadget state contains the
key activity_A.name or not.
4.8.5 SD5: Update Activity Wave Title (On Re-Naming of Modeled Activities)

GetModelWaves(IntegratingActivityWaves):
Check in Content if (AttributeName startsWith "wave\CurrentView")
{
    if(AttributeValue == activityWaveId)
    {
        Save activityId|domain|modelWaveId|modelWaveletId|blipId from AttributeName in List
    }
}
return list;
4.8.6 SD6: Update Activity Wave Title (On Re-Naming of Un-Modeled Activities)

Get ModelWaves(IntegratingActivityWaves):
Check in Context if AttributeName startsWith "Wave (Current/View)"
{
  $AttributeValue = activityWaved"
  $Save activityWaved from ActivityListPanel
}
return list;

UpdateActivityNameInActivityListPanels:
Check in Context if AttributeName startsWith "modelWaveId" & endsWith "Participant"
{
  modelWaveId = Get Model Wave Id from Contact AttributeName
  $modelWaveList does not contain modelWaveId // Activity not modeled
  $ update activity name in the value corresponding to the key starting with "ActivityList_"
}
4.9 Tools & Technologies Used

This section will summarize the tools & technologies used to develop the system and the motivation behind them, where necessary.

4.9.1 Google Web Toolkit (GWT)

| Short Description | One of the end-user requirements is that the system should be web-based. Google Web Toolkit (GWT) is an open source and completely free, AJAX based powerful toolkit from Google for developing complex web applications. Providing many UI components, the main power of GWT lies in the fact that the developers don’t have to deal with writing complex Javascript code but rather the all the code can be written in pure object-oriented fashion using Java language and after compilation, it generates optimized Javascript and HTML code supported for all the main browsers. Another feature of GWT is the inclusion of development mode which saves the development time to a considerable amount. In development mode, the developer does not need to compile the code after making changes. Just save the modified java file and press refresh button in the browser and the changes will be reflected. GWT plugin for Eclipse makes development and deployment even more convenient. |
| Version Used | 2.0.4 |
| URL | [http://code.google.com/webtoolkit/overview.html](http://code.google.com/webtoolkit/overview.html) |
| Modifications Made | None |

4.9.2 GWT-Connectors

| Short Description | Library providing dynamic connector capabilities to GWT applications. This library is used in this project to model temporal dependencies between the states of activities. |
| Version Used | 1.7.0 |
| Modifications Made | - Added functionality to receive events when an activity is dragged/moved. So rather than saving the complete model, we only save the new position of those activities and connectors which are being dragged/moved.  
- Added functionality to show the type of dependency (precedes, meets etc) as part of the connector.  
- Added functionality enabling to change the color of the connector in order to show dependency violations, warnings etc.  
- Added functionality to delete the connector from the model using ‘Delete’ key.  
- Added functionality to deselect already selected connector when another connector is selected by the user to prevent multiple selection of connectors. |
| Problems Faced | This library internally uses gwt-dnd library to provide drag and drop functionality. In gwt-dnd library, any widget which is made draggable is no more editable. This was really problematic because activities had to be named and after adding the custom created activity widget (composed of an image and a textbox) to the diagram, the textbox was not editable anymore. |
| Solution to the Problem | a) Rather than creating a composite widget comprising of an image and a textbox and making it draggable, only the image was made draggable and onMouseMove event of image, textbox was also moved to the appropriate location by getting the coordinates of the image.  
 b) But after that, we were using the title of the activity wave as the activity name. So the above solution a) became irrelevant. So currently we are using an image with two labels (activity name and current state) to show an activity on the activity dashboard. Each time the title of the activity wave is changed, the robot puts the new activity name in the gadget state and thus the new activity name is reflected on the activity dashboard. |
4.9.3 GWT-DND (Drag & Drop)

<table>
<thead>
<tr>
<th><strong>Short Description</strong></th>
<th>Library providing drag-and-drop capabilities to GWT applications. This library is used internally by GWT-Connectors library and it enabled the activities and dependencies to be dragged and dropped.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version Used</strong></td>
<td>3.0.1</td>
</tr>
<tr>
<td><strong>License</strong></td>
<td>Apache License Version 2.0, January 2004</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.apache.org/licenses/">http://www.apache.org/licenses/</a></td>
</tr>
<tr>
<td><strong>Modifications Made</strong></td>
<td>None</td>
</tr>
</tbody>
</table>

