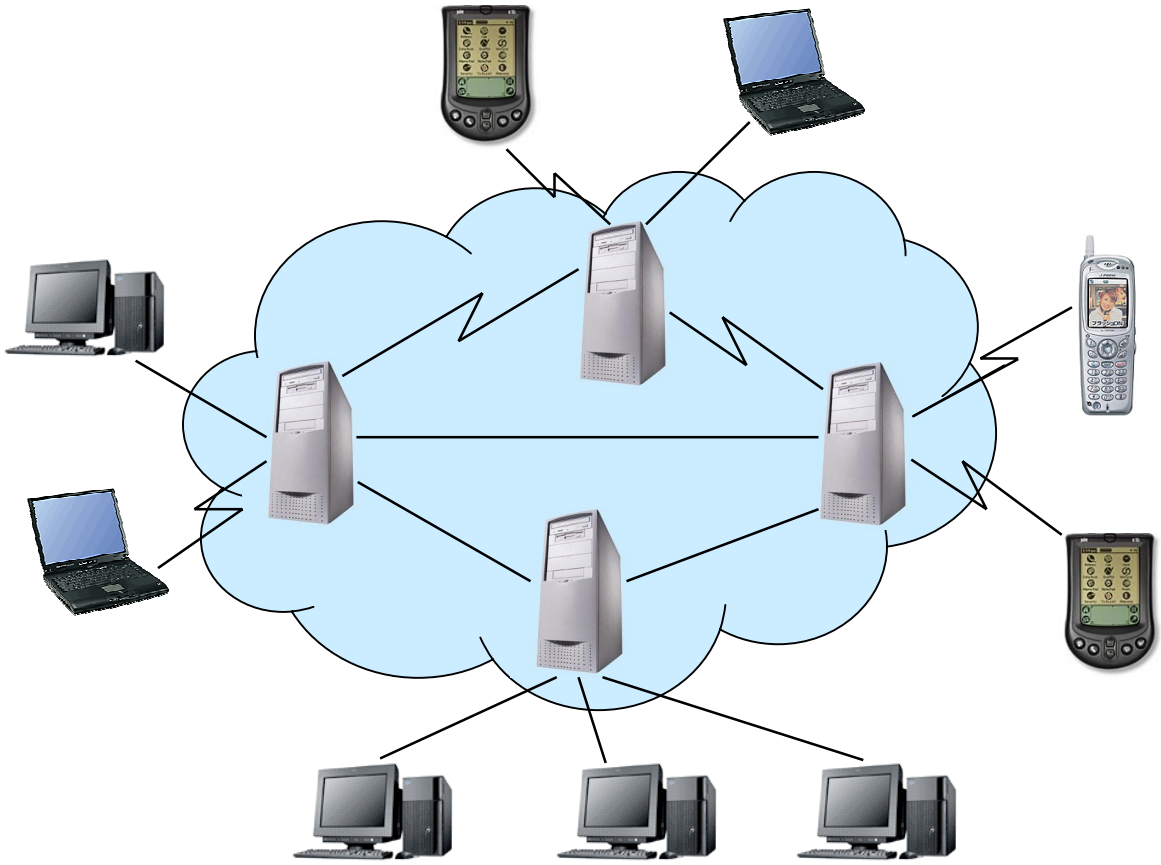
Context-Aware Rule-Based Data Distribution

Algorithms and Methods for Pervasive Computing

B.P.I. van der Poel
9659093

October 2002



TU Delft
Delft University of Technology

CAIP

Graduation commity

Drs. dr. L.J.M. Rothkrantz
Prof. dr. ir. E.J.H. Kerckhoffs
Prof. dr. H. Koppelaar (chairman)

PhD. I. Marsic (CAIP)
MSc. A.M. Krebs (CAIP)

van der Poel, B.P.I. (bvanderpoel@mail.com)

Master's thesis, September 2002
Context-Aware Rule-Based Data Distribution Algorithms and Methods for Pervasive Computing

Delft University of Technology, The Netherlands
Faculty of Information Technology and Systems
Data and Knowledge Engeering

Keywords: context awareness, dynamic data distribution, rule-base, publish-subscribe

ACKNOWLEDGEMENTS

This work has been done under the guidance of Mr. Marsic and Mr. Krebs at Centre for Advanced Information Processing (CAIP), Rutgers University, and Mr. Rothkrantz at the Delft University of Technology. I am thankful for their academic support they put into this research that helped me to complete my work.

I acknowledge the financial support of the Koninklijk Instituut van Ingenieurs (KIvI), the Universiteitsfonds Delft, the CvB of the Delft university of Technology, the Faculty ITS of the Delft University of Technology, and my parents. I thank you for the funding which has made it possible for me to work on my master thesis at CAIP.

The work at CAIP was supported in part by the US Army CECOM contract number DAAB07-02-C-P301 and a New Jersey Commission on Science and Technology excellence grant.

PROLOGUE

This thesis is the result of the research for my master's exam at CAIP at the Rutgers University, New Jersey. I worked at CAIP as an exchange graduate student for the period from September 2001 until August 2002. This work constitutes as the last part for my masters degree at the Delft University of Technology at the faculty of Information Technology and Systems at the chair of Data and Knowledge Engineering.

The research reported here is part of US Army CECOM Project. The project aims to develop hardware and software for future warfare. This research seeks to find means of efficient collaboration between commanders over wireless networks.

The title of this paper is "Context-Aware Rule-based Data Distribution Algorithms and Methods for Pervasive Computing". It addresses new research issues that will improve mobility support in networking. I hope the views in this paper inspire other researchers to continue this research.

Bart van der Poel
Oktober 16, 2002

ABSTRACT

The research field of pervasive computing is concerned with computing environments of diverse, possibly mobile computing units connected over wireless networks. The introduction of mobility in networking poses new challenges for the middleware with regard to accessibility and usability of data. To address these challenges this research focuses on context-aware data distribution algorithms and methods. Context-awareness provides environment dependent adaptation regarding relevance, timeliness and fidelity of data to the distribution.

In this thesis algorithms are introduced for client profiling and data selection. The client-profiling algorithm combines user-defined rules with contextual information to set up the selection rules. The data selection algorithm applies these rules to incoming data. Instead of Boolean decision the data selection algorithm maps the list of interested clients to priorities. Sending data to the clients is performed in order of priority.

For the implementation of the algorithms an expert system is embedded in the data distribution agent. This makes the subscription language for the client profiling highly expressive since conditions can be defined in conjunctive, disjunctive and negative forms. The subscription language is extended to support location aware conditions.

A collaborative system, called DISCIPLE, applying the proposed context-aware data distribution algorithms and methods demonstrates how the results of the research can be used in real applications. FLATSCAPE is a military collaborative application running on top of DISCIPLE that is developed for operational planning by commanders.

Grouping users with equal profiles is applied to improve scalability. Experimental results show that the agent performs well for a large number of data and users and consequently scalability of the agent is satisfactory.

TABLE OF CONTENTS

FIGURES

TABLES

LISTINGS

1 INTRODUCTION

This chapter will describe the problem of the research presented in this thesis. The setting of the problem, a description of the project the problem is part of, the definition of the problem, and applications of the problem are discussed in this order. The last section gives an overview of the remainder of this thesis.

1.1 Problem setting

Computing in this era has changed in a way that data needs to be accessible and exchangeable everywhere and all the time. Mobility of computing resources has become a prerequisite for users. Devices such as PDA's and cellular phones are used on a wide scale and are increasingly connected with each other. Mobility presents new challenges to networking because the accessibility and usability of data depends on the environment of the user. The accessibility is limited by the available bandwidth of the connection, and the usability in many cases depends on the location of the user. A solution to this problem is context-aware data distribution.

This new way of computing presents itself in many forms in everyday life. When driving in the car the driver will be interested in traffic information of the region he's driving in. A traffic application will track the driver's location with GPS and connects to a server to request the desired information. Mobile Internet on cell phones enables users to lookup news, weather, the stock market, etc. anywhere at any time. Waiters take orders on a handheld; the order is send to the kitchen that prepares the order, and to the cash register that calculates the check.

Figure 1-1: Three examples of applications of pervasive compting: traffic information system (left), mobile internet on cellphone (middle), and order system for waitering (right)

The field of research that is concerned with connecting large number of different (mobile) computing devices connected everywhere and all the time is called *Ubiquitous Computing* or *Pervasive Computing*. The pervasive computing vision assumes that future computing environments will compromise devices ranging from large computers to microscopic processing units, that will communicate over a wireless network. Mobility and dynamic reconfiguration will be inherent features in this environment [21]. We will discuss pervasive computing in more detail in section 2.1.

1.2 Context-aware data distribution

Within pervasive computing data distribution has become a very challenging task. Because of the diversity of the connected devices the data needs of each client will be different, as will the ways to present the data. Data distribution depends on the capabilities of the device, the location of the user, the quality of the connection, etc. In short, the distribution of data must be aware of the context of the client.

To achieve context-awareness data adaptation is necessary. Data distribution mechanisms match the contents of data with user profiles to decide to whom to forward the data. The adaptation of data comes down to adjusting the user profile to the environment, or context, of the client. This may result in *(i)* a selection of data for users or in *(ii)* a different representation of data. The first represents the interests of the user and the second the display capabilities. In the next subsection we discuss data adaptation in more detail.

1.2.1 Dimensions of data adaptation

In the research presented here, the focus is context-aware data distribution that addresses both network and location awareness issues. Data needs to be adapted to the current contextual state of the client. Important dimensions of data adaptation are relevance, fidelity, and timeliness, see Figure 1-2, where: (*i*) relevance is determined by user's interests and priorities; (*ii*) fidelity is dictated by computing platform's capabilities; and (*iii*) timeliness is determined by the requirements of the task [1].

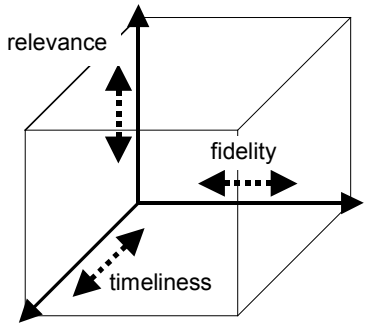


Figure 1-2: Dimensions of data adaptation for context awareness.

First of all the relevance of data has to be provided by the user and represent his interest in different types of data. This may depend on his work or the task he is performing. Fidelity of data is the level of detail that is to be applied during distribution of the data. An example of this may be that a device that displays the data textually doesn't need to receive graphical information of the data. The timeliness addresses timing constraints of data. Especially for real time applications this is an important dimension. Data with strict time constraint will have higher priority than loosely time constraint data.

1.3 Research description

The research that is topic of this thesis is part of a project called CECOM. The project is done for the US Army and aims to develop hardware and software for future warfare. The

problem the research described here focuses on, is the data distribution of a collaborative application for commanders.

1.3.1 Project description

The objective of the project is to develop and demonstrate a middleware framework for performance-optimized ubiquitous collaboration over heterogeneous networks. It will enable quality-of-service applications and support selective data distribution. The key component of the proposed framework is the collaboration bus, which is a middleware to support group communication in data-centric groupware. The research includes: a) algorithms and methods for dynamic monitoring of network capabilities, b) data transformation and adaptation policies for adjusting network traffic, and c) assessing and controlling the performance impact involving in broadcasting each message.

Some of the key elements of the research are the development of Data Adaptation Agents. These agents transform and change the data so that it can be used by different applications and hardware. The research will bring new capabilities for multi-user collaboration within reach.

1.3.2 Problem definition

The research problem this paper is concerned with is the design and implementation of a Data Adaptation Agent that is responsible for the distribution of data based on contextual information about clients. The agent makes decisions about which clients are interested in the data and replicates the data for each interested client. Subsequently the replicated data is sent to the clients.

Context awareness means that the decisions are at least partly based on the environment of the client. The agent has to construct profiles for the clients on which the decisions for distribution are based. For the selection of data an expert system will be embedded in the agent that consults a set of rules that represent the client profiles. A design with embedded expert system is used in order to define complex profiles.

1.3.2.1 Requirements

A couple of requirements exist for the design of a data distribution agent. The requirements mentioned here are the topics of research throughout this thesis. They are listed and discussed below.

- *Adaptive.* The decision of data distribution must be based on information about the environment. The agent should at least adapt to the following two environmental variables: the connection quality and the location of the client.
- *Dynamic*. A consequence of the support of mobile clients is a dramatically changing environment. Since the environment will change over time the adaptation should be dynamic. This makes monitoring of the environment necessary.
- *Profile definition by user*. Selection decisions must be based on user profiles. User-specific rules are to be generated from the profile. The user profiles are subject to adaptation, so the user should supply enough information for the server to interpret the profile in different environments.

- *Expressiveness*. To support expressive selection conditions a rule-based approach will be used. The rule-base enables the selection mechanism to make decisions deductively in rules constructed in first order logic.
- *Efficiency*. The data distribution must be as efficient as possible. The agent should be usable for real-time applications, for example for collaborative systems. Efficiency will be measured by the overhead the data distribution imposes.
- *Scalability*. It is important the system can support many users simultaneously without resulting in a high overhead. In collaborative environments many users may want to share some shared data set.
- *Robustness*. The whole data distribution system must be robust, so not all users are victims of a local failure in the network. This is an important prerequisite since the research is performed for the US Army.
- *Hierarchy*. The agent should support the hierarchy of the military. Rules exist that lay down the privileges and accessibility of data for commanders. These rules tell the system what data a commander may see or edit. The hierarchy is dynamic and consequently these rules should adapt to changes in the hierarchy.

1.3.2.2 Goals

The main goal of this research is the design and implementation of a data distribution agent that dynamically adapt to the contextual situation of the agent. To specify this, the following sub-goals are defined:

- *A model of adaptation and selection*. The agent needs to consult some model of user profiles to base selection decisions upon. The model is dynamic and should offer means for adaptation of these profiles. Important is that adaptation does not cause conflicting subscription rules.
- *Algorithm for adaptation*. The algorithm describes the steps the agent has to execute to adapt to its environment. New profiles should be calculated efficiently as a reaction of the monitoring of contextual inputs.
- *Approach for context-awareness.* A way to integrate context-awareness has to be designed. Extension of different kinds of contextual input should be supported. The environment must be monitored continuously.
- *A decision engine.* The decision engine selects to whom data should be forwarded. A rule-based approach for selection will be adopted. Important to the decision-making are the three requirements: expressiveness, efficiency, and scalability.
- *Framework to integrate these parts*. All the above-mentioned sub-goals should come together in a framework for context-aware data distribution. The framework should be flexible and easy to adopt. Flexibility makes redevelopment and replacement of parts possible.

We also aim at gaining more knowledge about how context-awareness can benefit pervasive computing and how to employ it. This knowledge will be useful to improve collaborative applications. We envision future applications for mobile clients with data presentation tailored to the specific needs of the client. This research makes this vision closer within reach.

The findings of this research result in a couple of deliverables that are described in this thesis. The deliverables are listed below:

- *Related work research*. First of all the literature about existing algorithms and systems of related research is to be overviewed. This will offer insight in what ways other researchers approached the problem and show the improvements that are to be made.
- *The design of algorithms for adaptive data distribution*. The design of adaptive data distribution algorithms is the core part of this research. The algorithms should integrate context awareness to adapt to.
- *Implementation of adaptive rule-base*. The designed algorithms are implemented as an adaptive rule-base. The rules should be adapted to the changes in the environment.
- *Development of data distribution agent*. The algorithms and rule-base have to lead to the development of the actual data distribution agent. The agent is to be programmed in Java and the rule-base is to be embedded in the agent code.
- *Evaluation of agent*. Once the agent has been developed it has to be tested. Measurements have to be performed to assess the performance of the agent.

1.3.2.3 Challenges

As mentioned above, data distribution for pervasive computing is a challenging task. Important challenges that will be addressed in this research are the following:

- *Efficiency, scalability, and expressiveness*. Three important requirements for data distribution algorithms are: *(i)* efficiency, the algorithm should classify and distribute the data fast, because it should support time critical data exchange; *(ii)* scalability, in order to support many users the algorithm must scale well; and *(iii)* expressiveness, precise selection of data is needed to avoid unnecessary network traffic. These requirements contradict each other, e.g. the more expressive the selection criteria are the worse efficiency will get. The context-awareness of the algorithm presented here puts the focus on expressiveness and scalability rather than efficiency. But efficiency will still be important for the algorithm to be useful.
- *Language for user profile definition*. The user must be able to define his preferences in an expressive way so complex profiles can be created. The more expressive the language for profile definition is the more precise the data selection for the users will be. However the language should be transparent for the user so it is simple and clear how to define profiles. In the design of the agent we must also make a decision about the form of profile definition. Should the user explicitly write rules or choose among templates? Or should the agent automatically create profiles based on user actions?
- *Monitoring of dynamic environment*. For mobile clients the environment will change drastically over time. Sensors are needed that measure the environmental variables constantly. The agent should monitor these sensors and react to changes in the measured values. A policy has to be chosen when and how these changes take place in the agent, actively or statically.
- *Accessibility of data*. For mobile clients with low bandwidth accessibility of data at a remote site will be a major problem. The data distribution agent must acknowledge this and find a way to deal with this problem. Adaptation of data selection to bandwidth is necessary.

- *Usability of data*. Roaming users want to receive information that is relevant to the place where they are. This means the data distribution has to be aware of the location of each client.
- *Timing of data*. The quality of the connection of mobile users will constantly change. The timing of data is thus relevant. At some points it will be wise to delay unimportant data to give more important data priority. The delayed data may be sent when the quality of the connection improves.
- *Recovery*. When a server in the network fails permanently the client has to reconnect to another server. The data and subscription information has to be recovered. The agent should offer a way to recover both. The data should be replicated over the servers and the subscription information must be re-sent by the subscriber.

1.3.2.4 Contributions and motivation

The contributions of this research to the current state of the research field Pervasive Computing are discussed in this subsection. Each contribution is motivated to express its importance to the research field.

- *Context-awareness.* Selection of data by the data distribution algorithm traditionally is only based on the users' interests. However the algorithm we have designed also takes the environment of the client into consideration. This environment is dynamic, which makes the algorithm more complex. The dynamics of the system makes adaptation of the otherwise static user profiles necessary. The environmental variables are used to conditionally select different user profiles in different contextual situations Context-awareness is motivated, as mentioned above, by the use of mobile devices in the network. It will enhance the accessibility and usability of data to the mobile clients.
- *Expressive selection language.* For the implementation of the data distribution algorithm an expert system is used, which makes the selection criteria more expressive than any existing algorithms. As a result more precise selections are possible. The expert system deductively maps many input variables, based on the contents of data and the context of clients, to its decisions for distribution. This is important for mobile users since a precise selection saves network resources by reduction of redundant data packets sent.
- *Priority sending.* The data distribution algorithm as a black box has data as input and a list of connections as output. In the algorithm presented in this paper the connections are mapped to priorities. The priority is used for the order of sending. The higher the priority the sooner the data will be sent over the connection. So priority sending addresses the timeliness dimension of Figure 1-2.

1.3.2.5 Approach

The problem will be approached in a couple of stages.

- The first stage will be literature research. This stage will provide the theoretical background and offer insight in related researches. After looking into the literature the current state of the research topic should be clear and global ideas for design will follow

- The ideas are used to make a global design that will lead to the first prototype. The prototype will demonstrate the features that will improve data distribution with contextual information. The ideas will be assessed and the good ones will be the ones that will be elaborated.
- The main phase of the research is the incremental development of the agent. Loops of redesign and implementation will stepwise improve the algorithms and methods.
- Evaluation is needed to assess the final agent design. The performance of the implemented software will be measured and also a theoretical analysis should give us qualitative information about the design.

1.3.3 Applications

An example of an application for such adaptive data distribution is operation planning for the military. Commanders are equipped with wireless communication devices and have to base operational plans on the received data about friendly and hostile units. The amount of data they are able to receive is limited by the bandwidth. The relevance of data will be different for each commander based on his task and the location of his unit. Fidelity of data depends on the rank of the commander; a low rank commander with an offensive task wants to be informed of the positions of the men in his unit and targets in the nearby area. The information a high rank commander will receive concerns positions of units instead of men. The timeliness of data about tanks will be more important to commanders than the timeliness of data about infantry, because tanks will move faster.

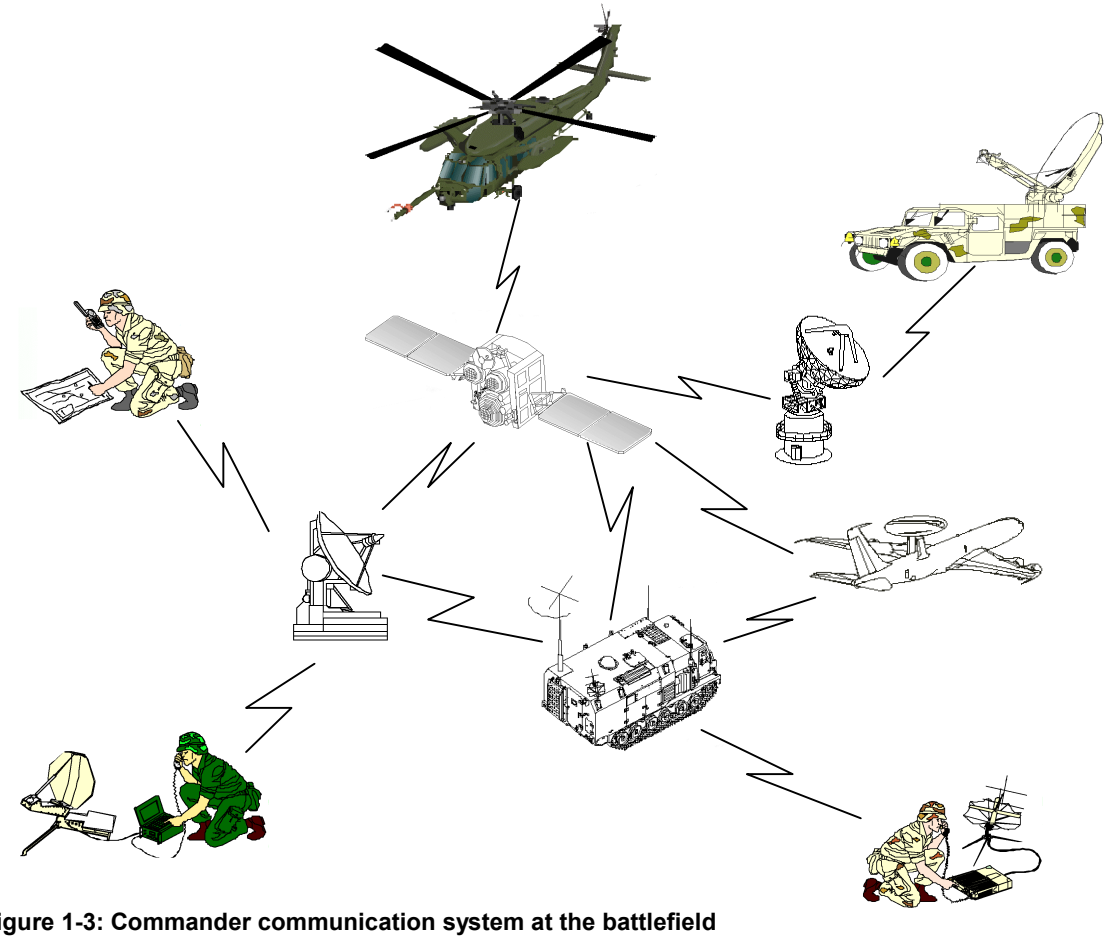


Figure 1-3: Commander communication system at the battlefield

1.4 Overview

The next chapter is dedicated to literature concerning the topics of this research. The text in this chapter introduces some theory and reviews related research. The related research is compared with the research presented in this paper. The third chapter will describe the architecture of the data distribution agent. The first sections cover the architecture with an increasing level of detail. A chapter describing the algorithms used by the agent follows this, which also covers the designed integration of the context awareness into the agent. Rule-base reasoning is topic of the next chapter, describing how the agent uses the rule-base to make decisions for distribution. Implementation issues are discussed in the fifth chapter. The subscription language and the replication code are issues that we will take a look at. The last section of this chapter shows the data-model. We take a look at the application of the agent in the sixth chapter. A collaborative system called DISCIPLE is discussed. The agent is integrated in this system. Also an application running on top of this system called FLATSCAPE is introduced. In the evaluation chapter some experimental results of measurements are shown and the algorithms used are analyzed. In the eighth chapter some topic for further research and enhancements of the current agent are proposed. The paper is finished with a chapter dedicated to conclusions.

2 LITERATURE REVIEW

The goal of literature research is to gain insight in the theory and terminology of the research topic. We will also take a look at some other research projects that are related to the one presented in this thesis. The related works are compared with our research to demonstrate the benefits of our research.

This chapter reviews literature on the main topic of our research. The topics are divided into three sections: the first pervasive computing, the second publish/subscribe middleware, the third context awareness, and the last rule-based notification service. The second topic describes the communication model between client and server that we will use. The section about context awareness focuses on network awareness and location awareness. The terminology introduced in this chapter is used throughout this paper.

2.1 Pervasive computing

Pervasive computing is the research field that is concerned with connecting many, diverse devices in a wireless network. The devices share and exchange data with each other but have complete different characteristics. Display, connectivity, computing power, and battery power are just a few such characteristics that will change for each device.

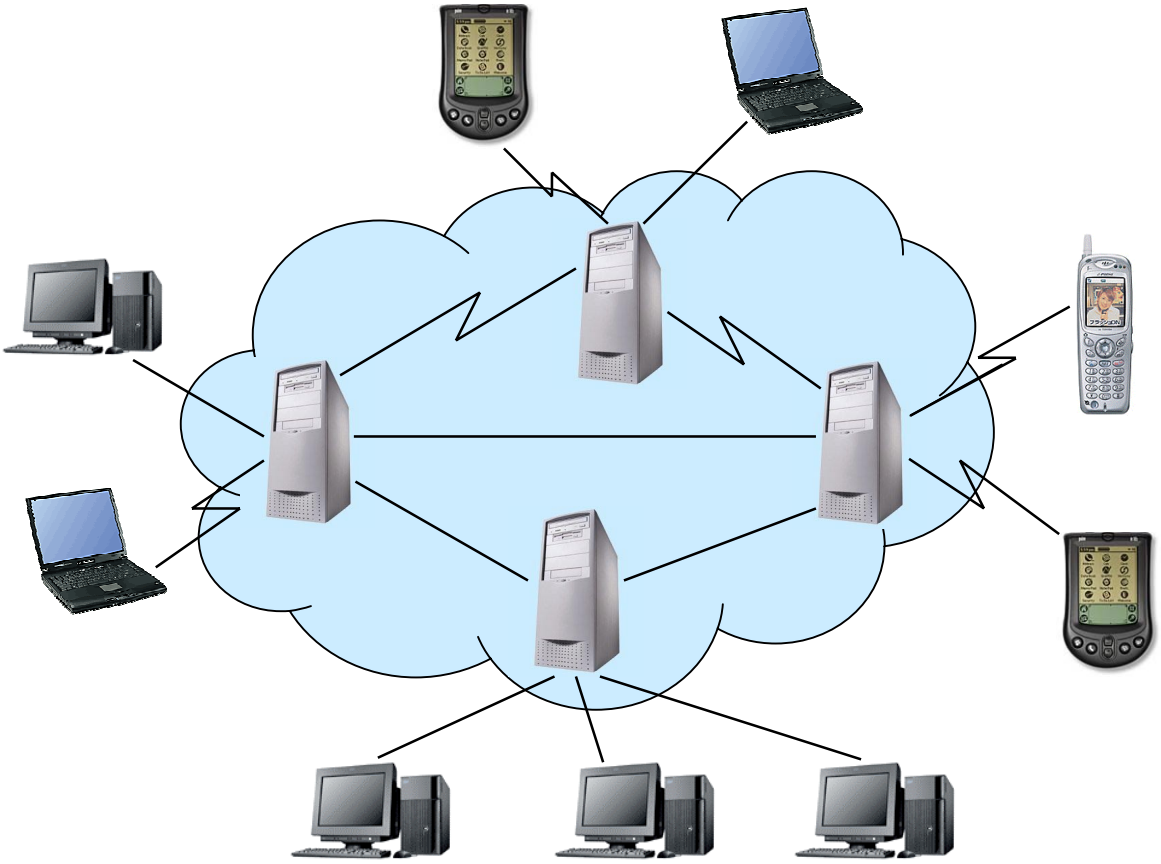


Figure 2-1: Pervasive computing, a network environment with many diverse devices

To illustrate how a pervasive environment works take a look at Figure 2-1. At the core of the environment is a network with servers and routers that connect the devices. The connections in the network may be either wired or wireless. The devices connect to the closest server, also wired or wireless. Possible devices are workstations, laptops, PDA's or cell phones. Each device should have an accurate representation of the shared data repository.

2.1.1 Related work

The Cooltown project of HP Labs is a good example of pervasive computing. It aims at supporting nomadic users with location aware information about the physical world. The research divides up physical entities in three categories: people, places, and things [23]. The people are the ones we communicate with, the places are the physical places we visit and the things are the devices we use. Entities in each category get a web-presence so the user can access them over the Internet.

Cooltown presents an interesting counterpoint to many other ubiquitous computing systems by placing the user in the control loop. It involves the user in the resource discovery [24]. This means it is uses pull-technology; the user pulls the data. The agent that we will design will use push-technology, removing the user from the control loop. This is a more challenging task because in this situation the agent has to make decisions about the information supply to clients.

2.2 Publish/subscribe middleware

In this section we review some literature on the publish/subscribe paradigm. First the theory and terminology of the paradigm are discussed. Next we take a look at some existing publish/subscribe systems.

2.2.1 Theory and terminology

The communication model used in the research is the publish/subscribe model. This model lets producers of data and consumers thereof communicate asynchronously. The producers are called the *publisher*, and the consumers the *subscribers*. The publisher pushes data, called *events*, to the network without any knowledge about who will receive this data. To let the network know what kind of information a subscriber wants to receive, he has to set up the conditions that the data must meet, in other words he has to subscribe. When the network receives data from the publisher that meets the conditions, the *subscription*, it forwards the data to the subscriber. The forwarded data is called an *event notification*. See Figure 2-2.

Publish/subscribe systems can roughly be divided in two groups: *subject-based* systems and *content-based* systems. The systems select event notifications for subscribers by subject respectively by content. This research is focusing on the content-based systems. Selection in these systems is more flexible and more precise, because subscribers can apply many-dimensional criteria instead of choosing between pre-defined groups.

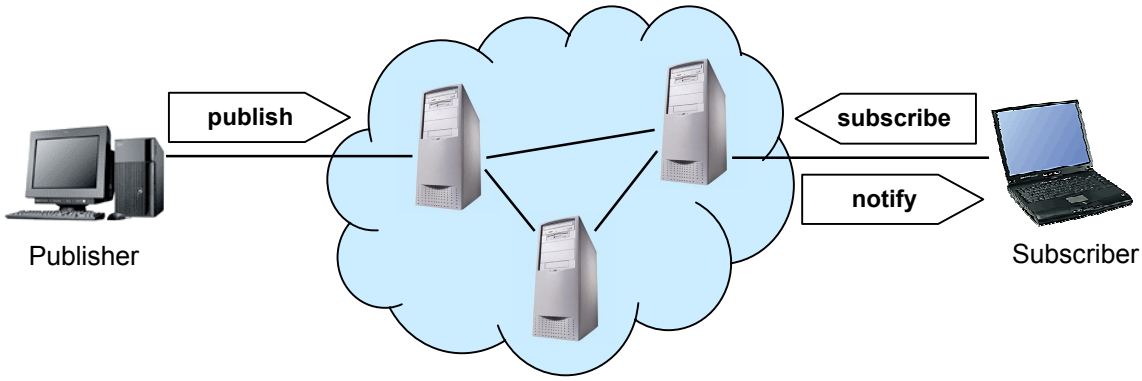


Figure 2-2: Publish/subscribe model

The goal of the use of subscriptions is twofold: *(i)* preservation of resources, and *(ii)* implementation of user preferences. The first goal prevents redundant event replication, and consequently unnecessary network traffic. The second assures the user is only bothered with events he is interested in. Care must be taken to ensure that the algorithms used to support filtering do not cause undue burden on distributed system resources [12]. The increased computational overhead of the filtering algorithm should be worth the improved performance of the system.

2.2.2 Related work

The best-known content-based publish/subscribe systems are GRYPHON [2], SIENA [3], ELVIN [4], [5], and KERYX [6]. The selection algorithm for GRYPHON emphasizes efficiency and scalability. It uses matching trees, which make its time-complexity sub-linear with the number of subscriptions. A drawback of this algorithm is the limited expressiveness: subscriptions can only be defined with conjunctions. In [7], a similar approach is suggested, using Binary Decision Diagrams, with a richer language for subscriptions. Expressiveness and scalability are the main interests for the SIENA selection algorithm. For this it uses covering relations, which are partial orderings with respect to subsumption. Though the expressiveness of subscriptions is still limited to conjunctive patterns. ELVIN uses the most expressive of the above-mentioned algorithm. It supports first order logic patterns, and regular expressions for selecting events. The algorithm in the KERYX system is expressive, but not very efficient. It uses a LISP-like filtering language.

The research in [10] and [11] both investigate the use of publish/subscribe for mobile nodes in a network, but concentrate on the network organization instead of the event selection. However both papers recognize that the publish/subscribe paradigm is a good candidate for mobile computing, because it fully de-couples event generators from receivers. However current publish/subscribe systems do not account for the possibility of dynamically reconfiguration [10]. Development of mobile devices demands publish/subscribe-systems not only to select events with content-based attributes, but also to adapt to the client's environment. Attributes that define computing capabilities of devices or the quality of the connection or location-dependent attributes should be considered as well. All systems mentioned above are not adaptive. [8] Proposes a stateful

approach that does take adaptation into account. The algorithm selects events with the conditions set by the user and the client state. The conditions are compiled at runtime. However the work in [8] does not have the expressive power for event selection comparable to the one showed in this thesis.

In contrast with the systems mentioned in this section, the selection algorithm in this thesis results into priorities, based on current network conditions. These priorities are used for the distribution of events. The advantage of a prioritized approach is that less relevant events can still be sent when the network is idle. Instead, all systems mentioned above make boolean decisions about whether an event has to be replicated. If an event is not sent to a client because of current conditions, it will never be sent, even when the conditions improve.

2.3 Context awareness

In [9] the following definition of context awareness is given:

> *Acquisition and utilization of any information that describes the setting of the users' activities, with emphasis on the physical attributes: time, place, people, physical artifacts, and computational objects.*

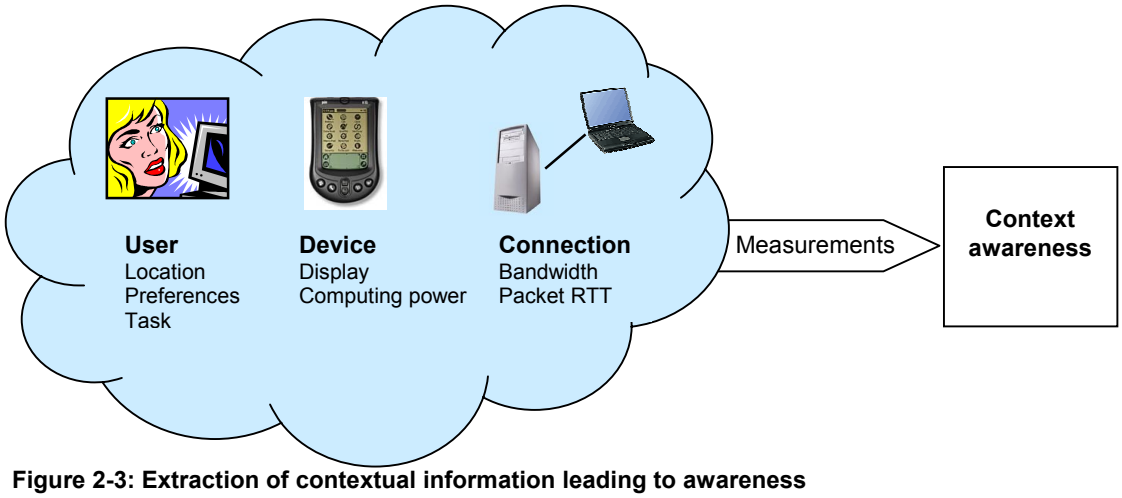


Figure 2-3: Extraction of contextual information leading to awareness

An illustration of how extraction of contextual information leads to awareness is shown in Figure 2-3. In the illustration different variables of users, devices and connections are measured. The vector of measurements is a representation of the current state of the environment. Interpretation of these values leads to awareness.

2.3.1 Theory and terminology

Context awareness means we have to adapt the data distribution to the environments of the clients. We distinguish two forms of adaptation: static adaptation and dynamic adaptation.

- *Static adaptation.* At design time we define the profiles for each possible context and during setting up the connection the agent looks at the context of the client and chooses the appropriate profile. This makes the adaptation process very efficient.
- *Dynamic adaptation.* The profile is generated on the fly based on the current context of the client. The user specifies just enough information so the agent can deduct the right profile. Consequently the agent has to monitor the environment continuously, because it may change at any moment. Each time the environment changes the profile has to be adapted.

We choose to design a data distribution agent that adapts dynamically. The reason is that the support of mobile clients results in drastically changing client environments. To correctly distribute data based on context we have to constantly adapt to the current context.

2.3.2 Related work

The definition of context awareness tells us that context aware information can have many different forms. In this research we have focused on two forms of context awareness: network-awareness (computational object) and location-awareness (place). We divide the related work review in these two parts.

2.3.2.1 Network awareness

The quality of network connectivity changes dramatically for mobile devices. To maintain an acceptable level of Quality of Service applications have to adapt to these changes. This is called network awareness. The parameters that play a key role in network awareness are available bandwidth and packet round trip time.

The limited capabilities and limited quality of network connection of PDA's are topics of the research in [16]. In order to avoid problems due to low bandwidth or even disconnection a pre-fetch of the data is performed so data can be manipulated locally. The application instructs the middleware what data needs to be cached. When the quality of the connection is sufficient, the data is replicated to the PDA. This approach is not possible in the setting of our problem. In order to pre-fetch data, it already has to be in the repository. However we have to deal with new incoming data and distribute it immediately.

In ODYSSEY [17] agility is marked as the main concern of adaptation. Sound adaptation decisions require accurate and timely knowledge of resource availability. ODYSSEY adapts the fidelity of data application-dependently. However the system does not select data for replication. Data selection, though, is the main interest of this research.

2.3.2.2 Location awareness

Data distribution for nomadic users is dependent on the physical environment. Selection based on the location of objects in the environment of the user or the location of the user self will help capturing only the useful information for the mobile user.

Location awareness is not a new feature in ubiquitous computing. Many traveler assistant systems, such as CYBERGUIDE [13], use location dependent information to support the traveler. However the data in these systems is static in contrast to the agent presented in this thesis, which uses the location-dependent information of dynamic data at the time it arrives at the server for distribution of the data. This makes the processing of location aware information performance-wise more complex. The application presented in [14] does provide dynamic information about the locations of users, however the locations of the other data, buildings, printers, etc. is still static.

The complexity of the problem at hand is described in [15] as follows: the middleware platform operating the applications must filter information for potentially millions of users, given their continuously changing location information, their profiles (or subscriptions), and the static and dynamic information about the environment. The system introduced in [15], combines traditional subscription language with SQL in order to create location-aware user profiles. Location information is stored in a database and accessed with SQL. In contrast the subscription language in this paper is extended with specific location-awareness tags.

2.4 Rule-base notification service

The subscriptions have a rule-based character. If the conditions of the subscription are satisfied then the event should be forwarded to the subscriber. A rule-base approach seems therefore obvious. The downside of a rule-base approach is the complexity of the evaluation of the rules. We must make sure it does not make the distribution inefficient.

2.4.1 Related work

As we have mentioned in the introduction we will use a rule-based approach to the event selection. By use of an expert system a more expressive means of defining subscriptions is obtained. The researchers of [20] apply a similar approach. In this publication a system is described that consults an expert system to map a set of sensor values to certain actions. The system waits until a predefined situation occurs to trigger an action. The situations and the actions are stated in the antecedents and the consequents, respectively, of the rules. An example of a rule is “if I’m logged into my workstation and I’m in my room, I want to be notified by email if my supervisor enters the lab”. More expressive means of event matching is also the reason for a rule-based approach for these researchers. The ECA Rule matching of [20] focuses on reactive applications instead of distributive applications that we are interested in.

The deterministic character of most new technologies in computing based on high-speed networking pose problems because they interact with non-deterministic environments [26]. In [26] the *adaptive processing* is mentioned as a main requirement for complex applications that depend on their environment in order to select the best strategy. A rule-base assigns dynamic parameters such as priority and deadline for each event. The rule-base approach is well suitable for real-time data distribution. However it does not consider semantic information of events but just environmental information. Another difference with our approach is that it uses multiple, decentralized agents while we have chosen for a centralized agent.

3 AGENT ARCHITECTURE

In this chapter the system architecture, middleware architecture and the agent architecture are covered. We start with a picture of the complete system and middleware, and work our way to a more detail picture of the parts and their interdependence of the data distribution agent. This will offer us a clear image of all components that are important for the data distribution.

3.1 System architecture

We start with the highest level of architecture that is the architecture of the system. The system is a network of clients and servers, like shown in Figure 2-1. On each of these clients and servers some middleware runs that enables that node in the network to communicate with the others. We look at the network of servers as one distributed server to which all clients subscribe.

The servers will communicate among each other in a peer-to-peer fashion. In this communication model each server is considered as equal and contains information about all its neighboring servers. The downside of this approach is an increase of overhead. However, the peer-to-peer model does not suffer from single-point failure, meaning that not all clients get disconnected when one server goes down. When a server goes down the clients of that server are disconnected, but for the other clients communication is possible after rerouting.

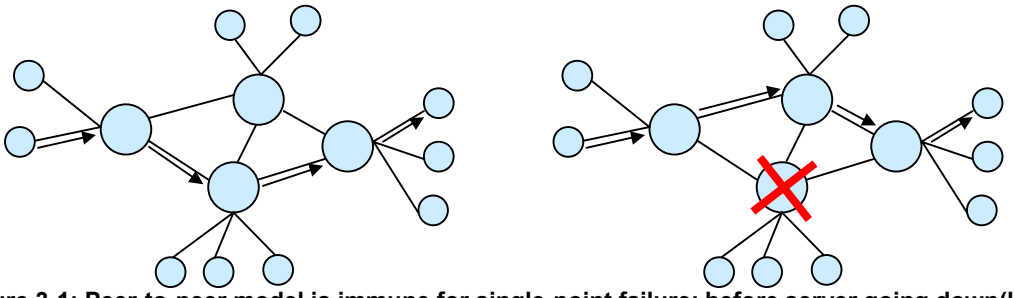


Figure 3-1: Peer-to-peer model is immune for single-point failure: before server going down(left), and after server going down (right)

When a server is down for some time the clients subscribed at that server should reconnect at a different server. Mobile clients can automatically look for the closest server and connect there.

The communication between clients and the distributed server takes place using the publish/subscription model as discussed in 2.2. The reason for this choice is that publishers and subscribers communicate asynchronously without being aware of each other's presence. This is beneficial in a network with mobile clients who may disconnect at runtime.

3.2 Middleware architecture

The software we are concerned with is middleware for collaborative data exchange. This layer is identical for the clients and the servers. The middleware is responsible for communication and data management for the client or server. Data distribution is one of these tasks. The complete architecture is shown in Figure 3-2.

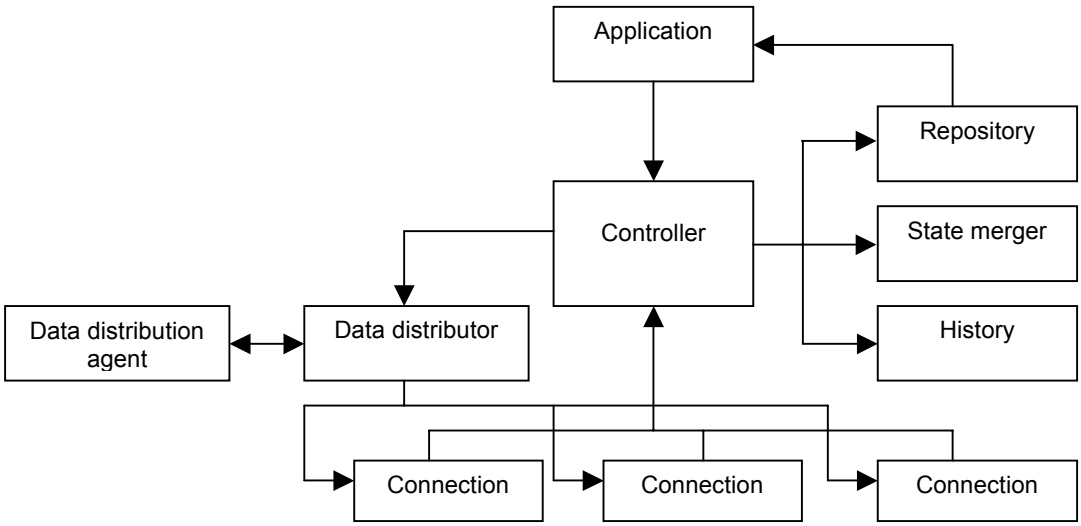


Figure 3-2: Architecture of middleware

3.2.1 Controller

At the heart of the architecture is the controller. It coordinates all data flows through the middleware. The controller maintains all connections with other clients or servers. The connections can be of a variety of types, for example TCP/IP or UDP/IP. The connections are threads that will continuously listen to events and incoming events are forwarded to the controller.

The controller interprets the events as commands and acts accordingly. Possible commands are login commands or data update commands. All events are saved in the repository. When an event is added to the repository it may reflect in the application to the user interface, depending on the type of command.

3.2.2 Data Distributor and Data Distribution Agent

The data distributor dispatches events generated by the client or received by the server. It consults the data distribution agent for the list of connections to forward the event to. The data distribution agent makes decisions about the selection of connections for each event based on client profiles. This is the component that we are interested in and which will be elaborated in the next sections.

The data distributor loops through the connection-list that is returned by the agent. For each connection it checks if the connection is not the one of the publisher. The publisher is the originator of the event and hence does not need to receive it again. The event is send to all the other connections. See the sample code in Listing 3-1.

Listing 3-1: Sample code for data distributor

```
for all connections in connection-list {
        if connection is not equal to publisher {
                Send event to connection
        }
}
```

3.2.3 State Merge and History

The State Merger updates the repository when a client logs in or reconnects. In the time that a client is disconnected the server may receive new events. When the client (re)connects it should be notified of these new events. For each of the events the clients need to be notified of the Data Distribution Agent should decide if the client is interested in it. It is also possible that the client creates new events during the time of disconnection. The State Merger also processes these events to the server repository.

The History component lists all events time-chronologically. When a client disconnects the component registers this time so it can easily deduct which events the client is aware of and which events happened during disconnection. This knowledge is used by the State Merger when the client (re)connects.

3.3 Data flows

The architecture of the middleware is based on the Client-Server model. On the client-side an application is running that communicates with the middleware on the server. Figure 3-3 shows the data flow through the middleware.