4.9.4 GWT-Gadget API

<table>
<thead>
<tr>
<th><strong>Short Description</strong></th>
<th>The gadgets library for GWT simplifies gadget development by automatically generating a Gadget Specification from Java source, inserting a script in the specification much like a regular GWT project. After compiling the gadget with GWT, all files are put in place to publish the gadget. In this project, this library is used to develop three gadgets namely “Activity Dashboard”, “Activity Specification” and “Activity Participant View” enabling them to run as mini applications into Google Wave client.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version Used</strong></td>
<td>1.0.3</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://code.google.com/apis/gadgets/">http://code.google.com/apis/gadgets/</a></td>
</tr>
<tr>
<td><strong>License</strong></td>
<td>Apache License Version 2.0, January 2004</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.apache.org/licenses/">http://www.apache.org/licenses/</a></td>
</tr>
<tr>
<td><strong>Modifications Made</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Problems Faced</strong></td>
<td>1) Unstructured documentation. 2) How to convert the GWT application into a gadget. 3) Problems deploying the gadget into Google App Engine. 4) Problem finding the URL to add the gadget into Google Wave. 5) Gadget performing functionality but images not being displayed.</td>
</tr>
<tr>
<td><strong>Solution to the Problems</strong></td>
<td>Consult Tutorials &amp; Useful links Section</td>
</tr>
</tbody>
</table>
### 4.9.5 CobogWave-Gadget

<table>
<thead>
<tr>
<th><strong>Short Description</strong></th>
<th>This library is a wrapper for the Wave Gadget API to be able to program Gadgets with GWT using Java. Our three developed gadgets mainly use the classes from this library for storing &amp; loading the state of the activities and dependencies.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version Used</strong></td>
<td>1.2</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://code.google.com/p/cobogwave/">http://code.google.com/p/cobogwave/</a></td>
</tr>
</tbody>
</table>
| **License**           | Apache License Version 2.0, January 2004  
[http://www.apache.org/licenses/](http://www.apache.org/licenses/) |

### 4.9.6 Google Wave Robots API

| **Short Description** | This project uses Google Wave Robots API to implement the backend/automated functionality required by the system. This functionality is implemented by developing a robot (an automated participant on a wave). A robot has many capabilities like any other human participant which may include reading the contents of a wave in which it is participating, inserting/modifying/deleting the contents of the wave, adding or removing wave participants, creating new waves and blips etc. Google Wave Robots API is used to create and manipulate Robots. In this project, we have developed a robot “UProMan” using Google Wave Robots API to perform the following actions:  
- Inserting gadgets in a wave.  
- Read/Insert/Update/Delete entries in the gadget’s shared state.  
- Adding participants to the activity dashboards and activity waves.  
- Storing information in the data store.  
- Supporting the Verification of Model and Execution of activities. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version Used</strong></td>
<td>V2-20100802</td>
</tr>
</tbody>
</table>
| **License**           | Apache License Version 2.0, January 2004  
[http://www.apache.org/licenses/](http://www.apache.org/licenses/) |
| **Modifications Made**| None                                                                                                                                  |
### 4.9.7 Google App Engine SDK

<table>
<thead>
<tr>
<th>Short Description</th>
<th>Google App Engine is a cloud computing technology which enables to run web applications on Google’s infrastructure. Differing from other cloud based platforms offering infrastructure as a service, App Engine provides Platform as a service. From a list of services offered by the platform, we use Google App Engine in the context of our project for deployment of gadgets and robot and the data store service for persistence.</th>
</tr>
</thead>
</table>
4.10: Chapter Summary

This chapter has presented the implementation details of the system by describing different technologies used to develop and deploy the system. The functionality of the gadgets "Activity Dashboard", "Activity Specification" and "Activity Participant View" as well as the robot "UProMan" is discussed in detail. The chapter also presents the static and dynamic behavior of the system described using UML models and diagrams.
Chapter 5

Evaluation

The project is still on-going, so a thorough evaluation of the system cannot be presented at this point. We however, did a small exercise, with experts from business process domain divided into command centers and field teams. We gave them the scenario mentioned in section 2.3 and asked them to use our prototype to model the processes by creating activities and dependencies as well as sharing certain activities with participants of other activity dashboards. The prototype also supported them in execution of activities and verification of models for inconsistencies. They confirmed that the system is easy to use and provide much flexible collaboration with the support of Google Wave. It was also confirmed in the exercise that the system meets its requirements allowing simple modeling, execution, monitoring as well as the inter-organizational aspects of the disaster response processes. The added feature of verification of the model was considered very beneficial in the context of disaster process management, enabling people to focus on their main tasks rather than spending time searching the models for inconsistencies. One suggestion was to introduce filters by criteria in the Activity Dashboard to restrict the view in case of huge models with many activities and dependencies. Another good comment was to visually differentiate between organization’s own modeled activities and the activities shared by other organizations.