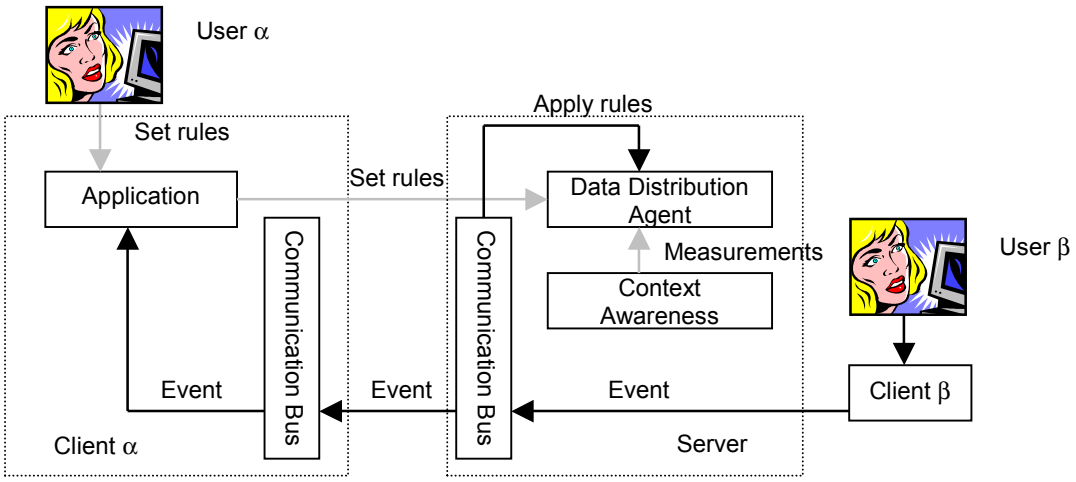


Figure 3-3: Data flow in communication middleware

The data flow illustration shows two separate data flows, color-coded with gray and black:

- *Subscription data flow (red)*. In order to start the communication subscriber $User_{\alpha}$ has to set up the rules that define what events he wants to receive. These rules are sent to the data distribution agent that, in combination with the measurement that follow form the context-awareness, interprets the user-defined rules.

- *Event selection data flow (blue)*. When the rules are set up for $User_{\alpha}$ the agent is ready to forward the desired events to the user. When $User_{\beta}$ publishes an event, the agent is consulted. Propagation of the event to $User_{\alpha}$ depends on the decision that follows after applying the rules.

For both data flows an algorithms should be designed to handle the tasks at the agent. So we need a subscription algorithm and an event selection algorithm. The first takes care of the data adaptation and the other matches events with subscriptions.

3.3.1 Interface of agent

We can easily deduct the interface from the architecture. As inputs to the agent are: *(i)* the user-defined rules, *(ii)* context-aware measurements, and *(iii)* the events. The only output of the agent is the decision to which users the event has to send. The agent of this research returns a mapping of connections to priorities. A list of connections is used so for the event replication a loop through this list is sufficient. By mapping the connections to priorities the sending of events to a client can be performed in order of importance. Time critical events could be sent first while the sending of low importance events or events with looser time constraints are postponed. In the next section we will discuss interface issues further.

3.4 Agent architecture

In this section we introduce the architectures of the data distribution agent. The architecture contains several components, which are illustrated with their interrelationships in
Figure 3-4. The figure shows the process flow after an event is triggered and how the different components interact with each other. The color-coding of the arrows corresponds to the color-coding used in Figure 3-3.

Data distribution agents use *replication algorithms* to make copies of an event for each interested client. The replication algorithm is executed by the Data Distributor, which sends the data to the selected connections. The other components are part of the actual Data Distribution Agent that carries out the selection of events for clients and adapts the data distribution.

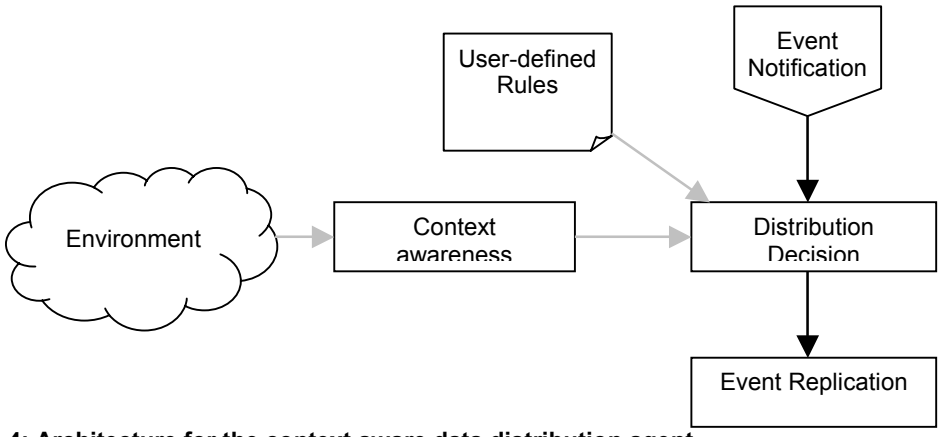


Figure 3-4: Architecture for the context aware data distribution agent

3.4.1 Event replication

Event Replication algorithms make copies for each of the clients that are interested in the event. The Data Distributor executes the algorithm, see also 3.2.2. Usually the Event Replication is based on a Boolean decision that is supplied by the Distribution Decision component. In this case the Data Distributor simple send the event directly to each of the connections in the list. The Event Replication algorithm proposed in this thesis extends the replication with the user of priorities. This enables us to influence the timing of sending of events.

For the architecture of the agent the prioritization means that the interface between the event selection component and the event replication component must enable the exchange of a mapping from connections to priorities.

3.4.2 Distribution Decision

Two possibilities exist where the selection of events for a client takes place:

- *At the server-side.* In this case the middleware at the server only replicate events to the clients that are interested.
- At the *client-side*. The software at the server replicates events to all the clients. The client middleware has to select of all incoming events if it accepts it or rejects it. This makes the replication at the server very efficient.

In our design the selection will take place at the server-side. When an event notification arrives at the agent the distribution decision component makes a selection to which connections the event is to be sent. This avoids unnecessary network traffic of redundant data packets that are send to uninterested clients.

The distribution decision will consult an expert system. The expert system contains rules that help the component to deduct a decision for each client. The users will supply the rules, as described in the next subsection. The fact-base of the expert system contains information about connections of clients, event notifications, and contexts of clients.

3.4.3 User-defined rules

In order to make decisions about the distribution, the users need to define a user profile that can be interpreted by the Distribution Decision component. The user defines rules that can be used to deduct a decision for distribution based on the context. This means that the profile needs to contain two kinds of information:

- *Semantic information*. This information describes what kind of contents the event should contain to be selected. This is the subscription of the client as described in 2.2.1. It is a logical expression that states the conditions that attributes of the event must meet.
- *Adaptation information.* This is information the agent needs to decide how to adapt the selection criteria in different contextual situations. This information describes the relevance of different types of data.

It is important that the user can simply define complex rules. A structural rule definition language, or subscription language, is eminent to achieve this. Several forms of profile definition are possible:

- *Fully user-defined*. Each user has to completely define his own rules. This is a time-consuming task for the users, but offers great flexibility and puts the user in control.
- *Templates*. Pre-defined rules are used as templates. The templates will be based on the role and/or task that is applicable to the user.
- *Automatically*. By monitoring the user the agent can create the profiles automatically. Completely automated profiling is probably not achievable. However in combination with templates or user-defined rules automated optimization of profiling is useful.

The agent we developed supports all three forms of profile definition. In FLATSCAPE, the system described in 7.2, we use templates that map roles and tasks to subscriptions.

3.4.4 Context awareness

The Context Awareness component extracts contextual information of clients by measuring environmental variables, such as bandwidth and location. This component is needed because both availability and usability of data change with the dynamics of the environment, with the introduction of mobility. In this thesis we will not discuss the methods used for the measurements, but we assume their existence.

Since we have already decided in 2.3.1 that we adapt to the context dynamically, the context needs to be monitored by threads. The threads perform the measurements periodically so the measured values always represent the current state of the environment.

The measured values will, in most cases, be numerical. The Context Awareness component needs to qualify these values for the Distribution Decision component to be able to reason with these values. Intervals need to be defined to assign values to classes with a qualitative value.

3.5 Decentralized agents vs. centralized agent

We distinguish two ways to organize the agent structure: decentralized or centralized, see Figure 3-5. A decentralized agent organization initiates an agent for each subscriber at the server when the subscriber connects. Each agent carries out the selection of events for one subscriber. The advantage of this approach is that the agent run simultaneously, so the selection of events per subscriber is processed in parallel.

When the server runs a centralized agent, one agent takes care of the event selection decisions of all the subscribers. As a result the overhead will increase, as the agent must keep track of the conditions of each subscriber. The benefit is that conditions that apply for several subscribers only need to be checked once.

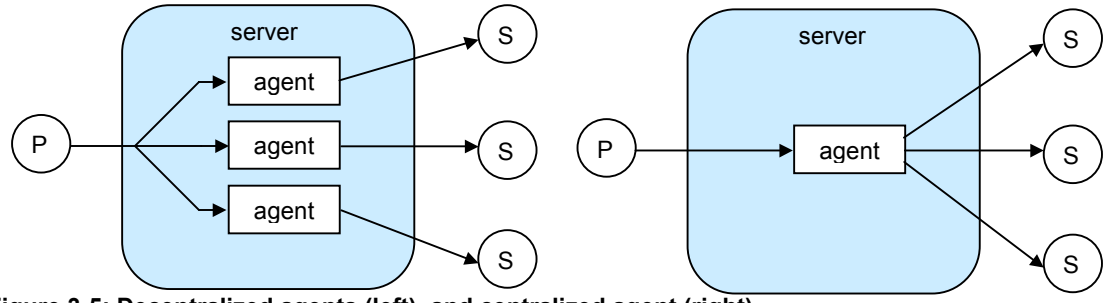


Figure 3-5: Decentralized agents (left), and centralized agent (right)

The centralized agent organization is preferred in our design for the reason that the agent will consult an expert system for the selection. When we choose for decentralized agents the temporal and spatial overhead will be unacceptably large, because then each agent must launch an expert system.

3.6 Resource manager

The middleware should also support resource management. A resource manager makes decisions about the distribution of resources. For example some total bandwidth is available through some wire. High rank users will be assigned a larger portion of this bandwidth maximum than low rank users. For military applications this is a interesting feature, because of the strict hierarchy.

The current middleware does not support resource management. This topic will not be mentioned further in this thesis, but is something for further research and development.

4 AGENT DESIGN

Designing a context-aware data distribution agent comes down to the following tasks:

- Modeling data distribution objects,
- Construction of an algorithm that handles the selection of events for all the clients,
- Integrating context awareness,
- Designing data distribution methods.

This chapter covers all these tasks, which will result in a design of the data distribution agent. First we take a look at the model of subscriptions and events. Section 4.2 deals with the construction of the data distribution algorithms. Two algorithms are described: one for the selection of events and one for adaptation. The next section of this chapter shows how to integrate awareness into the algorithm. How data distribution is used by methods is described in the last section. This section focuses on the replication algorithm.

4.1 Model for Data Distribution

First we need to construct a model to work with. Important requirements of the model are that it enables the agent to adapt the data distribution and that conditions used by several subscribers should only be evaluated once per event. The second requirement will improve performance of the event selection algorithm.

We have to model the subscriptions and the events. Users define the subscriptions before running the application to express what information they are interested in. This information is saved in a file that is read by the application during establishing the connection. The events are products by the users using the application at runtime. Every time a user produces data that need to be distributed this is send as an event.

4.1.1 Subscription

Subscriptions are logical expressions that select events for a client based on the contents of events. The expression states the conditions that must apply to attributes in order to be selected. Naturally, the attributes that appear in the subscription depend on the kind of events that need to be selected. An example of a subscription for selecting unit updates may be:

$$unittype \in \{infantry, armor, artillery\} \; AND \; size \leq 100 \qquad \textbf{Equation 4-1}$$

Equation 4-1 expresses what values the attributes *unittype* and *size* must have for the event to be selected.

Besides containing semantic information about the event the subscription should also have information how to adapt to changes in the environment. The actual subscription that is used by the event selection algorithm will be different for a user in every contextual situation.

4.1.1.1 Semantic information

Logical expressions are constructed by atomic expressions connected with logical operators. Atomic expressions are equations that compare attributes with values. To

define the semantic information of a subscription, we define the atoms and the logical form of the expression separately. This separated definition offers us the flexibility to evaluate atom and expressions differently. The atoms will be evaluated subscriber-independent and the expressions subscriber-dependent. In other words the Boolean value of an atom applies to all the subscribers and the Boolean value of a logical expression only applies to the subscribers that are subscribed to that expression.

Each atom is a tuple $\phi = (name_\phi, operator_\phi, value_\phi)$, and a subscription is a logical expression of these atoms. The name $name_\phi$ is the identifier of the attribute. The atom applies the operator $operator_\phi$ with the value $value_\phi$ to the attribute. When the subscription language supports not only expressions in conjunctive form, but also allow disjunctive logical connectives, the subscription should also describe the form of the logical expression the atoms should be interpreted in. We define a form function Γ, as the application of the logical form of the expression to a set of atoms. So $\Gamma(\Phi)$ is the complete logical expression of the subscription, of all atoms.

Table 4-1: Atom set of Equation 4-1

Id	Name	Operator	Value
ϕ_0	unittype	=	infantry
ϕ_1	unittype	=	armor
ϕ_2	unittype	=	artillery
ϕ_3	size	≤	100

For example, when we observe Equation 4-1 we distinguish four atoms, namely the ones listed in Table 4-1. A set of atoms is denoted by Φ. The form function Γ is equal to the following expression:

$$(\phi_0 \text{ OR } \phi_1 \text{ OR } \phi_2) \text{ AND} \phi_3 \qquad \textbf{Equation 4-2}$$

The application of the form function to the atom set of Table 4-1, $\Gamma(\Phi)$, results in the logical expression of Equation 4-1.

4.1.1.2 Adaptation information

The semantic information of a subscription describes the global interest of the subscriber. However some events selected with this expression may be more important to him than other events. When the contextual situation prescribes to decrease the amount of network traffic over the connection, e.g. in the case of low bandwidth, the expression should be more restrictive than the total semantic expression.

The agent needs some extra information about the relevance of each part of the semantic expression so it can adapt to the different contextual situations. We define the relevance vector of the subscription to express the relative interests of the subscriber. The relevance vector r for a connection specifies how the subscription should be interpreted under different environmental conditions. Each element in the vector, $r(\phi)$, assigns the importance of the corresponding atom in Φ for a client. The possible values for $r(\phi)$ should be in the discrete set ρ that contain the classifications of $r(\phi)$. It has to be a

discrete set because later on we have to fully define how each classification has to be prioritized.

Take for example the following relevance vector for the semantic expression of Table 4-1 and Equation 4-2:

$r = (3\ \ 2\ \ 1\ \ 3)$ **Equation 4-3**

In this relevance vector $\rho = \{0, 1, 2, 3\}$, where *0* represents the class of no relevance, and 1, 2, and 3 the classes of low, medium and high relevance, respectively. So there is a high relevance for information about units with infantry with less then 100 men. Events that match this profile should be replicated no matter what the context. Information about artillery is evidently of less interest and should only be sent when the context permits it, e.g. in the case of high bandwidth.

4.1.1.3 Prioritization

Events should be matched with the subscriptions and be assigned a priority for each subscriber. To achieve this each subscription is associated with a priority for the user. When the event matches the subscription the priority is returned.

When the server is started the number of priorities *N* for the agent is chosen. Let π be the discrete set of possible priority values. For each of the *N* values in π, a subscription is created for each user. So when *N* different priority levels are considered, we have *N* subscriptions for each subscriber, associated with the corresponding priorities. The assignment of priorities to subscriptions for a subscriber depends on the context.

Suppose we have the following set of priorities: $\pi = \{1, 2, 3\}$, with 1 as the highest priority and 3 as the lowest. The relevance vector of Equation 4-3 applies. We define three subscriptions for a subscriber as shown below:

Table 4-2: prioritized subscriptions

Subscription	Description	Priority
s_1	Events about units with infantry smaller than 100	1
s_2	Events about units with armor smaller than 100	2
s_3	Events about units with artillery smaller than 100	3

When one of these subscriptions match with an event than the associated priority is triggered for the subscriber for the event. So if the event contains information about an armor unit smaller than 100 men, subscription s_2 matches and priority 2 is assigned to the event for the subscriber.

4.1.2 Events

The contents of data are captured by event notifications. An event notification *E* is a set of name value pairs, called attributes and denoted by $\varepsilon = (name_\varepsilon, value_\varepsilon)$. *E* defines the contents of the event. The more attributes the event contains the more accurate the

representation of the event will be and the more accurate the selection of event can take place.

Table 4-3 shows an example of an event that represents an event about a hostile unit with infantry of the size of 85 men.

Table 4-3: Example of event notification.

Name	Value
unittype	Infantry
affiliation	Hostile
size	85

4.2 Data distribution algorithms

In this section we discuss the algorithm used in the *Distribution Decision* component of Figure 3-4. Actually we distinguish two algorithms: *(i)* the subscription algorithm, and *(ii)* the event selection algorithm. The first algorithm establishes a connection and sets up the rules for the client. The event selection algorithm handles incoming events and distributes it to the subscribed clients.

4.2.1 Subscription algorithm

First we describe how clients subscribe to the server. At the server side a set Φ of logical atoms is defined of all the atoms that may appear in a subscription. A subscription is a logical expression of these atoms. The form of this logical expression is defined in a form function Γ. The atoms and form function are described in 4.1.1. The subscriber supplies this information.

The state diagram in Figure 4-1 shows the process of subscription. Each client subscribing to the server will follow these states. The process is context aware; it monitors environmental variables and adapts subscriptions accordingly.

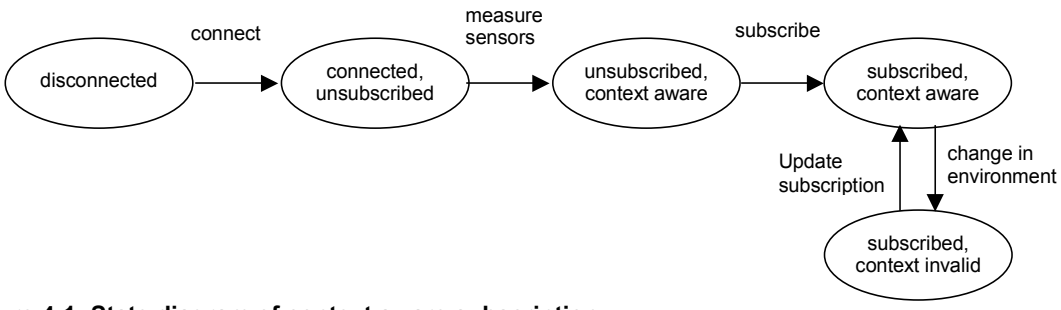


Figure 4-1: State diagram of context aware subscription.

For establishing a connection the connecting party has to send a relevance vector *r*. The relevance vector for a connection specifies how the subscription should be interpreted under different environmental conditions. The relevance vector is also described in 4.1.1.

The second step is to measure or predict environmental variables. This information makes the method context aware. If we use *M* different environmental variables the state of the connection is defined in a *M*-dimensional context space, called *C*. This space maps

the *M*-dimensional state to a context value *c*. The context value should be an element of the discrete set χ of all context classes, so $c \in \chi$. The context of the connection should be monitored all the time. When a change in the environmental state is registered, the context the subscription is based on is invalid, and the subscription has to be updated.

Now we know the relevance values for each atom and the context value of the state, we can assign subscriptions. For each priority value, a subscription is defined for each connection. So when *N* different priority levels are considered, we have *N* subscriptions for each client, associated with the corresponding priority. Let π be the discrete set of possible priority values. A priority matrix *P* is used to assign the priority to each atom in Φ, see Table 4-4. The rows of the matrix represent the elements of χ, and the columns the elements of ρ. Each entry in the matrix is the priority $p \in \pi$, for the particular context and relevance value. The relevance values and context values 1, 2, 3, represent a low, medium, high value respectively. The priority values 1, 2, 3, represent a high, medium, low value respectively. The *N* prioritized subscriptions can now be generated by first determining the subset of atoms for each priority level. Once we have these prioritized atom subset Γ is applied to the subset. The subset of atoms for priority level *p* and its prioritized subscription are given by the following two equations:

$$\Phi_p(c) = \{\phi | P(r(\phi), c) \geq p\} \quad \textbf{Equation 4-4}$$

$$s_p(c) = \Gamma(\Phi^p_{sub}(c)) \quad \textbf{Equation 4-5}$$

Atoms that are used in the subscription with a higher priority are also used in the subscriptions with a lower priority, so the latter is less restrictive. This is an important property, because it ensures the right order of subsumption of the prioritized subscriptions. This means that the low priority subscription selects at least all the events the high priority subscription selects.

$$s_{p=HIGH}(c) \preceq s_{p=LOW}(c) \quad \textbf{Equation 4-6}$$

Table 4-4: Example of priority matrix, with *N*=3.

		Relevance value *r*(ϕ)		
		1	2	3
Context value *c*	1	3	2	1
	2	2	1	1
	3	1	1	1

The assumption is made that disjunctive operators connect the atoms that differ between the prioritized subscriptions for a client. The reason behind this lies in the fact that conjunctive operators connect the atoms of different attributes and should all appear in each prioritized subscription. It is important that this assumption is kept because it ensures that the higher priority subscription is more restrictive. The relevance vector should enforce this property.

4.2.2 Event Selection algorithm

The algorithm for event selection is much simpler. New events trigger the process to decide if the event should be replicated to other clients. The event selection algorithm selects events for the clients based on their prioritized subscriptions and the contents of events. In this way the server will only replicate events for the interested clients, which saves unnecessary network traffic. The contents of data are captured by event notifications as described in 4.1.1.

The process first matches all the atoms with the notification name-value pairs. The function MATCH(ϕ,ε) returns true for atom ϕ and event attribute ε if and only if the names fields are equal and the operator field of ϕ applied to the value field returns true. All atoms that do not match any notification name-value pair are false. We say that atom ϕ is matched when at least for one event attribute ε the MATCH function returned true.

$$\text{MATCH}(\phi,\varepsilon) = \begin{cases} true \Leftrightarrow name_{\varepsilon} = name_{\phi} \wedge operator_{\phi}(value_{\varepsilon}, value_{\phi}) \\ false, otherwise \end{cases} \quad \textbf{Equation 4-7}$$

Next we see if the logical expression of the application of Γ to the atoms of the prioritized subscription $s_p(c)$ results to true. If it does, we say that subscription $s_p(c)$ fires. We define the function EVALUATE($s_p(c)$), that returns priority p when $s_p(c)$ fires for event notification E and the null value ∅ when it does not fire.

$$\text{EVALUATE}(sp(c)) = \begin{cases} p, & if\ sp(c) = true \\ \varnothing, & else \end{cases} \quad \textbf{Equation 4-8}$$

When $s_p(c)$ fires, priority p is returned for the data object for the connection. More prioritized subscriptions may fire for a connection because of the property in Equation 4-6; in this case the value representing the highest priority is returned. The priorities for all interested clients are returned as a mapping from connection to priority, as discussed in 3.3.1.

$$p = MAX\left\{priority \middle| \forall p \in \pi \cdot priority = EVALUATE(S_p(c))\right\} \quad \textbf{Equation 4-9}$$

4.2.3 Example algorithms

To clarify the algorithms, consider the following example. First we will take a look at the subscription algorithm. Suppose the subscriber sends the subscription information to the server. The subscription information contains Φ, the relevance vector, and Γ. The subscription information is shown in Table 4-5 and Equation 4-10.