After discussions with people from public safety organizations in France, we got the idea that the terminology used in the system is aligned with the terminology of these organizations. It also came out in the discussion that there might be situations to which the responders have to react but there is no information technology support available (e.g. Fire in a forest). In these situations, the system can still be used by the command centers for planning and decision making. However, this is not limited to our system only.

The computational complexity for execution of an activity is \(O(N)\), where \(N\) is number of temporal dependencies, associated with the activity.

The performance and scalability of the system is inherited from Google Wave. Google Wave is highly scalable due to its distributed and federated architecture allowing any organization or individual to become a wave service provider and start collaborating at any time with other
organizations. The high performance of our system is ensured by the underlying Google Wave Federation protocol over XMPP, allowing near real time communication between wave servers.

The privacy issues are handled by the concept of “Model Wave” for each organization and “Activity Wave” corresponding to each activity. Only participants of a model wave and in turn the designated members of an organization can view the Activity dashboard and make updates to the models. Similarly information related to an activity is only visible to the participants who are added to the corresponding activity wave. This is the result of a very clean approach of only wavelet participants having access to the content of the wavelet.

With regard to security concerns, again the system leverages the security features of Google Wave. As an extension to XMPP core protocol, Google Wave federation protocol ensures that the connection must be secured using the TLS feature of XMPP.
Chapter 6

Conclusion & Future Work

Keeping in view the shortcomings and flaws of the current practices for coordination and collaboration between different public safety organizations as well as the inflexibility of current process management approaches, we have presented an approach for modeling, execution, monitoring and cross-organizational aspects of disaster response management. The web based system developed using this approach leverages existing internet standards and collaboration infrastructure (Google Wave). The system developed using Google Wave extension mechanisms (Gadget, Robot) provides seamless real-time communication and scalability by taking advantage of Google Wave Federation Protocol which provides support for interoperability among heterogeneous organizations. Currently, this system focuses on disaster response management but we hope that it may be used to manage unstructured processes in other domains.

The development work presented in the master thesis is based on the research done by Joern Franke (industry supervisor for this thesis) at SAP Labs France within the Public Security Research team.

My contribution in the overall project included software development based on this research, testing and documentation. 80% of time was dedicated to software development while 20% to research activities which mostly included evaluating different technologies to be used and how they suit in this context. The development effort includes 5 Java projects (3 for Gadgets, 1 for Robot and 1 for Activity Management Engine) in total comprising of approximately 8000 lines of code. The format, structure and contents of this Master thesis document have been reviewed by Vladimir Vlassov (Thesis examiner) whose continuous guidance helped the author to get this work done.

In the future, the system can be enhanced to provide more support for visualizing the information based on the context and usage. For example, people in the field require much simpler view than the people in the command centers.

The system currently supports only temporal dependencies but it can be enhanced to introduce other dependencies such as resource and spatial dependencies between the activities. A
thorough evaluation of the system can be done along with the end-users and experts from disaster management and other domains.

The system is being developed with the vision of integrating it with various products of SAP. So integrating it with SAP Net Weaver [19], SAP StreamWork [20] etc can also be part of the future work.
References

1. E.L. Quarantelli, *Catastrophes Are Different From Disasters: Some Implications for Crisis Planning and Managing Drawn From Katrina*, Disaster Research Center (DRC), University of Delaware, Delaware, 2005.


Appendix A

A.1: Examples of Recent Disasters

A.1.1 Haiti Earthquake: Happened in 2010, at a magnitude of 7.0 with more than 100,000 casualties.
A.1.2 Earthquake in Pakistan: Happened in 2005, at a magnitude of 7.6 with more than 100,000 casualties.
A.1.3 Flooding in Pakistan: Happened in 2010, affecting more than 20 million individuals with estimated damages of more than $45 billion. Worse than the combined damages of 2010 Haiti Earthquake, 2004 Tsunami and Pakistan Earthquake 2005.