Table 4-5: Φ of example subscription information

Id	Name	Operator	Value	Relevance
ϕ_0	unittype	=	infantry	3
ϕ_1	unittype	=	armor	2
ϕ_2	unittype	=	artillery	1
ϕ_3	size	≤	100	3

$(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ **Equation 4-10**

Now the agent has to execute the subscription algorithm to this information. The agent will be looping through a piece of code that checks the context value and if necessary executes the algorithm. We define three values for the context: $\chi=\{1, 2, 3\}$, where 1 is the worst context value and 3 the best context value. So the lower the context-value the more restrictive the subscriptions should be.

Table 4-6: Results of subscription algorithm

Context	Priority matrix	Atom subsets	Subscriptions
1	Relevance value r(ϕ): 1 2 3 Context Value c: 1 → 3 2 1; 2 → 2 1 1; 3 → 1 1 1	$\Phi_3(1) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ $\Phi_2(1) = \{\phi_0, \phi_1, \phi_3\}$ $\Phi_1(1) = \{\phi_0, \phi_3\}$	$S_3(1)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ $S_2(1)=(\phi_0\ OR\ \phi_1\ )\ AND\ \phi_3$ $S_1(1)=\phi_0\ AND\ \phi_3$
2	Relevance value r(ϕ): 1 2 3 Context Value c: 1 → 3 2 1; 2 → 2 1 1; 3 → 1 1 1	$\Phi_3(2) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ $\Phi_2(2) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ $\Phi_1(2) = \{\phi_0, \phi_1, \phi_3\}$	$S_3(2)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ $S_2(2)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ $S_1(2)=(\phi_0\ OR\ \phi_1\ )\ AND\ \phi_3$
3	Relevance value r(ϕ): 1 2 3 Context Value c: 1 → 3 2 1; 2 → 2 1 1; 3 → 1 1 1	$\Phi_3(3) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ $\Phi_2(3) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ $\Phi_1(3) = \{\phi_0, \phi_1, \phi_2, \phi_3\}$	$S_3(3)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ $S_2(3)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$ $S_1(3)=(\phi_0\ OR\ \phi_1\ OR\ \phi_2)\ AND\ \phi_3$

The results of the subscription algorithm are summarized in Table 4-6. Suppose the context value is measured and qualified as 1. This situation is listed in the first row. The first line in the priority matrix is used. We look up the priority values of each atom ϕ_i. This is 1, 2, 3, 1, for ϕ_0, ϕ_1, ϕ_2, and ϕ_3 respectively. Sequentially we construct the atom subsets for each priority level, see the third column. The algorithm finishes with the application of the form function of Equation 4-10 to the atom subsets.

If the measured context values are qualified as 2 or 3, than the results will be as shown in the second and third row of the table. The atom subsets and subscriptions in these cases have similar results for different priorities. For example when the context value is 3 all the prioritized subscriptions are the same. Since the event selection algorithm chooses the highest priority in these cases, this results into a situation where all the data is either ignored (the events do not match the subscriptions) or selected with the highest priority. In this case the context is good enough, e.g. very high throughput, to send all interesting events directly.

When we compare between the rows we can conclude that the subscriptions of the lower context values is more restrictive. When the context value is low (context = 1) only information about infantry is send immediately, all other events are postponed. When the context value is in the middle (context = 2), information about infantry as well as events about armor units are sent directly. And when the context is high (context = 3) all selected events are sent at once.

When the prioritized subscriptions are constructed for the client, the agent is ready to select events for the client. The event selection algorithm is illustrated in Figure 4-2. The MATCH function is executed for all atom ϕ_i, and event attribute ε_j combinations, using Equation 4-7. If $\varepsilon_0 \vee \varepsilon_1$ is *true* for an atom then the atom is satisfied. Subsequently we use the evaluation function to determine which prioritized subscription fire. The highest priority of the fired subscriptions is the priority that is used for replication of the event to the client, as in Equation 4-9. This means in this example the resulting priority for the event is 2.

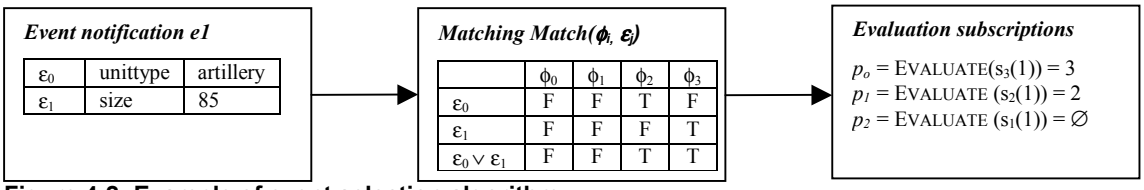


Figure 4-2: Example of event selection algorithm

This algorithm is applied to all the events that arrive at the agent. In Table 4-7 a list of events is shown that are evaluated by the event selection algorithm. The table shows how the priority responds to contextual changes and event contents. In the first two columns the ID's and contents of events are shown. The third column states the context value. The context changes because the environment is dynamic. In the next column are the subscriptions that fire and in the last column the priority that is assigned to the event.

Table 4-7: Events that are prioritized y the event selection algorithm

Event ID	(unittype, size)	Context	Subscription	Priority
e1	artillery, 85	2	$S_2(2)$	2
e2	artillery, 85	3	$S_1(3)$	1
e3	armor, 10	1	$S_2(1)$	2
e4	artillery, 32	1	$S_3(1)$	3
e5	artillery, 26	1	$S_3(1)$	3
e6	infantry, 78	1	$S_1(1)$	1
e7	armor, 16	2	$S_2(2)$	2

4.2.4 Algorithm issues

A couple of issues about the algorithms still have to be discussed. In this section we pay attention to these issues. First we want to make some comments about the separation of tasks between the two algorithms. Second, the priority matrix will be discussed in more detail.

4.2.4.1 Adaptation and selection

While constructing the algorithms a deliberate choice is made to let the data adaptation take place separate of the event selection. The reason for this is to decrease the overhead of the event selection algorithm. The efficiency of the event selection is very important to support real-time application. To have a separate algorithm for the adaptation, the subscriptions are adapted when a change in the environment takes place. When an event comes in, the current subscription only has to be evaluated by the event selection algorithm. Another benefit will be that subscribers using the same subscriptions can use

one subscription together. So these subscriptions only have to be evaluates once for all the subscribers that use it.

Another approach would be to check the context values during the event selection and integrate the contextual variables in the logical expression of the subscription. However \this will slow down the event selection, because it has to consult the context awareness module. When this approach is applied subscribers will in no case be able to use the same subscription because the context variables in the subscriptions are personal.

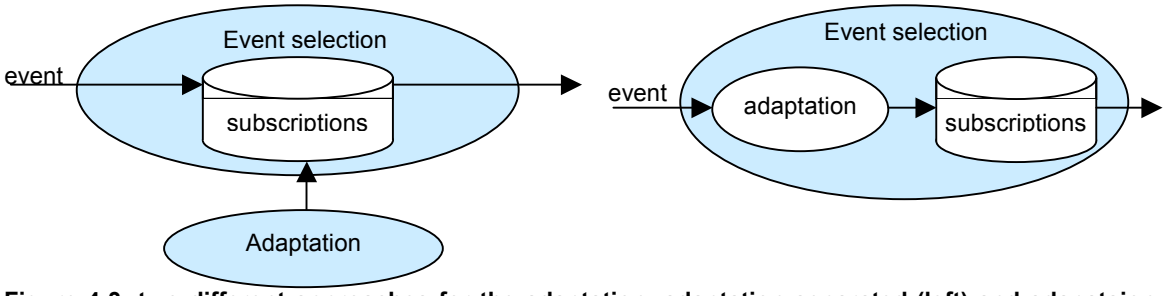


Figure 4-3: two different approaches for the adaptation: adaptation separated (left) and adapataion integrated (right)

The total overhead of the two algorithms will be different in the two approaches. It depends on the number of contextual changes and the workload of events that will come in. We expect that especially the workload of events will contribute to the total of the overhead. So our approach will save time because the event selection will be less time consuming.

4.2.4.2 Priority matrix

The priority matrix contains the information how to adapt the subscriptions to contextual changes. It represents the adaptation strategy. We distinguish two major strategies for adaptation:

- *Event delay*. This strategy delays the sending of events with a low relevance value. So when the context is low, e.g. low throughput, the events that are less important to the subscriber are sent later when either network traffic is low or throughput increases.
- *Event restriction*. When this strategy is applied the selection of events is more restrictive. If the context is low the events with low relevance are not sent at all. Only important events are sent.

The strategy is visible when we analyze the priority matrix. The event delay strategy results in a decreasing priority for atoms with a decreasing context value, but in all cases a priority is assigned. An example of such a priority matrix is shown in Table 4-8. Notice that the decrease of priority means an increase of the priority value (3 is the lowest priority in this case). The event restriction strategy needs to appoint a priority value representing exclusion to the events with low relevance. In this case we define the priority level –1 when an event is restricted. An example of such an priority matrix is shown in Table 4-9.

Table 4-8: Event delay strategy

		Relevance value r(ϕ)		
		1	2	3
Context value c	1	3	2	1
	2	2	1	1
	3	1	1	1

Table 4-9: Event restriction strategy

		Relevance value r(ϕ)		
		1	2	3
Context value c	1	-1	-1	1
	2	-1	2	1
	3	1	1	1

4.2.4.3 Adaptation to changing roles and tasks

The agent must support adaptation to the changing roles and tasks of the subscribers. For example a task of commander might change from offense to defense when some goal has been achieved. In the case of a new role or task the subscriber sends a new relevance vector that represents his new interests. The subscription algorithm is triggered because the context of the subscriber has changed and adjusts the subscriptions accordingly.

4.3 Integrating awareness

The context-awareness component in Figure 3-4 is discussed in this section. As mentioned earlier, we focus on two types of awareness: *network awareness* and *location awareness*. The methods used for the measurements to obtain the values of network parameters or the position of users and events is of no interest in this discussion. Of interest is how these values are used in the data distribution.

First of all we distinguish data-independent awareness and data-dependent awareness. Whether the awareness depends on data is crucial in the approach how to integrate the awareness. Data-independent awareness, such as network awareness, can be assessed and processed separately from the event selection. However when the awareness is data-dependent, like location-awareness, the awareness has to be integrated in the event selection. In this section we will first take a look at data-independent awareness. After this data-dependent awareness is covered.

4.3.1 Data-independent awareness

Data-independent awareness concerns information about the environment that does not change based on the data that is to be distributed. Examples are network awareness and awareness of device properties such as battery power. In the design of the agent data-dependent awareness is integrated in the subscription algorithm such as described in the previous section. The motivation for applying context awareness in the subscription algorithm instead of the event selection algorithm is performance. During execution of the event selection algorithm reduction of the overhead is very important, so that data distribution is efficient. The agent achieves this in two ways: *(i)* reduction of overhead per subscription, and *(ii)* decreasing the number of subscriptions. The first follows from the fact that the data-dependent awareness does not need to be considered in the event selection algorithm since the subscription algorithm already did that. Consequently the actual adaptation of the subscription may be done parallel to the event selection. The second follows from the fact that the clients with equal profiles can be grouped and processed together, because connection-dependencies, such as available bandwidth, are already taken into account by the subscription algorithm.

This environmental state of a client is derived of sensor values that measure characteristics of the environment. The sensor values are the inputs to the *M* different awareness modules A_i that classify the inputs and return the classified value a_i. For example, awareness module A_1 may be about network awareness and uses the inputs bandwidth and packet round trip time. The possible values for its output may be α_1={HIGH, MEDIUM, LOW}. When bandwidth is high and the variance of the round trip time is low than the value of a_1 is probably HIGH.

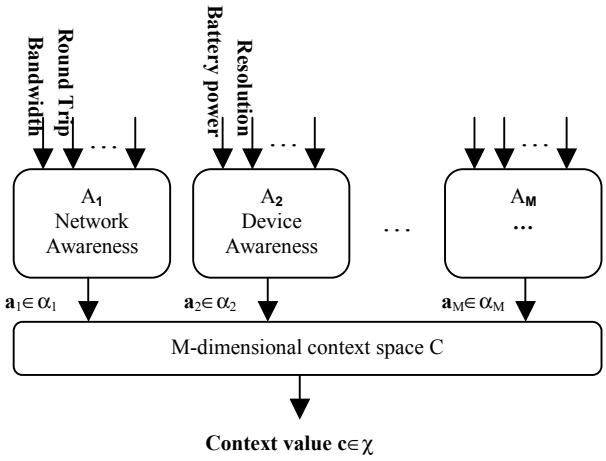


Figure 4-4: Outline of context awareness module.

All the values a_i, are used to find the context value *c* that represents the current state of the environment. This context value is found in the pre-defined *M*-dimensional context space *C*. This space maps each vector of awareness values to a context value. So the context value is given by the following formula.

$$c = C(\underline{a})$$ **Equation 4-11**

4.3.2 Data-dependant awareness

When data contains information that is used for the awareness the integration of data has to be processed differently than described above. An example of this kind of awareness is *location awareness*. Data-dependant awareness takes place in the event selection algorithm.

In order to use data-dependent awareness some preprocessing is needed before the event selection algorithm as described in 4.2.2 is executed. The data-dependent awareness preprocessing processes awareness information of users, previous event notifications, and the current event notification. The awareness conditions defined in the subscriptions are calculated and added to the current event notification. This context-aware event notification is passed to the Event Selection component. The process is illustrated in Figure 4-5

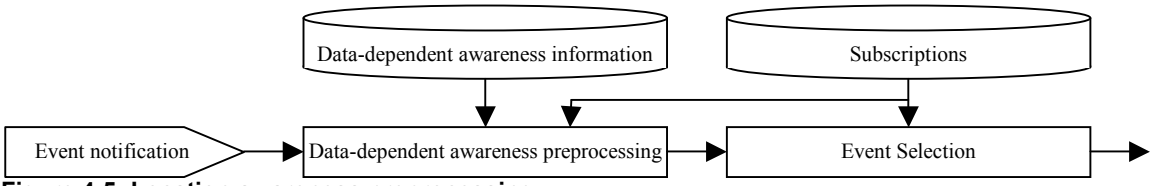


Figure 4-5: Location awareness preprocessing

Because the data-dependent awareness preprocessing is based on the awareness conditions in the subscription, no unnecessary processing is done. Only the desired awareness values are calculated.

As consequence of the fact that the data-dependent awareness is part of the event selection algorithm the subscription language need to support conditions based on the awareness. For example, a subscription language that supports location-awareness should be able to express distance conditions.

4.4 Data distribution methods

The methods describe how the prioritized events, that are the result of the distribution algorithms, are used in the agent. The method that is described below shows how the event replication takes place for the events that are assigned priorities.

4.4.1 Prioritized Event Replication

We have discussed how to prioritize events for each subscriber. However the question remains how to use the prioritized events. In this subsection the event replication method is discussed that answers this question. The method takes care of the ordering of events per subscriber by priority and is applied for each subscriber.

The number of priorities levels *N* is set when a server is started. For each subscriber *N* event buffers are created, one for each priority level. When an event is assigned the priority level *p* for the subscriber the event is placed in the event buffer assigned to *p*. Simultaneously events are read out on the buffers to be sent to the subscriber. The events in the highest priority buffer are read out first. When this buffer is empty the second highest buffer is read.

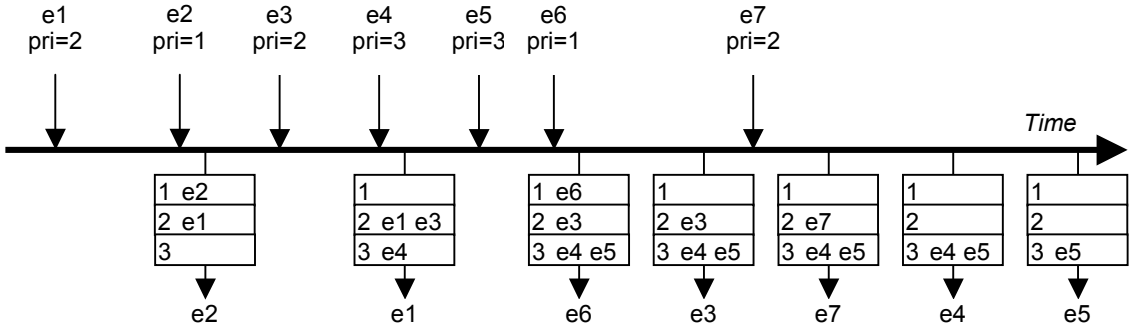


Figure 4-6: Example of prioritized event replication method with above time line incoming event and below timeline outgoing events

Figure 4-6 illustrates the method. As input events we take the ones listed in Table 4-7. Above the timeline we see the incoming events with their priorities, and below the timeline the outgoing events are shown with the event buffers for every priority level. In this example we have 3 priority levels, priority level 1 is the highest priority and priority level 3 the lowest.

In the case of the first outgoing event we see that two events, *e1* and *e2*, are buffered with priorities 2 and 1 respectively. Since event *e2* is buffered in the highest priority level queue that one is sent. We see that events with low priorities, *e4* and *e5*, are sent at the end when no other events are buffered.

One problem that is not tackled by this method is the dynamic behavior of the priorities. Since the priorities are assigned based on a certain context of the subscriber, the priorities of events that are buffered for some time should be adjusted when the context changes. However this problem only arises with very heavy workload.

4.4.2 Buffer reordering

A problem with the method described above is that when the context changes the priorities also change. Consequently the buffers have to be reordered. First the new subscriptions rules are set. Secondly the rules are applied to the events in the buffers to calculate the new priorities. At last we can order the priority buffers again.

Listing 4-1: Possible JAVA code for priority buffer re-ordering after a change in context

```
for (i=1; i<4; i++) {
        Vector currentBuffer = priorityBuffer[i];
        while (currentBuffer.hasElements())
                totalBuffer.addElement(currentBuffer.nextElement());
        priorityBuffer[i] = new Vector();
}
while (totalBuffer.hasMoreElements()) {
        Event currentEvent = totalBuffer.nextElement();
        int priority = applyRules(currentEvent);
        priorityBuffer[priority].addElement(currentEvent);
}
```

In Listing 4-1 possible JAVA code is shown that does take care of the re-ordering of the priority buffers. First all the buffered events are gathered in one buffer. The method loops through this buffer and calculates the priority of each event and puts the event in the right buffer.

5 RULE-BASED REASONING

When a server is started it initiates two distribution agents: one for the clients connected to the server, and one for the neighboring servers. We distinguish these two agents because the replication for clients and the replication for peers are executed by two different threads. An event notification triggers both agents to decide to which connections the update has to be replicated to.

In this chapter we take a closer look at the embedded rule-base of the agent. We will discuss the benefits of JESS first. Next the architecture of the rule-based agent is covered. In the third section the acquisition and usage of the knowledge in the rule-base is discussed.

5.1 Jess

A rule-based distribution agent matches incoming event notifications to the prioritized subscriptions. The data distribution agent utilizes an embedded expert system for the matching process. We use the JESS Expert system, because it *(i)* is easily integrated in the existing Java software and *(ii)* it uses the efficient Rete algorithm for rule matching [18].

JESS is a CLIPS-like expert system developed in JAVA. The scripts are interchangeable with CLIPS and the JAVA objects can easily be represented by facts in JESS. We can define templates for facts that are similar to the object structure. JESS even offers API calls that automatically generate such templates and convert objects to JESS facts.

In expert systems the rule-base is mostly static and the set of facts in the expert system is more dynamic. Though large part of the fact-base will be static as well. The Rete algorithm uses this knowledge to make inference efficient. Complexity of inference is measured in the number of rules in the rule-base (R), the number of facts (F), and the average number of facts in the rules (P). The Rete algorithm reduces the complexity to $O(RFP)$, by remembering which facts in the rules are already satisfied. The next time the inference engine is run, only the facts in the rules that did not yet satisfy need to be evaluated.

5.2 Embedding Jess

The agent utilizes an embedded expert system, namely JESS, for making decision about distribution. For the JAVA code of the agent to interact with the JESS code some interface is necessary. The different tasks of the agent are divided between the JAVA part and the JESS part. Figure 5-1 shows the components of the agent with an embedded expert system.

The upper part of Figure 5-1 shows the JESS components and the lower part shows the JAVA components of the agent. Since the expert system is embedded only the JAVA part receives input and returns output to the other parts of the system. The JESS part only communicates with the JAVA part of the agent.

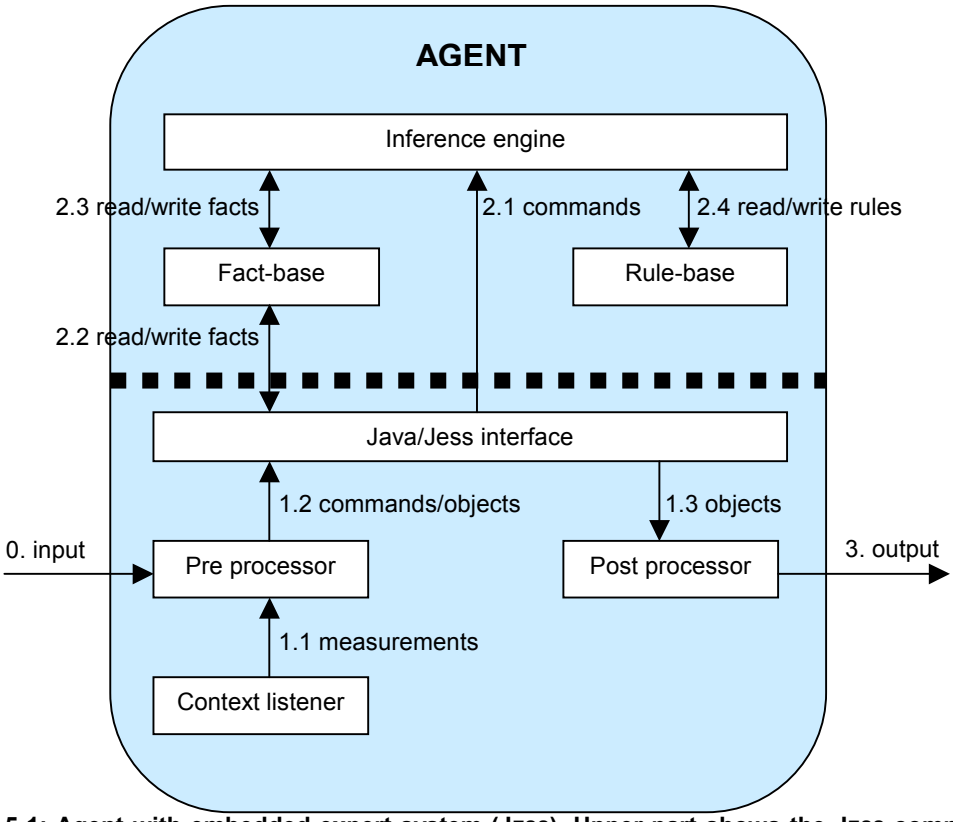


Figure 5-1: Agent with embedded expert system (JESS). Upper part shows the JESS components an lower part the JAVA components.

5.2.1 Data flow

All data flows, represented by the arrows, are numbered in Figure 5-1. We will discuss then here in order of their numbering.

0 *Input*. The input of the agent can be either the subscription information, when a subscriber established a connection, contextual information, or events. The pre-processor calls *setRules()* or *applyRules()* depending on the type of input. These methods are described in section 6.5.

1.1 *Measurements*. The measurements are the values of the contextual variables. These values are passed to the pre-processor. The variables are continuously measured.

1.2 *Command/object*. Depending on the input the pre processor sends commands to be executed by the inference engine or data in the form of objects. When *setRules* is called a command will be send with some data to set up a subscription rule for the connecting subscriber. If the *applyRules* is executed the event is send in the form of an object.

1.3 *Objects.* The outcome of the rule-based reasoning is some data that maps connections to priorities. In the JAVA part this will be some object.

2.1 *Commands*. The JAVA/JESS interface maps objects to facts and sends commands to the inference engine. The JAVA code initiates a *Rete* objects whose methods are API's for the expert system. The most used method is *run()* that makes the inference engine evaluate all the rules that are activated by the new facts. Another useful method is *executeCommand()* that parses a string parameter as a JESS command.

2.2 *Read/write facts.* The objects are mapped to facts by the JAVA/JESS interface, are saved in the fact-base. These are the new facts that will activate rules. When the *run()* method is executed the activated rules are evaluated. Facts are also read by the JAVA/JESS interface, namely the output.

2.3 *Read/write facts.* When the inference engine runs, the facts in fact-base are read to see if the patterns of rules are satisfied. The consequents of rules may assert new facts, which are written to the fact-base.

3 *Output.* The output is an object that maps connections to priorities. This data is read out of the fact-base.

5.2.2 Process flows

We distinguish two process flows through the architecture, corresponding to the two algorithms described in 4.2. The subscription process starts with the subscription information and relevance vector as input at the controller. The controller runs the inference engine and a new subscription rule is generated. This is explained in more detail in 5.4.1. This process can also be triggered by the Context Listener, which continuously checks the context changes. When the context is changed the subscription process is triggered.

When an event arrives at the controller the event selection process is initiated. The inference engine checks which subscription rules fire and return the connections and priorities of the rules that do. The JAVA/JESS interface passes the connections and the priorities to the post-processing component that only return the highest priorities of the connections that appear in the connection list.

5.3 Fact-base

The fact base contains short-term and long-term facts. The short-term facts change constantly, because of the dynamic adaptation. Since clients change subscriptions all the time the subscription facts are short-term facts. Long-term facts are not changed after they are asserted and can be considered as the memory of the expert system. Each event notification fact that has been asserted is saved permanently as a long-term fact for future reference.

5.3.1 Deftemplates

All the facts in the fact-base will be ordered facts. This means the form of the facts are defined as templates. A template describes an entity with attributes and thus is quite similar to objects in JAVA. The templates are defined at design time. To define a template we use the *deftemplate* command in JESS. In Listing 5-1 the definition of the templates of form function and atom are shown.

Listing 5-1: Templates definitions of form function and atom

```
(deftemplate form_function (slot id) (multislot nodes) (multislot branches)
       (multislot connections))
(deftemplate atom (slot id) (slot name) (slot operator) (slot value) (slot include)
       (slot satisfied))
```

The attributes in the template definitions are called slots of multislots. A slot can contain a single value and a multislot can contain a list of values. So the form function contains a single id, multiple nodes and branches (the form function is saved as a tree), and multiple connections.

5.3.2 Write facts

For the Java/Jess interface to write a fact in the fact-base it has to create a fact according to the template. First the template of the fact should be assigned and secondly the slots should be assigned values. The slots are identified by the slot-name. The values are objects with the actual value and a type of values. The value types are RU.ATOM (Do not confuse this ATOM with the atoms used throughout this paper. The ATOM mentioned here is just a JESS identifier.) for single slots and RU.LIST for multislots.

In Listing 5-2 a piece of code is shown to illustrate how the form function object can be parsed to a fact. The *setSlotValue* methods of the *Fact* object assign a value to a slot. For id the type RU.ATOM is used and for nodes and branches RU.LIST. With an assert commando the initiated fact is actually saved in the fact-base and can be used by the expert system.

Listing 5-2: Write form function fact

```
Fact f = null;
try {
        f = new Fact("form_function", engine);
        f.setSlotValue("id", new Value(id_, RU.ATOM));
        f.setSlotValue("nodes", new Value(nodes_, RU.LIST));
        f.setSlotValue("branches", new Value(branches_, RU.LIST));
}
catch (Exception e) {
        e.printStackTrace();
}
engine.assert(f);
```

5.3.3 Shadow facts

JESS offers methods to automatically generate templates and facts out of class definitions and objects. In this way JAVA objects can be represented as facts in JESS. The facts are called shadow facts and can be used in patterns for the rules. Lets take look at an example.

Listing 5-3: Generation of template for connection

```
Rete r = new Rete();
r.executeCommand("(defclass connection disciple.cbus.TcpIpConn)");
```

The JAVA code in Listing 5-3 first initiates a *Rete* object. The *Rete* object is the inference engine of the expert system. The engine calls the JESS command *defclass* that generates a template based on the class *disciple.cbus.TcpIpConn* and is named “connection”. This connection template contains attributes, called slots, which correspond to the properties of the *TcpIpConn* class.

Listing 5-4: Generation of shadow fact for connection

```
TcpIpConn conn = new TcpIpConn();
r.store("TCPIP", conn);
r.executeCommand("(definstance connection (fetch TCPIP) dynamic)");
```

Listing 5-4 shows the code that generates a shadow fact for the connection. First it initiates a new *TcpIpConn* object. A reference to this object is stored in the expert system, identified with "TCPIP". The *definstance* command creates a connection fact with the stored connection object reference.

At connection time the connection object of a subscriber is saved in Jess this way. We do not really need the fact, but to return a list of connection objects it needs the object references that are stored.

5.4 Rule-base

The rule-base is a set of rules that are checked when the inference engine is running. It is a dynamic process since new rules will be generated at run-time. In this section we take a look at how the knowledge in the rule-based is captured and how it is used for deduction.

5.4.1 Acquisition of knowledge

The acquisition of knowledge is done dynamically. Based on the subscription information the client submits, the subscription rules are generated. The relevance vector the client sends defines how the agent has to adapt the subscription of the client.

When we look at the rule sequencing in the inference engine after a *run()* command is executed we get the global diagram, shown in Figure 5-2. The bold labels at the transitions represent rules and labels in italics will be discussed in more detail below. When an arrow is labeled with ε, no rule is executed for the state transition. The ovals are different states of the rule-base.

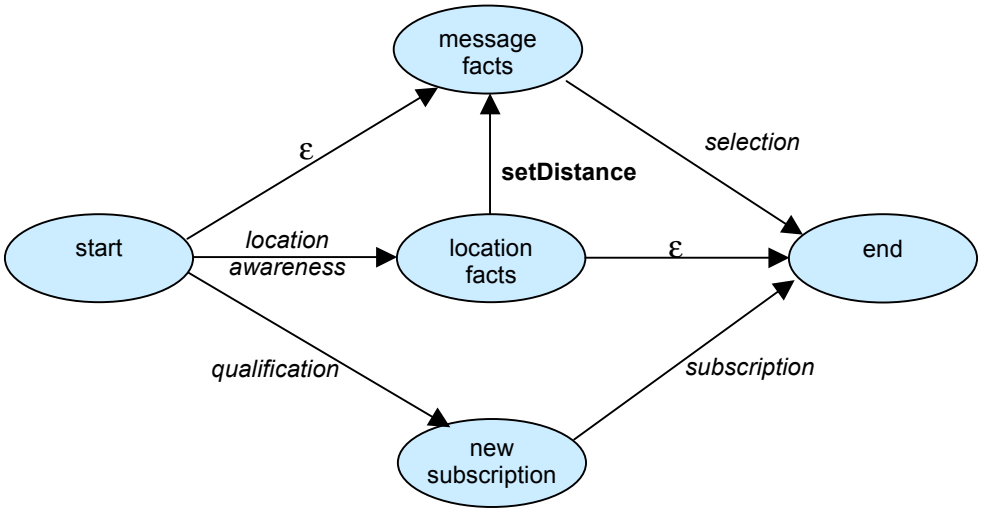


Figure 5-2: Global rule sequencing

The qualification and subscription transitions are executed when the subscription algorithm is executed and the selection transition when the event selection algorithm is executed. If a user or some data changes location the location awareness transition is triggered. This can be followed by the selection transition when a new distance is calculated.

5.4.1.1 Qualification rules

The first step in the generation of subscription rules is to qualify the network conditions and the relevance conditions. These qualifications are needed for determination of priorities, which subsequently result in a fact for a new subscription. This rule sequencing is depicted in Figure 5-3.

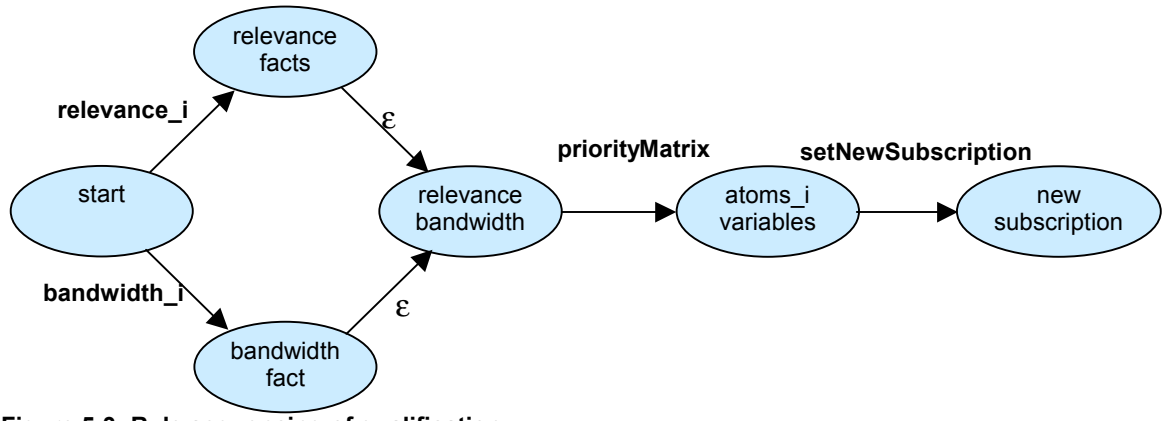


Figure 5-3: Rule sequencing of qualification

The qualification rules for relevance and bandwidth facts depend on the number of priorities, because we divide the spectrum of the values into this number of intervals. The number of priorities is set when the agent is initiated and at that time these rules are generated and saved in the rule-base. In Listing 5-5 an example of such a rule is demonstrated.

Listing 5-5: Qualification rule for network condition when bandwidth is between 0 and 1250 kbs.

```
(defrule bandwidth_0
    ?f <- (bandwidth (value ?value))
    (test (and (> ?value 0.0) (<= ?value 1250.0)))
    =>
    (assert (bandwidth_quality (value 0)))
    (retract ?f)
)
```

At this point we know the qualities of the network conditions and the relevance values. The priority matrix is now used to determine the priority for each atom as is explained in 4.2.1. The rule that represents the priority matrix bind variables for each priority level to the set of atoms that has to appear in the subscription of that priority. Since we use integer values for the qualifications we can calculate the priority for each atom in a straightforward manner to obtain a priority matrix with the form like Table 4-4, where priority 0 is the highest priority.

Listing 5-6: Rule representing the priority matrix

```
(defrule priorityMatrix
    "If we know the quality value of the relevance of the atom and the quality value
     of the bandwidth, then determine the priority level of the atom"
    (declare (salience 5))
    ?rq <- (relevance_quality (id ?r_id) (value ?r_value))
    ?co <- (bandwidth_quality (value ?b_value))
    (number_of_priorities (value ?priorities))
    =>
    (bind ?p (- ?priorities 1))
    (bind ?priority (min ?p (+ ?r_value ?b_value)))
    (while (> ?priority 0)
        (bind ?var_name (str-cat "$?atoms_" ?priority))
        (eval (str-cat "(bind " ?var_name " (create$ " ?r_id " " ?var_name "))"))
        (bind ?priority (- ?priority 1))
    )
    (retract ?rq)
)
```

Subsequently the inference engine creates *new_subscription* facts for each priority level, containing the atoms, the connection ID and the priority of the new subscription. In Listing 5-7 an example of such facts are shown.

Listing 5-7: Example of new subscriptions with 3 priority levels

```
(deftemplate new_subscription (atoms a0 a1) (connection conn0) (priority 0))
(deftemplate new_subscription (atoms a0 a1 a3) (connection conn0) (priority 1))
(deftemplate new subscription (atoms a0 a1 a2 a3) (connection conn0) (priority 2))
```

5.4.1.2 Subscription rules

The next step is to implement the new subscriptions. For this the decision-tree in Figure 6-1 has to be evaluated which is explained in more detail in 6.1.2. When the subscriber is already registered to another subscription with the same priority the subscriber is first removed from that incorrect subscription. Next we can add the subscriber to the correct subscription.

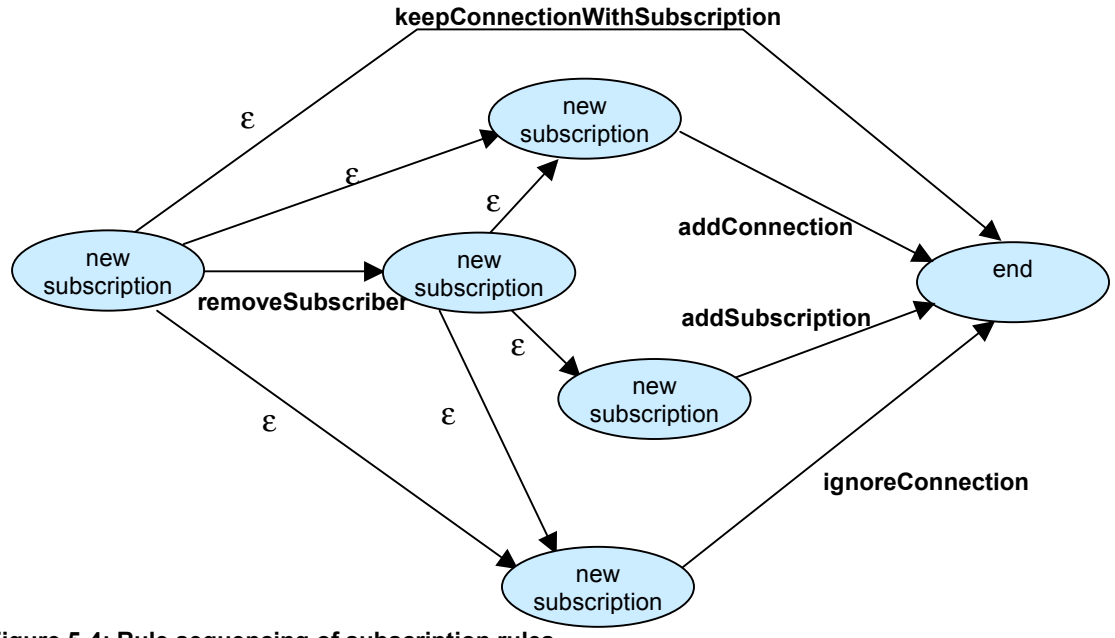


Figure 5-4: Rule sequencing of subscription rules

In the case that the subscription does not already exist the subscription rule has to be generated. For this the following rule is used.

Listing 5-8: Rule for the generation of a new subscription rule

```
(defrule addSubscription
    "If the new subscription does not yet exist, then create the new subscription and
    add the connection with its priority to it"
    ?new <- (new_subscription (atoms $?atoms) (connection ?connId) (priority ?pri))
    ?sub <- (sub_count (value ?count))
    (form_function (id ?formId) (nodes $?n) (branches $?b) (connections $?connections))
    (not (subscription (atoms $?atoms) (form_id ?formId)))
    (test (neq (member$ ?connId $?connections) FALSE))
    =>
    (assert (subscription (id ?count) (atoms $?atoms) (connections (create$ ?connId))
        (priorities ?pri) (form_id ?formId)))
    (createRule ?count $?atoms)
    (modify ?sub (value (+ ?count 1)))
    (retract ?new)
)
```

This rule takes the *new_subscription* fact and the *form_function* fact and tests that the connection is not yet registered to another subscription. If the rule fires a *subscription* fact is asserted into the fact-base, that maintains the connections of the subscribed clients and the priorities, and call the createRule function to generate the new rule. The new fact and the new rule have the same ID to reference each other.

Listing 5-9: generated subscription rule

```
(defrule subscription0
    (atom (id p0) (satisfied ?p0))
    (atom (id p1) (satisfied ?p1))
    (atom (id p4) (satisfied ?p4))
    (atom (id p5) (satisfied ?p5))
    (atom (id p7) (satisfied ?p7))
    (subscription (id 0)(atoms p7 p5 p4 p1 p0) (connections $?connections)
                  (priorities $?priorities))
    (test (or (or ?p0 ?p1) (and ?p4 (or ?p5 ?p7))))
    =>
    (assert (result (connections $?connections) (priorities $?priorities)))
)
```

The result of the subscription process is a generated rule in the rule-base that represents the subscription. The form function, the subset of atoms that apply to the client's prioritized subscription, the connection ID of the client, and the priority for the client are input to the generation of a rule. The antecedent of the rule is a pattern of all included atoms, the subscription fact that lists the connections with appropriate priorities, and the form function as a logical test. The name of the rule is based on the value in the id-field of the subscription fact. An example subscription rule is shown in Listing 5-9.

5.4.1.3 Selection rules

The selection is performed in two steps. First the atoms are matched and secondly the subscription rules are evaluated.

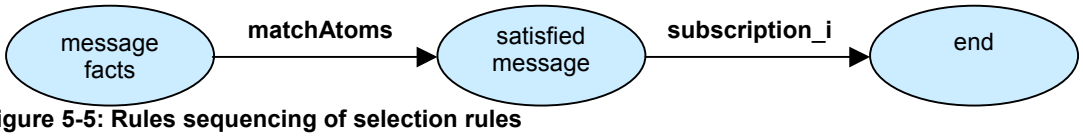


Figure 5-5: Rules sequencing of selection rules

A rule in the rule-base called *match_atom*, see Listing 5-10, tests if an event notification matches an atom. All event attributes in the notification are asserted as separate facts. The antecedent is a pattern with a notification and an atom and a logical test that implements Equation 4-7. When the rule fires it sets, the satisfied field of the atom fact to TRUE. Next the subscription rules can be checked.

Listing 5-10: match rule rule

```
(defrule match_atom
    ?nf <- (notification (name ?n_name) (value ?n_value))
    ?fs <- (atom (id ?a_id)(name ?a_name)(operator ?a_operator)(value ?a_value)
        (satisfied ?a_satisfied))
    (test (and (eq ?n_name ?a_name)(eq ?a_satisfied FALSE)
        (eval (str-cat "(" ?a_operator " " ?n_value " " ?a_value ")"))))
    =>
    (modify ?fs (satisfied TRUE))
    (retract ?nf)
)
```

5.4.1.4 Location awareness rules

The locations of three entities are maintained: of the user, of the past UForms, and of the new UForm. When the location of one of these changes a location fact is asserted or modified.

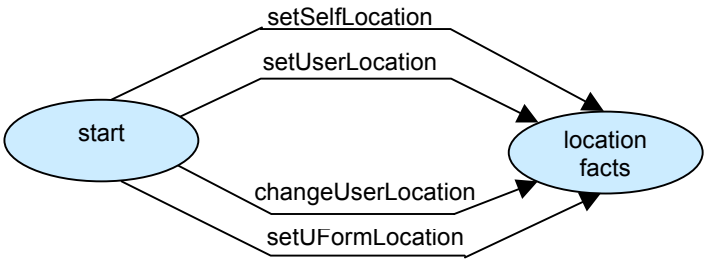


Figure 5-6: Rule sequencing for location awareness rules

When a distance condition is defined in a subscription file a distance fact is asserted in the fact-base that registers the two objects between which the distance is to be measured. For each user or event notification in the fact-base that match a distance parameter a location fact is generated. Next the distance can be deducted by the straightforward rule *setDistance*.

Listing 5-11: Rule for calculating distance

```
(defrule setDistance
    "Calculate the distance of two locations indicated by the parameters"
    (distance (id ?id) (param1 ?p1) (param2 ?p2))
    (location (id ?p1) (x ?x1) (y ?y1) (z ?z1))
    (location (id ?p2) (x ?x2) (y ?y2) (z ?z2))
    (newUform)
    =>
    (bind ?distance (sqrt (+ (** (- ?x1 ?x2) 2) (** (- ?y1 ?y2) 2) (** (- ?z1 ?z2) 2))))
    (assert (message (name ?id) (value ?distance)))
)
```

When the distance is part of a count condition we can count the distance facts that satisfy the requested distance with the following query.

Listing 5-12: Query for counting distance facts

```
(defquery countDistances
        "Count all the distance facts with the given id"
        (declare (variables ?id ?value))
        (message (name ?id) (value ?mesValue))
        (test (<= ?mesValue ?value))
)
```

6 IMPLEMENTATION ISSUES

Issues concerning implementation of the agent are described in this chapter. We start off with the subscription language. In this section is shown how the subscription language is structured. In the second section of this chapter we discuss the code used for the prioritized replication. We continue this chapter with a section about the data-model that is followed by a section about context aware measurements. In the last section of this chapter we discuss how the rules are set and applied by the agent.

6.1 Subscription language

The subscription information is defined in an XML file. An example of such a file is shown in Listing 6-1. The subscription is structured like a tree with logical operators as nodes and atomic expression as leafs. The tree-structure of logical operators is the Γ function, introduced in section 4.2.1, and the atom elements form the set Φ of atoms. Thus the total subscription file represents $\Gamma(\Phi)$. This is the structure clients may subscribe to by sending relevance vectors *r*. The size of the vector must equal the number of atom elements in the subscription XML file. Each element in the relevance vector describes the importance of a corresponding atom for the client. When the value of relevance vector is equal to zero the client is not interested in the atom at all and it will not appear in the client's prioritized subscriptions.

Listing 6-1: Example of the XML subscription file

```
<SUBSCRIPTION filename="subscription.xml">
  <LOGICAL_OPERATOR value="OR">
    <LOGICAL_OPERATOR value="OR">
      <ATOM name="type" operator="=" value="document"/>
      <ATOM name="type" operator="=" value="time"/>
      <ATOM name="type" operator="=" value="overlay"/>
      <ATOM name="type" operator="=" value="chat"/>
    </LOGICAL_OPERATOR>
    <LOGICAL_OPERATOR value="AND">
      <ATOM name="type" operator="=" value="unit"/>
      <LOGICAL_OPERATOR value="OR">
        <ATOM name="affiliation" operator="=" value="H"/>
        <ATOM name="affiliation" operator="=" value="F"/>
        <ATOM name="affiliation" operator="=" value="N"/>
      </LOGICAL_OPERATOR>
    </LOGICAL_OPERATOR>
  </LOGICAL_OPERATOR>
</SUBSCRIPTION>
```

Suppose we use the subscription of Listing 6-1, and the priority matrix of Table 4-4. When the context value *c* is *LOW* and we submit the relevance vector (`H M M L H H M L`), the server will setup the following prioritized subscriptions for the client.

Table 6-1: Example prioritized subscriptions.

Priority	Subscription
LOW	type $\in$ {document, time, overlay, chat} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H, F, N})
MEDIUM	type $\in$ {document, time, overlay} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H, F})
HIGH	type $\in$ {document} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H})

When the quality of the connection or the relevance vector changes over time the prioritized subscriptions for the connection are adjusted to the new values.

6.1.1 Location awareness in subscription language

To facilitate location awareness the subscription language must be able to express location-aware conditions. To achieve this the subscription language is extended with some keywords that are treated differently. The two keywords introduced are *distance* and *count*. Atoms using these keywords are parameterized as show in the listings Listing 6-2 and Listing 6-3. The parameter tags have two fields: name and properties. The name is the type name of the object referred to by the tag. The possible type names are: user, uform, this, or distance. The distance type name can only be used by the *count* keyword. The properties field summarizes the conditions that apply to the referred object. The user type parameter references a user object that can be specified by the username. The uform type specifies previous processed event notifications that may be conditioned by the contents of the notification. When the type is equal to *this* it refers to the current event notification, which is not further conditioned because the conditions that must apply the current conditions are set in the other atoms. Finally the distance parameter specifies the distance within which the counted objects must be.

Listing 6-2: Sample code for *distance* keyword

```
<ATOM name="distance" operator="&lt;=" value="150">
        <PARAMETER name="user" properties="username=user1"/>
        <PARAMETER name="uform" properties="affiliation=H"/>
</ATOM>
```

To condition a subscription to a certain distance between two objects the code looks like the one in Listing 6-2. In the atom field is specified which operator and value applies to the condition. The parameter atoms describe the objects between which the distance is to be measured.

Listing 6-3: Sample code for *count* keyword

```
<ATOM name="count" operator="&lt;=" value="3">
       <PARAMETER name="uform" properties="affiliation=H"/>
       <PARAMETER name="user" properties="username=user1"/>
       <PARAMETER name="distance" properties="value=150"/>
</ATOM>
```

Listing 6-3 illustrates an atom with the count keyword. The atom fields specify the operator and the value that apply to the count. The parameter tags specify the objects between which the distance is to be measured, and the value that must apply to the measured distance. A count atom may also just count a certain type of object. In that case it only has one parameter referring to the object to be count.

6.1.2 Grouping subscribers

As the clients subscribe, a lot of subscriptions will overlap or even be identical. For performance reasons, clients with identical subscriptions are grouped. For each subscription we register the clients that are using the subscription with their priorities. When *n* clients use a subscription only one rule has to be evaluated instead of *n*.

The grouping of subscribers makes subscription management necessary. When a subscriber changes its subscription or priority the rule-base must be kept consistent. In

Figure 6-1 is shown how consistency of subscriptions can be maintained. When a new subscription for a client is set up we first have to check if it already exists. If it does not we make sure that the user-priority combination is not used in another subscription, and subsequently add the new subscription. In the case that the subscription does exist we see if the user is already subscribed to it, and if so with the right priority. We ensure that the user-priority is not registered to another subscription and add the right user-priority combination to the new subscription.

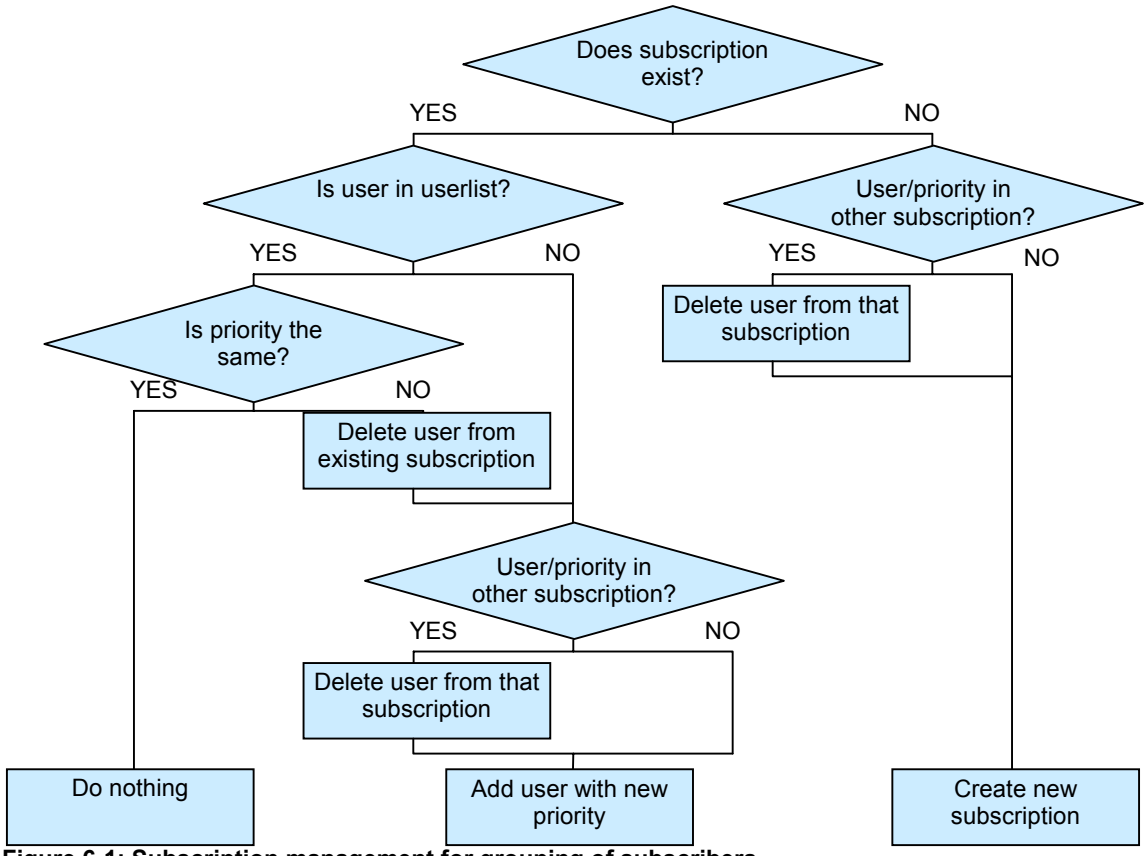


Figure 6-1: Subscription management for grouping of subscribers

6.1.3 Multiple subscription files

Each user has the possibility to send a subscription XML file, as discussed in the beginning of this section, to the server. The server maintains a list of all defined subscription XML files. When the user just wants to use an existing file, only sending the filename is sufficient. The rule-base kept track of which users are using what form function. For each defined subscription XML file the rule-base asserts a form function. The atoms that appear in the various form functions are aggregated as one global set of atoms so no redundant atoms are saved.

The benefit of multiple subscription files is flexibility. Each user has the freedom to construct a subscription XML file that fulfills his specific needs.

6.2 Prioritized replication

The advantage of prioritization of the subscriptions is that we know the urgency of the replication of different events for each client. The server can order events by urgency and send it by magnitude of priority. When we assign a high priority to a time critical event we can be assured it is sent before other events with lower priority.

In this section we discuss issues of prioritized replication. We first take a look at how the number of priority levels should be chosen. Next we show sample code for the replication algorithm using prioritization.

6.2.1 Number of priority level

The optimal number of priorities to use depends on the type of system the data distribution agent is used for. With an increasing number of priority levels the interval classes of the context will be smaller with the result that subscriptions have to be reevaluated more often. However a more accurate ordering of importance of events will be possible, which is useful in the case of heavy data traffic. On the other hand, in the case many users are logged into the system cutting down adaptation time by decreasing the number of priority levels may be crucial.

6.2.2 Replication code

A sample code for the replication algorithm, using prioritization is presented here, see Listing 6-4. The priority levels are integers, with 0 as highest priority and as the integer increases the priority will decrease. For each priority level a dirty queue is assigned that contains the events that has to be sent with that priority. Initially all the events are placed in the dirty queue of priority 0. Next the real priority of the event for the user is calculated. When the real priority is 0 then the event is sent immediately; if it is unequal to 0 the event is placed in the dirty queue of the real priority.

Listing 6-4: Example code for replication algorithm

```
for (int priority=0; priority<numberOfPriorities; priority++) {
        for (int i=0; i<dirtyQueue[priority].size()) {
                Event event = dirtyQueue[priority].next();
                if (priority == 0)
                        int newPriority = getUFormSendingPriority(event);
                        if (newPriority != 0) {
                                dirtyQueue[priority].remove(event);
                                if (newPriority > 0) //-1 is no replicate
                                        dirtyQueue[newPriority].add(event);
                                continue;
                        }
                }
                connection.send(event);
         }
}
```

6.2.3 Synchronization

The synchronization of writing and reading prioritized events is solved by sequentially: determining the priority of the event, putting it in the right buffer, and reading the buffers. No further synchronization is needed.

6.3 Data model

The data model of the agent is depicted in Figure 6-2. On top of the model is the abstract class *DataDistributionAgent*, which models the interface of the agent. The class has a repository that points to the global data repository of the server. With the methods *setRules* and *applyRules* the agent sets up the subscription and test an event with the subscriptions respectively.

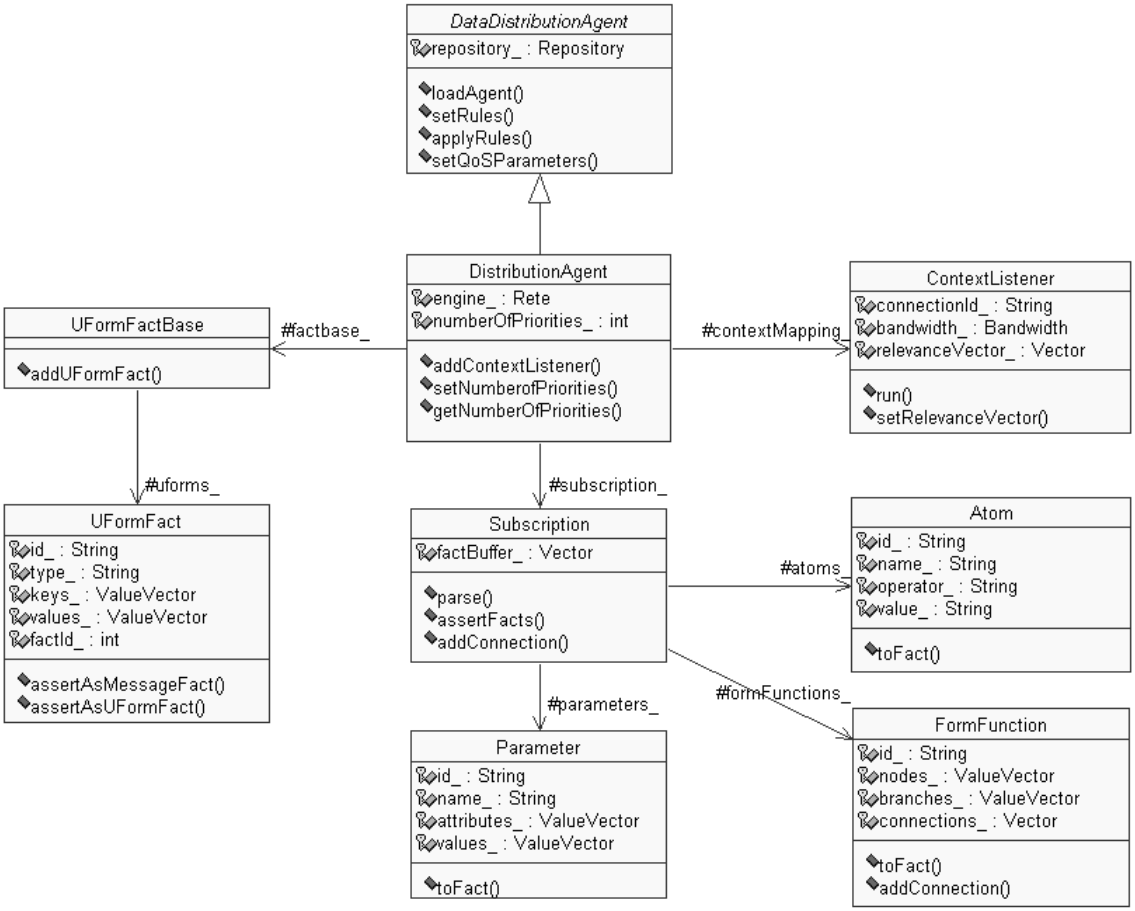


Figure 6-2: Data model of data distribution agent

The *DataDistributionAgent* class is a generalization of the class *DistibutionAgent* at the center of the model. The engine property is the inference engine of the expert system. When a client logs in, the *DistributionAgent* object executes *addContextListener()*, which results in a new *ContextListener* class with the connection ID, the bandwidth, and the relevance vector set. The subscription information is set using the *setRules()* method. The subscription XML file, see Listing 6-1, is parsed by the *Subscription* object into *Atom* objects, *FormFunction* objects and *Parameter* objects. Both the *Atom* class, the *FormFunction* class as the *Parameter* class have a method called *toFact()* that converts the object to a Jess Fact. All facts are saved in the *factBuffer* property of the Subscription object. When the method *assertFacts()* is executed all the facts in *factBuffer* are assert in the fact-base of the expert system.

The event notifications in the system are called UForms (Universal Form). When an UForm arrives at the server the agent adds the UForm to the UformFactBase. The UForm is parsed by the *UformFact* class and *assertAsMessageFact()*, which asserts a temporary fact, is called. Subsequently the *applyRules()* method of the agent is executed to match the UForm with the subscription. After the application of the rules the UForm is permanently saved in the fact-base for future use by calling the *assertAsUFormFact()* method of the *UformFact* object.

6.4 Context awareness measurements

We will not discuss the method of measurements of the context awareness. However two ways of passing the awareness information are distinguished. The two ways are only relevant for data-independent information, and are described below.

- *Push approach*. A thread continuously measures the contextual variables. When the context changes the thread notifies the agent of this change. Thus the contextual information is pushed to the agent. The advantage of this approach is that the values are always up-to-date.
- *Pull approach*. When an agent needs the context information it asks the context awareness component to measure the variables and pass the values. The agent pulls the information. The benefit of this approach is that the measurements are only performed when needed.

The implemented agent uses the push approach. This is a prerequisite for our approach because the subscription algorithm needs to be aware of changes in the context to maintain up-to-date subscriptions.

For network awareness, a thread constantly measures the bandwidth. This thread is defined in the Bandwidth class, and the changes are stored in a variable that is shared with the *DistributionAgent* class. The *contextListener* method in this class waits for changes of this variable, and when needed the subscription algorithm is called to adapt the subscriptions in the rule-base.

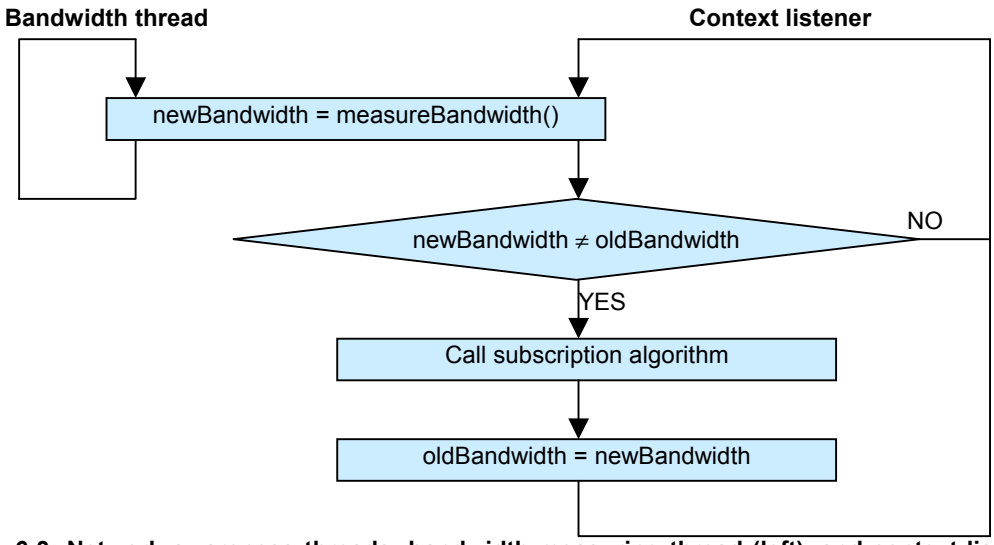


Figure 6-3: Network awareness threads: bandwidth measuring thread (left), and context listener of *DistributionAgent* class (right)

6.5 Setting and application of rules

The abstract class *DataDistributionAgent* has two major method, namely setRules() and applyRules(). The first sets up subscription rules for the subscriber when he connects and the second applies the rules when events arrive at the agent. Both interact with the expert system to achieve their tasks.

6.5.1 Set rules

The input to the method setRules() is an subscription XML file as described in 6.1, and the connection ID of the subscriber. First the method will parse the XML file to a DOM-tree so it is able to read the subscription information. Each subscription file has a name with is used as identifier of the subscription. The method checks if the subscription is already added to the rule-base by looking up the name of the subscription file in a list of added subscriptions. If this is not the case the subscription is added to the rule-base. The last step of the method is to actually subscribe the connection ID to the subscription. In Figure 6-4 the method is illustrated with a diagram.

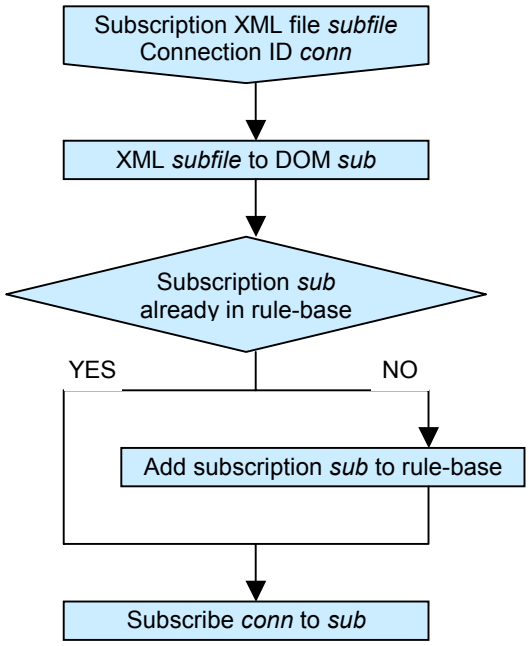


Figure 6-4: Diagram of the setRules() method

6.5.2 Apply rules

The *applyMethod()* of the *DistributionAgent* class is triggered when a UForm, an event, arrives at the agent. The UForm is added to the fact-base of the expert system and the inference engine runs the subscription rules. A result fact is created with a list of connections and a list of priorities. The connections and priorities in the lists correspond with each other. All connections and priorities are placed in the new lists *connList* and *priList*. When a connection already exists in the *connList* the corresponding priority is set to the highest priority of the current priority in *priList* and the priority of the new element in the list of the result fact. The combination of *connList* and *priList* is returned. The method is illustrated in Figure 6-5.

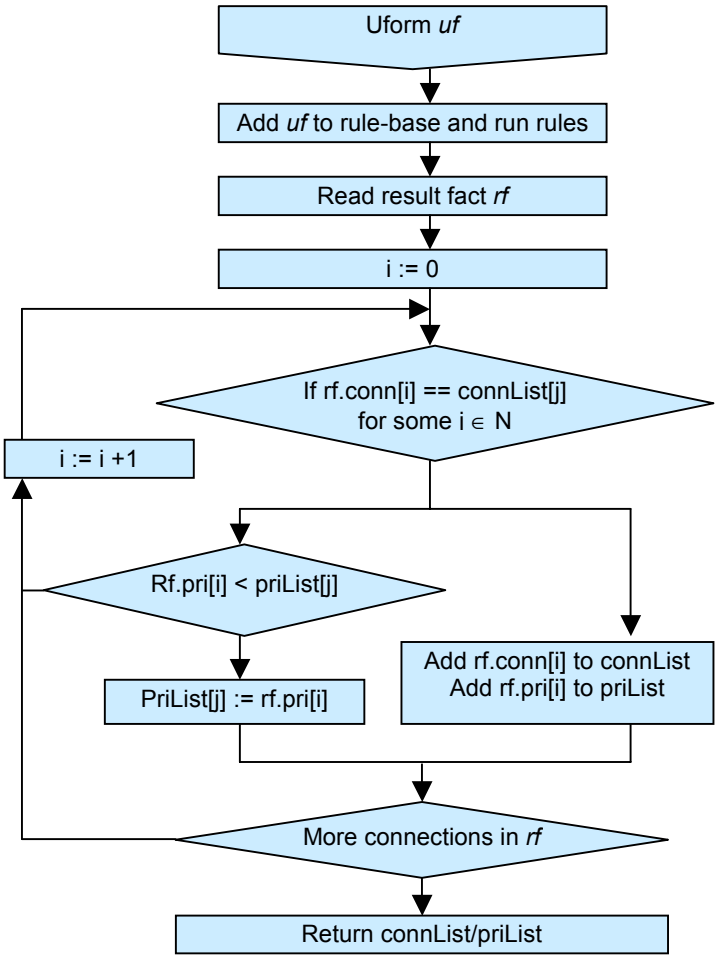


Figure 6-5: Diagram of the applyRules() method

6.5.3 Synchronization

Prioritized subscriptions are set by one algorithm and used by another. The two algorithms have to be synchronized else the event selection algorithm might use a subscription that is being updated by the subscription algorithm. Because both algorithms are executed by the expert system Jess deals with the synchronization. When the expert system is running for one algorithm the other algorithm has to wait until the expert system is done.

7 APPLICATION OF AGENT

In this chapter is demonstrated how the data distribution is applied. The system we will be reviewing is called DISCIPLE. The data distribution agent is part of this system. The application running on this system is called FLATSCAPE. We demonstrate how the system works by simulation of a couple of scenarios.

7.1 DISCIPLE

DISCIPLE (acronym for Distributed System for Collaborative Information Processing and Learning) is used for collaboration environments that share data between wired and wireless devices with widely disparate capabilities [19]. In Figure 7-1 the architecture of the DISCIPLE system is depicted. The system approached the problem of different platforms data-centric, and transforms the data locally to the characteristics of the device.

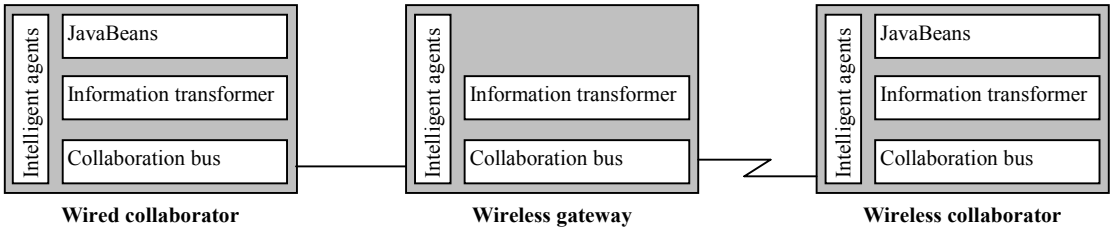


Figure 7-1: DISCIPLE architecture with wired and wireless collaborator

The JavaBeans are actually not part of the system, but are supplied by the application developer. The information transformers manipulate incoming data to the specifics of the platform and abstract outgoing data. The collaboration bus is an application-independent communication channel, which is used to send events between the clients and server. The intelligent agents plane support all the previous mentioned layers. The data distribution agent subsides in this plane.

7.1.1 Distributed server

For better scalability the subscribers are distributed over multiple servers. The servers communicate with each other in a peer-to-peer manner. Each server defines a relevance vector to specify what kind of events it wants to be notified by. This avoids unnecessary network traffic between servers. The relevance vectors are stated in an XML file that is read when a server is started. Each server starts up a data distribution agent for replication of events to its neighboring servers. Next it sets up the relevance vectors of these servers in the agent.

Listing 7-1: Server specification XML file

```
<SERVERLIST>
  <SERVER hostname="128.6.237.92" port="5000" name="1" value="0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9"/>
  <SERVER hostname="128.6.237.92" port="5300" name="2" value="0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9"/>
  <SERVER hostname="128.6.237.61" port="5500" name="3" value="0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9"/>
</SERVERLIST>
```

When we observe the network of servers as one global server the whole system works like a client-server system that communicates as a publish/subscribe model.

7.1.2 State merge

For collaborative systems the sharing of data is the main focus. Each client needs to have a representation of the same dataset to work on. As a consequence when a client connects to the server the local dataset needs to merge with the global dataset.

When a user logs on to the system, the system synchronizes the data to the local data repository of the client. This is called the state merge. The merger collects all the events that have to be sent to the client in an array. This array is send to the data distribution agent that filters out the events that match the client's subscription. A method called *refresh()* loops through the array of events and calls the *applyRules()* method for each event.

Listing 7-2: Code of refresh() method

```
public UForm[] refresh(long id) {
        Vector update = new Vector();
         UForm[] uforms = repository_.getUForms();
         for (int i=0; i<uforms.length; i++) {
                if (testUformForUser(uforms[i], id)) {
                        update.addElement(uforms[i]);
                }
         }
         UForm[] result = new UForm[update.size()];
         for (int j=0; j<update.size(); j++) {
                result[j] = (UForm)update.elementAt(j);
         }
         return result;
}
```

In Listing 7-2 the code of the refresh method is shown. First an array lists all event notifications, called UForms. Then for each UForm in the array we call *testUformForUser*(), which tests if the connection of the user appears in the list returned by the method *applyRules()*. All the selected UForms are listed in the update vector that is converted to an array called *result*.

7.2 Flatscape

FLATSCAPE is an application that runs on top of DISCIPLE. It allows collaborators to work on a military operational plan. Friendly, hostile or neutral units of different types can be added, removed, or moved in an area to stage the operational plan. The collaborators will be commanders of different ranks working on wired workstations as well as on wireless handhelds in the field.

7.2.1 Tasks and roles

Based on the rank of the commander different types of data will be relevant. For example, a team leader will only be interested in information about his team members and about the hostile units in the neighborhood. On the other hand, a high rank commander will be interested in the locations of all units and hostile units in the whole area. The relevance vectors are based on a task and the role of the commander. The role corresponds with the rank of the commander and the task indicates the current work of the commander. The relevance vectors are defined in an XML file. Listing 7-3

demonstrates such a file for two roles: team leader and platoon leader, for three different tasks: offense, defense, and intelligence. The relevance vectors in this XML file are based on the subscription XML file of Listing 6-1. With each relevance-vector a default field tells if the role-task combination is the default combination when FLATSCAPE is start up.

Listing 7-3: Roles and task definition

```
<RELEVANCE>
        <ROLE name="team leader">
                <TASK name="offense">
                        <RELEVANCE value="1 0.1 0.3 0.8 1 1 0.9 0.2" default="no"/>
                </TASK>
                <TASK name="defense">
                        <RELEVANCE value="1 0.3 0.1 0.8 1 1 0.8 0" default="no"/>
                </TASK>
                <TASK name="intelligence">
                        <RELEVANCE value="1 0 0 0.8 1 1 0 0.8" default="no"/>
                </TASK>
        </ROLE>
        <ROLE name="platoon leader">
                <TASK name="offense">
                        <RELEVANCE value="1 0.3 0.3 0.8 1 1 0.9 0.5" default="yes"/>
                </TASK>
                <TASK name="defense">
                        <RELEVANCE value="1 0.3 0.3 0.8 1 1 0.5 0.9" default="no"/>
                </TASK>
                <TASK name="intelligence">
                        <RELEVANCE value="1 0 0.2 0.8 1 1 0 0.8" default="no"/>
                </TASK>
        </ROLE>
</RELEVANCE>
```

7.2.2 User interface

In this section we will briefly discuss the user interface of FLATSCAPE. Figure 7-2 shows the interface. The most important view is the map and the units that are placed on it. With the tools pane, units can be added and modified, in the unit pane, units can be specified, in the users pane all users logged into the system are listed, and the overlays pane shows the structure of all the objects on the map. The Task pane has two choice boxes, one for the role and one for the task of the commander. The values in these choice boxes are derived from the XML file that specifies the roles and tasks, see Listing 7-3.

When a unit is added, removed or moved around by the user, an UForm containing the up-to-date information of the unit, is sent to the server. At the server the UForm is passed to the data distribution agent that decides to which other users and with what priority the UForm is to be distributed. Next DISCIPLE loops through the list on connections and sends the UForm. At the client side the UForm is passed to FLATSCAPE that updates the unit's graphical representation on the map.

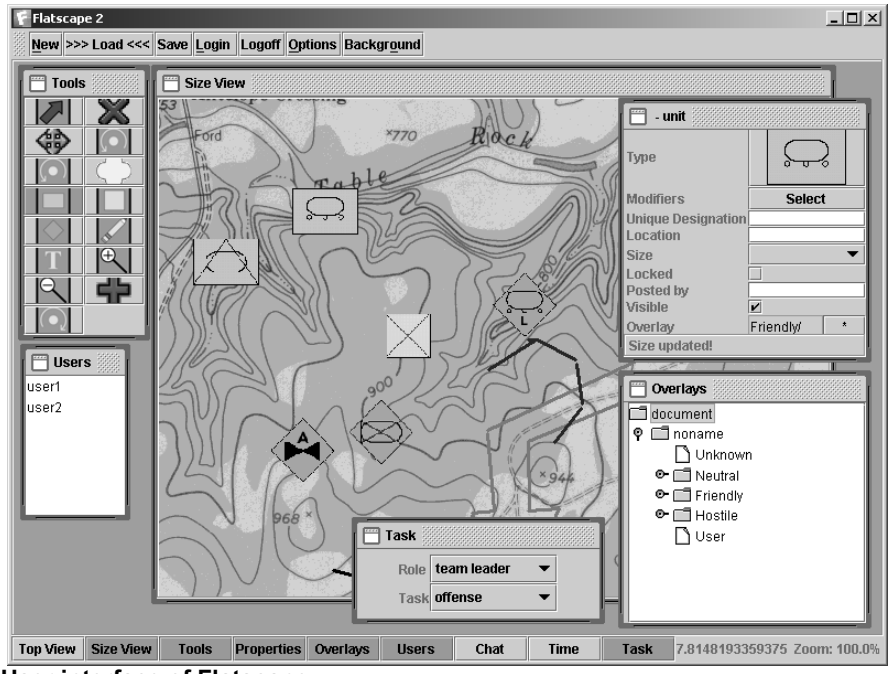


Figure 7-2: User interface of Flatscape

7.3 Simulation

To demonstrate the effect of the data distribution agent three scenarios have been simulated. The scenarios focus on the *(i)* selectivity of the agent *(ii)* the network awareness of the agent *(iii)* the location awareness of the agent.

7.3.1 Selectivity

The selectivity simulation tests if the event selection algorithm works correctly. The test is executed simply by exchanging events between clients with different interest. The resulting datasets should reflect the different interests.

7.3.1.1 Goal and setting

The goal of this simulation is to demonstrate the selective property of the agent. The core task of the agent to select event for subscribers so it is crucial this task is performed satisfactory. Two clients will connect to a server. Both will use the subscription information as defined in Listing 6-1 and the roles and tasks as defined in Listing 7-3. Client A will perform the role of team leader with a defensive task, and client B will perform the role of team leader with an intelligence task. So both clients have different relevance vectors.

Both clients will send one UForm with hostile affiliation, one with friendly affiliation, and one with neutral affiliation. Because of the different relevance vectors each client will have a different data-subset in the end. Client A is interested in hostile and friendly units, and client B is interested in hostile and neutral units.

7.3.1.2 Hypothesis and results

The result of this simulation is shown in Figure 7-3. The diamond, rectangle, and square icons represent hostile, friendly, and neutral units respectively. Client A added the icons with a cylindrical form and client B added the icons with two stripes. So it is to be expected that client A only receives diamond and rectangle icons with two stripes from client B, and client B only receives diamond and square icons with cylindrical forms.

Client B did not receive the friendly unit, rectangle with cylindrical form, added by client A, because the relevance vector for a team leader with an intelligence task is 0. And since a team leader with a defensive task has 0 relevance for neutral units, client A did not receive the neutral unit, square with two stripes, sent by client B. Of course client A does see the neutral (square) unit with cylindrical form because it sent it itself. The same applies to client B for the friendly unit sent by client B.

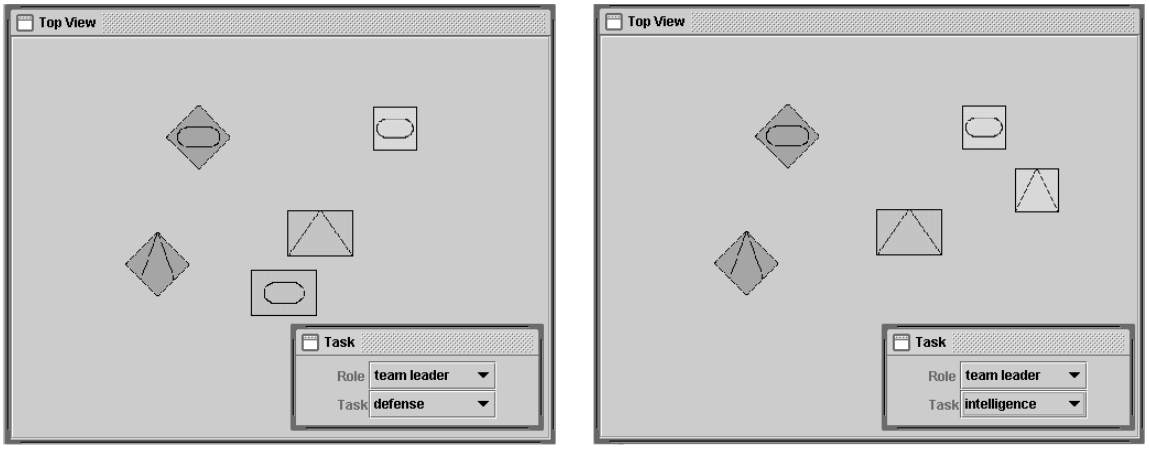


Figure 7-3: Result of selectivity simulation for client A (left) and clientB (right)

7.3.1.3 Concluding remarks

We see that the results justify the hypothesis. The selection of events for both clients is as we expected. This means that the implementation of the event selection algorithm works correctly. Clearly the expert system makes the right decisions based on the right subscriptions.

7.3.2 Network awareness

The subscription algorithm should also be tested. This algorithm is responsible for the adaptation of the subscriptions to the context. The scenario described here tests the adaptation to network awareness.

7.3.2.1 Goal and setting

This second scenario will show the network awareness of the agent. One client will connect to the server with the subscription information of Listing 6-1 and the role and task file as in Listing 7-3. The user is a platoon leader with an offensive task. We will take a look at the fact-base and the rule-base to show the adaptation at two different times

when the bandwidth of the connection is different. The priority level of the agent is set to 3.
The client is running on a T20 laptop and is connected through a wireless LAN. Walking around with the laptop will make the bandwidth change. We make two snapshots with different bandwidths that should certainly result in different subscriptions for the client.

7.3.2.2 Hypothesis and results

When the quality of the connection is high the client does not assign different priorities to events. In this situation the throughput of data is so good that all event can be sent immediately. If the bandwidth is low the events will be assigned different priorities in order to sort the events that are waiting to be sent by importance. We expect to witness this by a changing subscription set where the number of possible priorities assigned to events increases when the quality of the connection decreases.

The bandwidth at the first point in time is equal to 3750.0 kbs. The subscription facts and rules are shown in Listing 7-4. The listing shows the subscriptions for priority 0 and priority 1. No subscription rule is defined for priority 2, because it is the same as for priority 1. The subscription with ID set to 0, and with priority 0, contains all the atoms a0 to a7, while the other subscription for priority 1, does not contain any atom. Consequently the second subscription will not select any events.

Listing 7-4: Subscription facts and rules for platoon leader with offensive task when bandwidth = 3750.0 kbs

```
(subscription (id 0) (atoms a7 a6 a5 a4 a3 a2 a1 a0) (connections
    tcpipconn243932837794985469) (priorities 0) (form_id subscription.xml))
(subscription (id 1) (atoms ) (connections tcpipconn243932837794985469) (priorities 1)
    (form_id subscription.xml))

(defrule subscription0
    ""
    (declare (salience 0) (node-index-hash 0))
    (atom (id a0) (satisfied ?a0))
    (atom (id a1) (satisfied ?a1))
    (atom (id a2) (satisfied ?a2))
    (atom (id a3) (satisfied ?a3))
    (atom (id a4) (satisfied ?a4))
    (atom (id a5) (satisfied ?a5))
    (atom (id a6) (satisfied ?a6))
    (atom (id a7) (satisfied ?a7))
    (subscription (atoms a7 a6 a5 a4 a3 a2 a1 a0) (connections $?user_list) (priorities
        $?priorities))
    (test (or (or ?a0 ?a1 ?a2 ?a3) (and ?a4 (or ?a5 ?a6 ?a7))))
    =>
    (assert (result (connections $?user_list) (priorities $?priorities))))

(defrule subscription1
    ""
    (declare (salience 0) (node-index-hash 0))
    (subscription (connections $?user_list) (priorities $?priorities))
    (test (eq TRUE FALSE))
    =>
    (assert (result (connections $?user_list) (priorities $?priorities))))
```

After a while the bandwidth decreases to 195.6 kbs. The subscriptions are adapted as shown in Listing 7-5. The selection of data is now more diversified over the prioritized selections, leading to the delay of middle and low priority UForms. Three subscription

rules are created, for each priority level one. The atom-sets or higher priority (lower integer) is a sub-set of the atom-set of a lower priority (higher integer). So a higher priority subscription rule is more restrictive than lower priority subscription rules.

Listing 7-5: Subscription facts and rules for platoon leader with offensive task when bandwidth = 195.6 kbs

```
(subscription (id 2) (atoms a6 a5 a4 a3 a0) (connections tcpipconn243932837794985469)
        (priorities 0) (form_id subscription.xml))
(subscription (id 3) (atoms a7 a6 a5 a4 a3 a0) (connections
        tcpipconn243932837794985469) (priorities 1) (form_id subscription.xml))
(subscription (id 4) (atoms a7 a6 a5 a4 a3 a2 a1 a0) (connections
        tcpipconn243932837794985469) (priorities 2) (form_id subscription.xml))

(defrule subscription4
    ""
    (declare (salience 0) (node-index-hash 0))
    (atom (id a0) (satisfied ?a0))
    (atom (id a1) (satisfied ?a1))
    (atom (id a2) (satisfied ?a2))
    (atom (id a3) (satisfied ?a3))
    (atom (id a4) (satisfied ?a4))
    (atom (id a5) (satisfied ?a5))
    (atom (id a6) (satisfied ?a6))
    (atom (id a7) (satisfied ?a7))
    (subscription (atoms a6 a5 a4 a3 a2 a1 a0) (connections $?user_list) (priorities
$?priorities))
    (test (or (or ?a0 ?a1 ?a2 ?a3) (and ?a4 (or ?a5 ?a6 ?a7))))
    =>
    (assert (result (connections $?user_list) (priorities $?priorities))))

(defrule subscription3
    ""
    (declare (salience 0) (node-index-hash 0))
    (atom (id a0) (satisfied ?a0))
    (atom (id a3) (satisfied ?a3))
    (atom (id a4) (satisfied ?a4))
    (atom (id a5) (satisfied ?a5))
    (atom (id a6) (satisfied ?a6))
    (atom (id a7) (satisfied ?a7))
    (subscription (atoms a7 a6 a5 a4 a3 a0) (connections $?user_list) (priorities
$?priorities))
    (test (or (or ?a0 ?a3) (and ?a4 (or ?a5 ?a6 ?a7))))
    =>
    (assert (result (connections $?user_list) (priorities $?priorities))))

(defrule subscription2
    ""
    (declare (salience 0) (node-index-hash 0))
    (atom (id a0) (satisfied ?a0))
    (atom (id a3) (satisfied ?a3))
    (atom (id a4) (satisfied ?a4))
    (atom (id a5) (satisfied ?a5))
    (atom (id a6) (satisfied ?a6))
    (subscription (atoms a6 a5 a4 a3 a0) (connections $?user_list) (priorities
        $?priorities))
    (test (or (or ?a0 ?a3) (and ?a4 (or ?a5 ?a6))))
    =>
    (assert (result (connections $?user list) (priorities $?priorities))))
```

7.3.2.3 Concluding remarks

In Listing 7-4, the bandwidth is high, only two subscriptions are created: subscription0 with priority 0 for the subscriber, and subscription1 with priority 1. However subscription0 will select all the hostile, friendly and neutral units, while subscription1 will select no units at all. So all units will be selected with priority 0.

In the second case, with low bandwidth, three new subscriptions apply for the same subscriber: subscription2, subscription3, and subscription4 with priorities 0, 1, and 2 respectively. In this case the subscriptions select different events with different priorities.

The hypothesis was right: with different bandwidth values the prioritization of events is also different. The number of possible priorities for high bandwidth and low bandwidth values are 1 respectively 3. So in the case of low bandwidth events with low relevance are deliberately delayed. We can conclude that the adaptation works and so the implementation of the subscription algorithms works for network awareness.

7.3.3 Location awareness

A mentioned earlier we distinguish data-independant and data-dependant context awareness. In this simulation we will test the data-dependant context awareness of the agent.

7.3.3.1 Goal and setting

In this simulation round we will demonstrate the location awareness of the agent. Two clients connect to the server. Client A submits the subscription file listed in Listing 7-6 and the relevance vector `(1 0.3 0.3 0.8 1 1 1 0.9 0.5)`. Only units within the distance of 150 from the user will be forwarded to client A.

Listing 7-6: Location aware subscription

```
<SUBSCRIPTION filename="test2.xml">
    <LOGICAL_OPERATOR value="OR">
        <LOGICAL_OPERATOR value="OR">
            <ATOM name="type" operator="=" value="document"/>
            <ATOM name="type" operator="=" value="time"/>
            <ATOM name="type" operator="=" value="overlay"/>
            <ATOM name="type" operator="=" value="chat.message"/>
        </LOGICAL_OPERATOR>
        <LOGICAL_OPERATOR value="AND">
            <ATOM name="type" operator="=" value="unit"/>
            <ATOM name="distance" operator="&lt;=" value="150">
               <PARAMETER name="user" properties="username=user1"/>
               <PARAMETER name="this" properties=""/>
            </ATOM>
            <LOGICAL_OPERATOR value="OR">
               <ATOM name="affiliation" operator="=" value="H"/>
               <ATOM name="affiliation" operator="=" value="F"/>
               <ATOM name="affiliation" operator="=" value="N"/>
            </LOGICAL_OPERATOR>
        </LOGICAL_OPERATOR>
    </LOGICAL_OPERATOR>
</SUBSCRIPTION>
```

Client B is only started to generate UForms. The UForms generated are clustered in two groups. Client A will move from the center of one cluster of UForms to the center of the other cluster of UForms.

7.3.3.2 Hypothesis and results

Because of the limited distance client A witnesses, we expect that it can only see all the UForms of one cluster at once. When client A is close to one cluster it will only see the UForms of that cluster and will not see the other UForms.

All the UForms generated by client B are shown in Figure 7-4 at the left side. Client A starts at position (0, 0) and after a while moves to (-100, 150). The results of the data distribution are shown in the middle and right side pictures of Figure 7-4. At the first position client A sees the four UForms of cluster A and none of cluster B. At the second position the situation is the other way around: client A sees all three UForms of cluster B and none of cluster A.

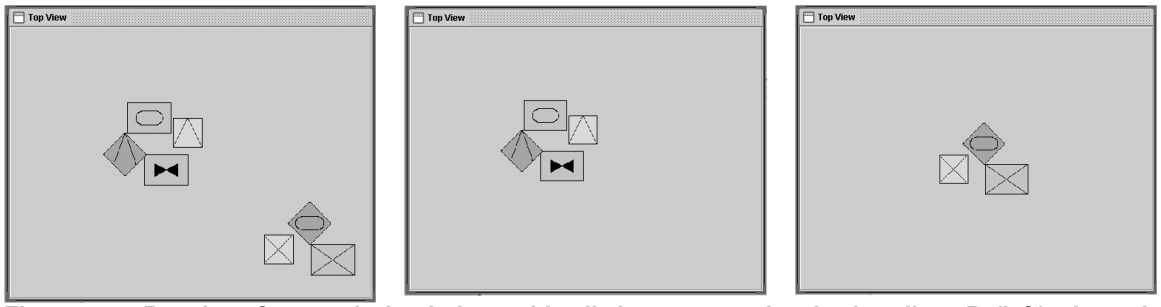


Figure 7-4: Results of second simulation, with all the generated units by client B (left), the unit received by client A at (0, 0) (middle), and the units received by client A at (-100,150)

7.3.3.3 Concluding remarks

The effect of the location-aware subscription of client A is clear. It only sees the UForms of one cluster at a time. The data-dependant adaptation is thus successful for location awareness.

8 EVALUATION

In this chapter we will evaluate the data distribution agent. This will give us insight in the computational performance of the agent. In the first section we will take a look at our experimental results and in the second section we will analyze the algorithm further.

8.1 Measurements

To test the performance of the data distribution agent experimentally we have performed a set of measurements. The first measurements concern the overhead of the agent, the second the scalability of the agent, and the last measurements are about the performance gain of the use of the agent.

8.1.1 Overhead

We have measured the overhead of the selective event replication with use of the agent compared to the event replication without selection. In this measurement, we recorded the time used for events to be distributed among 10 other clients. We are interested in the time increase of the replication with the increase of the number of events that are transmitted. We calculate the overhead by subtracting the time lapses of data distribution with and without agent and divide this by the time of data distribution without agent. See Equation 8-1.

$$overhead = (T_{conditional} - T_{unconditional}) / T_{unconditional} \times 100\%$$ **Equation 8-1**

The results of these measurements are shown in Figure 8-1. We see that most measurements fluctuate between 0.5 % and 3.5 %, with an average of approximately 1.5%. We can conclude that the data distribution agent scales well with an increasing number of events. The data distribution agent does not result in an increasing overhead.

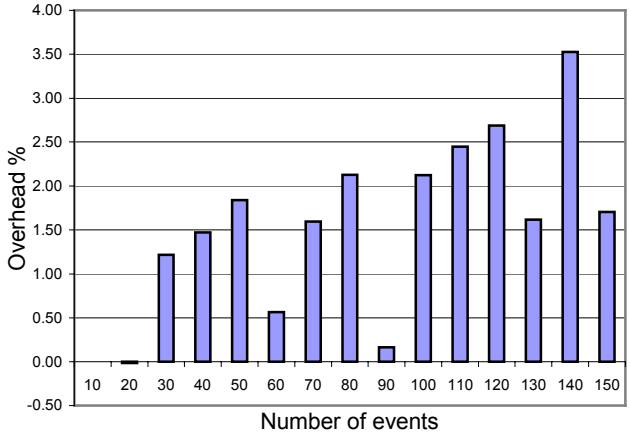


Figure 8-1: overhead of data distribution agent

An average overhead of just 1.5 % is very satisfactory. It means that the event selection algorithm does not impose a lot of extra execution time for the distribution of data. We

can also conclude that the overhead does not increase with the number of events. So the algorithm scales well for an increasing number of events.

8.1.2 Scalability

In order to assess the scalability of the data distribution agent the next set of measurements we are interested in the increase of time with an increasing number of users. The time recorded is between the moment an event is sent and the moment all users received the event. To show that the scalability can be further improved with an increasing number of servers the measurement is performed for a network with one, two and three servers that connect the users. The users are evenly distributed among the servers. The results are shown in Figure 8-2.

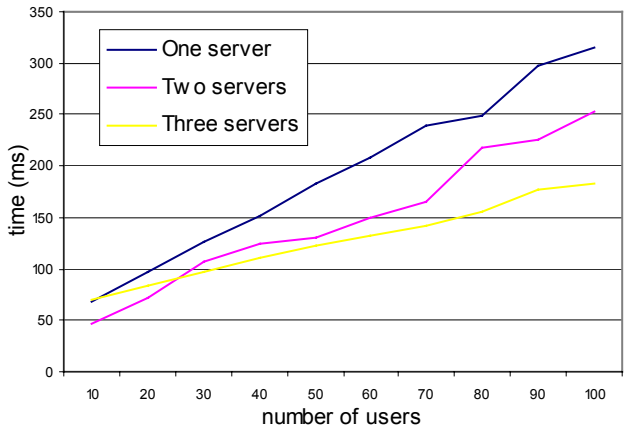


Figure 8-2: Scalability of data distribution agent

We can clearly see that all three graphs are linear, though the slope decreases with an increasing number of servers in the network. The graphs show us that for less than 25 users two servers are preferred and for more than 25 users a configuration of three servers performs better. It is to be expected that for a high number of servers the graph will be almost flat. This means the increase of users does not influence the time when we use many servers. Because of the linearity of the scalability graphs we can conclude that the data distribution agent scales well to an increasing number of users.

8.1.3 Performance gain

The last set of tests measure the performance gain of the data distribution agent compared to a system that does not use the agent. Since the agent avoids redundant replication of events computational time is saved. The aim of this test is the demonstration of this effect. We take a certain amount of events and increase the percentage of events that is selected. The time measured starts when the publisher sends the first event and ends when the subscriber receives the last selected event. The order in which events are sent is random, but obviously we have best and worst case. The best case is that all events to be selected are send first. The worst case is the situation when the last event sent is to be selected. If we are using a system without using the data distribution agent, all updates will have the same priority and thus be constant in this graph, this is shown by the

horizontal line. The time measured for distribution with this system starts when the first data item is sent and ends when the last data item arrives.

The results of the measurements are shown in Figure 8-3. In a normal situation the distribution time will be somewhere between the best case and the worst case. The distribution time increases linearly with increasing percentage of selected events. When less than 90% of the events are selected the use of the data distribution agent is beneficial.

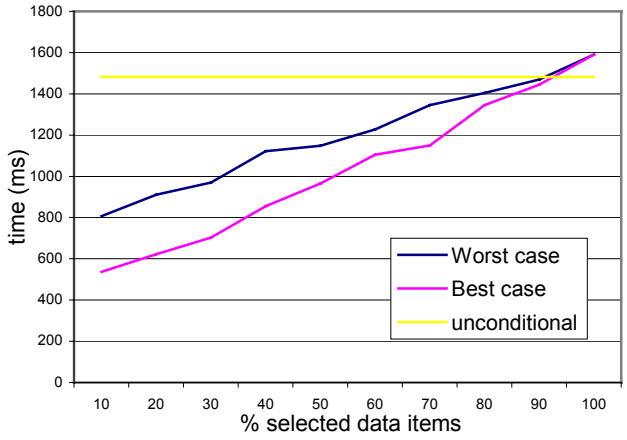


Figure 8-3: Performance gain of data distribution agent

It is to be expected that in normal situations the percentage of selected events will be between 30% and 60%, so a considerate amount of time will be gained with the use of the agent. We can conclude that the use of the event selection algorithm saves a lot of execution time and so the use of it is beneficial.

8.2 Analysis

In this section a probabilistic model is presented of the algorithms discussed in 4.2. This probabilistic model is used to analyze the benefit of grouping of subscribers.

8.2.1 Probabilistic model

Suppose N_A atoms are defined for a form function in the rule-base. Since each atom may of may not appear in a subscription rule the maximum number of subscriptions for the form function will be given by:

$$N_S = 2^{N_A}$$ **Equation 8-2**

This is the theoretical maximum of subscriptions; however the real maximum will be much lower because many possible subscription rules are not useful. Lets denote N_E as the number of subscriptions that already exist for the form function. Every time a user sets up its subscription rule, there are two possibilities:

1. The subscription rule already exists with a the probability of $P_1 = N_E/N_S$,
2. The subscription rule does not exist with a probability of $P_2 = (N_S\text{-}N_E)/N_S$

In the second case a new subscription is created and so the N_E is incremented. Over time N_E will increase and consequently P_1 will increase and P_2 will decrease. Eventually N_E will reach its maximum N_S. The increase of N_E is depicted in Figure 8-4.

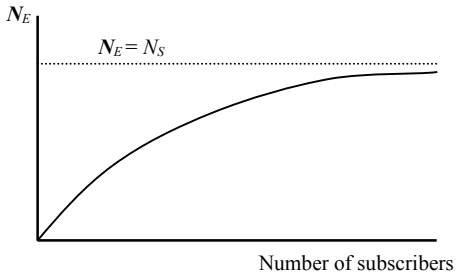


Figure 8-4: Increase of number of subscriptions

The execution time of the event selection algorithm depends on the number of subscription rules. Lets assume that each rule takes a constant amount of time, k ms. With use of grouping of subscribers as discussed in 6.1.2, the execution time will be $k \cdot N_E$, because we have N_E rules. However without use of the grouping of subscribers a rule will exist for each subscriber. In this case, with $N_{subscribers}$ as the number of subscribers, the execution time will be $k \cdot N_{subscribers}$. The comparison is demonstrated in Figure 8-5.

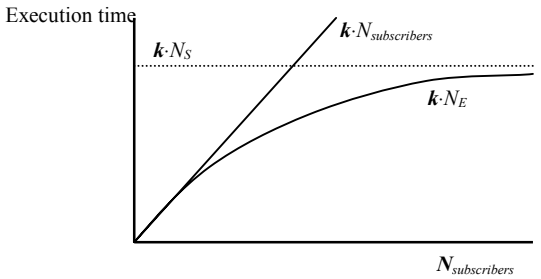


Figure 8-5: Execution time with and without grouping of subscribers

8.2.2 Complexity

The complexity of the algorithms is expressed with the number of rules in the rule-base that are evaluated. First we take a look at the complexity of the subscription algorithm and than we discuss the complexity of the event selection algorithm.

Determining the complexity of the subscription algorithm comes down to a summation of the complexities of the following tasks: determining context value, lookup of priorities in priority matrix, and rule creation. The complexity first task depends on the number of environmental variables M. For each variable a rule has to determine the classified value of the variable. One rule is needed to map all outputs of the awareness modules to a single context value. The number of atoms in a form function determines the complexity of second task. For each atom the priority is looked up in the priority matrix. Since a rule is setup for each priority, the complexity of the last task depends on the number of priorities. The total complexity of the subscription algorithm is thus given by:

$$O(M + N_{atoms} + N_{priorities} + 1)$$ **Equation 8-3**

,with N_{atoms} as the average number of atoms for a form function and $N_{priorities}$ as the number of priorities.
The complexity of the event selection algorithm is easily deducted. When an event notification arrives at the server the agent has to match all the atoms with all the event attributes. When N_{atoms} is the total number of atoms defined in the rule-base, and $N_{attributes}$ is the average number of event attributes, the complexity of the atom matching is given by Equation 8-4.

$$O(N_{atoms} \cdot N_{attributes}) \qquad \textbf{Equation 8-4}$$

Next the subscription rules have to be evaluated. In the previous subsection we stated the execution time of the subscriptions per form function as $k \cdot N_E$. Lets denote the number of form functions in the rule-base as N_{ff}. Consequently, the total complexity of the event selection algorithm is:

$$O(N_{atoms} \cdot N_{attributes} + N_E \cdot N_{ff}) \qquad \textbf{Equation 8-5}$$

Since all the parameters in the parameters in Equation 8-5 are limited by the number of facts asserted in the rule-base the complexity of the event selection algorithm will grow with the size of the fact-base.

9 FUTURE WORK

In this chapter we take a look at useful improvements of the current data distributed agent. The four topics of future work covered in this chapter are fidelity, semantic routing, location aware state merge, and improvement of context-awareness.

9.1 Fidelity

In the introduction of this paper the three dimensions of adaptive data distribution are mentioned. The agent design presented in the previous chapter mainly focused on relevance and timeliness of data distribution. Fidelity is the dimension that is not really covered by the agent.

Integrating a method to also select the level of detail of data in the agent would be a real improvement. When the events are structured as XML the depth of the tree structure will indicate the level of detail. Consequently, the events can be adapted to the wishes of each subscriber. This will reduce unnecessary network traffic even further.

An example of the benefit of adapting event based on fidelity is commanders of different rank that want to receive events in different detail. A team leader wants to see the location of all the men in his team and a higher rank officer wants to see the locations of teams instead of men.

9.2 Semantic routing

To take advantage of the fact that each server in the network has defined a relevance vector, see 7.1.1, the relevance vectors should adapt to the local data demand. In the current implementation the relevance vector of a server is predefined in an XML file. However, the relevance vectors of the subscribers connected to a server determine the kind of data the server should receive. By adapting the relevance vector of a server the routing of data will be based on local needs. This is called semantic routing

A straightforward approach to semantic routing will be that each server calculates the average of the relevance vectors of the connected subscribers and send this to all the neighboring servers. To express the different ranks of the subscribers a weighted average may be more suitable.

Because semantic routing also generates network traffic, the sending of the adapted relevance vector, it should only be adopted if it decreases the overall network traffic. In other words the updating of the relevance vectors should be worth the saved redundant data sending.

9.3 Location aware state merge

When a user moves around his relative location towards other object will change. Because of his new location past events that did not match his subscription may now be selected. A refresh of the user's local repository is needed every time his location changes. In the current implementation refreshing the repository is preformed as

described in 7.1.2. However this is a time-consuming method since all the events have to be checked. A state merge method that takes location awareness into account should be designed.

The trick is to only refresh the events that are selected for the new location. A possible solution is to structure the events in a location-based graph, where each node represents an event and edges exist between the directly neighboring events. We take the new location of the user and start the refresh method with the nearest nodes in the graph. When an event is selected the events of the neighboring nodes are checked. The refresh method stops at a node that is not selected.

9.4 Improve context-awareness

The current implementation focuses on network and location awareness. Another form of awareness that will be useful is client awareness. For example, the data distributed could be adapted to the battery power of clients. Or the agent may adapt the fidelity of events based on the display of the device: there is no need to send 3D graphical information when the client only displays the data in 2D.

9.5 Learning relevance vectors

The users have to define relevance vectors themselves in the current agent. However, by observing user actions the system can be enhanced by a learning mechanism that automatically creates relevance vectors that are representative for the users. The user actions reveal information about the user interests. This information can be feed back to the agent that adapts the relevance vector to this information.

In [22] such an approach is described for users that browse the World Wide Web. The links they choose and the information they read tells the agent about the interests of the user. The same could be applies to the data distribution agent. This method could be additive to the user-defined relevance vector to obtain more accurate adaptation.

10 CONCLUSION

In the introduction of this thesis we described context-aware data distribution and pervasive computing. We stated that the three dimensions of data adaptation are relevance, fidelity, and timeliness. We set the requirements and discussed the challenges of this research. Three important, contradicting requirements were efficiency, expressiveness, and scalability of the data distribution agent. The contributions mentioned were the context-awareness of the data distribution, the expressiveness of the subscriptions, and the prioritization of the distribution. These are all topics that we will review in this chapter.

The agent design we have described in this thesis uses two main algorithms: one for subscription and adaptation, and one for event selection for subscribers. The first algorithm sets up user profiles based on information supplied by the user and contextual information. The second one uses the user profiles to make decisions for data distributions. Instead of boolean decisions the agent maps events per subscriber to priorities. The events are sent in order of priority.

A rule-based approach is used for the data distribution. The subscriptions are saved as rules that are evaluated when event facts are asserted to the fact-base. The rules are adapted according to the changes in the environment of the subscriber.

In the remainder of this chapter we will discuss the issues stated in the introduction and assess to what extend we have succeeded. The issues covered are in order of appearance: the dimensions of data adaptation, the requirements, and the contributions.

10.1 Relevance, timeliness, and fidelity

When we look back at the three dimensions of data distribution adaptation, see Figure 1-2, mentioned in the introduction, we can conclude that the agent presented in this paper operates in the relevance-timeliness plane. The relevance vectors explicitly express the relevance of data. The values of the vector represent the importance of each condition in the subscription, so the agent is able to adapt the subscription, and thus the selection of data, according to the environmental state of the client. The agent addresses timeliness issues by means of prioritizing events. Time critical events will be assigned a high priority and the events without time constraints may be delayed until network traffic decreases. Fidelity is not explicitly supported. However, when data of different levels of detail are sent in separate events, subscriptions that select different levels of detail can be defined.

10.2 Expressiveness and scalability

The focus of the agent is expressiveness and scalability, and less focus on performance. To achieve an expressive subscription language the agent approaches the event selection in a rule-based manner. An expert system is embedded and rules are constructed that represent the subscriptions. Consequently the subscription language supports conjunctive, disjunctive and negative conditions.

Looking at the results of the measurements in 8.1 we can conclude that the system is scalable for an increasing number of events as well as for an increasing number of subscribers. The small overhead in Figure 8-1 and the linearity of the graphs in Figure 8-2 support this conclusion.

10.3 Context-aware data distribution

We stated that in order to support mobility in networking adaptive data distribution in a prerequisite. The reason for this is to avoid unnecessary network traffic. A means of processing context awareness information is therefore needed.

Algorithms are presented in this paper that adapt dynamically to the environment of the users. The algorithms offer a framework to integrate different forms of context awareness based on data dependency. The distinction of data dependency is necessary to process data independent awareness separate from the event selection. This makes the grouping of subscribers possible and will result in a more efficient event selection.

This research has focused on two forms of context awareness: network awareness and location awareness. Both represent a different class of awareness, namely data-independent and data dependent awareness respectively. The network awareness aims at adaptation to the availability of data and location awareness at the usability of data. Together these forms of awareness result in a highly dynamic data distribution agent.

10.4 Prioritized replication

Instead of boolean decisions for data distribution this paper suggests the option of prioritized decisions. The priorities offer a more flexible way of distribution. Information is classified by importance instead of merely selected or not selected. This makes delays of unimportant information possible.

The benefit of prioritized data replication is evident for mobile clients. When network conditions are low the agent focuses only on important data and waits with sending unimportant data until network conditions improve. In this way mobile clients are still able to receive background information, though delayed.

11 REFERENCES

[1] B. van der Poel, Y. Tan, A.M. Krebs, I. Marsic, "Prioritized Replication With Dynamic Selection For Mobile Environments," *Submitted for publication*, 2002.

[2] M. K. Aguilera, R.E. Storm, D.C. Sturman, M. Astley, T. Chandra, "Matching events in a content-based subscription system," *Proc. 18th ACM Symp. Principles of Distributing Computing (PODC)*, 1999.

[3] A. Carzaniga, D.S. Rosemblum, A.L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," *Proc. 19th ACM Symp. Principles of Distributing Computing (PODC)*, 2000.

[4] J. Gough, G. Smith, "Efficient recognition of events in a distributed system," *Proc. ACSC-18*, 1995.

[5] B. Segall, D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proc. Australian UNIX and Open Systems User Group Conference*, 1997.

[6] S. Brandt, A. Kristensen, "Web push as an Internet notification service," *Proc. W3C Workshop on Push Technology*, 1997.

[7] A. Campailla, S. Chaki, E. Clarke, S. Jha, H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," *Proc. Int'l Conf. Software Engineering (ICSE)*, Toronto, Canada, 2001.

[8] M. Ionescu, I. Marsic, "A stateful approach for publish-subscribe systems in mobile environments," *Submitted for publication*, 2001.

[9] T.P. Moran, P. Dourish, "Context –Aware Computing," *Special Issue of Human-Computer Interaction, Volume 16*, 2001.

[10] G. Cugole, E. di Nitto, "Using a publish/subscribe middleware to support computing," *IFIP/ACM Middleware 2001 Conference*, 2001.

[11] Y. Huang, H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Proc. 2nd ACM Int'l Workshop on Data Engineering for Wireless and Mobile Access (MobiDe '01)*, 2001.

[12] C. O'Ryan, D.C. Smith, J.R. Noseworthy, "Patterns and performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, CRL Publishing, 2001.

[13] G.D. Abowd, "Software Engineering Issues for Ubiquitous Computing," *In proceedings of ICSE'99*, 1999.

[14] J. Anhalt, A.Smailagic, D.P. Siewiorek, F. Gemperle, D. Salber, S. Weber, J. Beck, J. Jennings, "Toward Context-Aware Computing: Experiences and Lessons," *IEEE Intelligent Systems*, 2001.

[15] H.A. Jacobsen, "Middleware Services for Selective and Location-based Information Dissemination in Mobile Wireless Networks," *IFIP/ACM Middleware 2001 Conference*, 2001.

[16] L. Capra, W. Emmerich, C. Mascolo, "Middleware for Mobile Computing: Awareness vs. Transparancy," Online at: http://www.cs.ucl.ac.uk/staff/l.capra/hotos.pdf

[17] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, K.R. Walker, "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th ACM Symposium on Operating System Principles*, 1997.

[18] E. Friedman-Hill, "JESS: The Java expert system shell," Sandia National Laboratories, Livermore, CA. Online at: *http://herzberg1.ca.sandia.gov/jess/*

[19] I. Marsic, "Adaptive Collaboration for Wired and Wireless Platforms," *IEEE Internet Computing*, Vol.5, No.4, pp.26-35, July/August 2001.

[20] D. Lopez de Ipina, E. Katsiri, "An ECA Rule-Matching Service for Simpler Development of Reactive Applications," *IEEE Distributed Systems Online*, Vol. 2 , No. 7, 2001.

[21] K. Geihs, "Middleware Challenges Ahead," *IEEE Computer*, Vol. 34, No. 6, June 2001.

[22] J. Goecks, J. Shavlik, "Learning Users' Interests by Unobtrusively Observing Their Normal Behavoir," *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, pp. 129-132, 2000.

[23] T. Kindberg, J. Barton, "A Web-based Nomadic Computing System," *Computer Networks*, Elsevier, vol 35, no. 4, pp. 443-456, March 2001.

[24] N. Davies, H.W. Gellersen, "Beyond Prototypes: Challenges in Deploying Ubiquitous Systems," *IEEE Pervasive Computing*, Vol. 1, No. 1, 2002.

[25] Z. Lei, N.D. Georganas, "Context-based Media Adaptation in Pervasive Computing," *Canadian Conference on Electrical and Computer Engineering*, Vol. 2, 2001.

[26] I. Benyahia, M. Hilali, "An Adaptive Framework for Distributed Complex Applications Development," *Proceedings of 34th International Conference on Technology of Object-Oriented Languages and Systems*, pag. 339-349, 2000.

APPENDIX A: PUBLICATION FOR HICCS 36 CONFERENCE

Prioritized Replication With Dynamic Selection For Mobile Environments

Bart van der Poel[*†], Yingzhen Tan[†], Allan Meng Krebs[†], and Ivan Marsic[†]

[†]*Center for Advanced Information Processing (CAIP)*
Rutgers — The State University of New Jersey
Piscataway, NJ 08854-8058 USA
+1 732 445 4208

[*]*Delft University of Technology*
Zuidplantsoen 4
2628 BZ, Delft, The Netherlands
+31 15 278 7504

{bart, krebs, marsic}@caip.rutgers.edu, tan@paul.rutgers.edu

Abstract

The proliferation of small mobile devices and wireless networks has resulted in an increasing demand to support the applications found in wired environments on mobile wireless devices. In real-time replication systems, such as collaborative systems, this trend gives some new problems to address. The properties of wireless networks are low bandwidth and high latency, which change dynamically over time. The risk of the network getting congested is therefore high with the result that the user will not receive the important information in time. Consequently there is a need to develop algorithms and methods for adaptive data distribution, to get the relevant data to the user at the right time, with the right fidelity. This paper presents an algorithm and methods for intelligent data distribution for both publish/subscribe and peer-to-peer systems using dynamic selection rules. The generation of the selection rules is based on the status of the computing and communication resources. The rules enable prioritizing of the data being delivered to the clients. By sending data in order of priority, important data is sent immediately and unimportant data is sent when conditions improve. An example system was implemented using an expert system as a decision maker supporting both the publish/subscribe model between clients and server and the peer-to-peer model among multiple servers. The decision maker in the example system adapts to the available bandwidth. The evaluation of the example system shows satisfactory scalability of the algorithm.

Keywords: Mobile computing, data replication, rule-based systems.

1 INTRODUCTION

Currently networks are increasingly supporting mobility. Wireless networks make the replication of data objects in nodes more complex, because the accessibility of the mobile nodes depends on the environment it is in. Dynamic properties, such as bandwidth and position, influence the way data is distributed to mobile clients. With low bandwidth a user may want to receive only high priority information, while he is also interested in background information should the available bandwidth increase. Therefore, to support mobile clients, a context aware replication algorithm is needed.
Important dimensions of data adaptation are relevance, fidelity, and timeliness (see Figure 1), where: (*i*) relevance is determined by user's interests and priorities; (*ii*) fidelity is dictated by computing platform's capabilities; and (*iii*) timeliness is determined by the requirements of the task.

The user provides the relevance information to state his interests, the device and the application determine the fidelity of the data, and timeliness issues time constraints to data delivery, set by the requirements of the tasks. Evaluation of data with respect to the three dimensions results in a priority. This priority represents the relevance of data to the user in a certain environmental state. The more relevant data is, the higher its priority will be, and the more certain it is to be sent. With different priorities for levels of details we can select at what level of detail data should be sent. Sorting data by priority before sending addresses the timeliness: high priority data is sent immediately and low priority data is sent later when network conditions improve.
An example application in the real world could be a military application for situational awareness on the

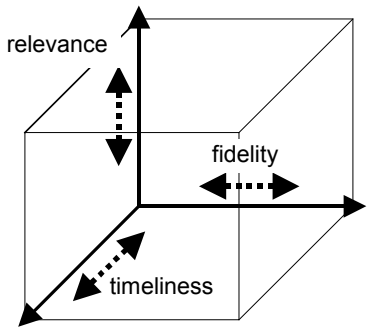


Figure 0-1: Dimensions of data adaptation for quality of service (QoS).

battlefield. Commanders, equipped with wireless communication devices, have to base their decisions on the received data about friendly and hostile units. The amount of data they are able to receive is limited by the bandwidth. The relevance of data will be different for each commander based on his task, and the fidelity of data depends on the level of command. For example, a squad or platoon leader with an offensive task wants to be informed of the positions of the men in his unit and targets in the nearby area. A higher-level commander would be more interested in receiving information about positions of units than of men. The timeliness of information about tanks will be more important to commanders than the timeliness of information about infantry, because tanks can move faster.
We present a prioritized replication algorithm with dynamic selection. The criteria used for selection depend on the environmental variables, and is therefore dynamic. When these variables change the selection criteria have to adapt to the situation. As we mentioned above, this is motivated by the increasing mobility of network nodes.
Other algorithms replicate data based on a Boolean selection; instead our approach uses priorities, returned by the selection criteria, to replicate data. The benefit of this approach is that data is sent in order of importance to the receiver. Sending of less relevant data may be postponed until conditions improve.
We applied our prioritized replication algorithm on a publish/subscribe system with distributed servers. The servers replicate data to each other using a peer-to-peer epidemic algorithm. In an epidemic algorithm each peer propagates data it receives.

The next section of this paper reviews the related research. We continue with describing the selective and prioritized replication algorithm. The fourth section presents our algorithm and serves as an example application. Then we evaluate the performance of the implemented system by discussing the result of measurements we performed. Finally, we conclude the paper and discuss the future work.

2 RELATED WORK

First we discuss similar research in publish/subscribe systems that focus on the selection algorithm. Next related peer-to-peer systems are discussed. The discussion of peer-to-peer systems is more concerned with the replication algorithm. We conclude this section by comparing our system with the other systems mentioned here.

2.1 Publish/Subscribe

Publish/Subscribe systems can roughly be divided in two groups: subject-based systems and content-based systems. The systems select notifications for subscribers by subject respectively by content. This research is focusing on the content-based systems. Selection in these systems is more flexible and more precise, because subscribers can apply many-dimensional criteria instead of choosing between pre-defined groups.
The best-known content-based systems are GRYPHON [1], SIENA [3], ELVIN [5],[4], and KERYX [6]. The selection algorithm for GRYPHON emphasizes efficiency and scalability. It uses matching trees, which make its time-complexity sub-linear with the number of subscriptions. A drawback of this algorithm is the limited expressiveness: subscriptions can only be defined with conjunctions. In [7], a similar approach is suggested, using Binary Decision Diagrams, with a richer language for subscriptions. Expressiveness and scalability are the main interests for the SIENA selection algorithm. For this it uses covering relations, which are partial orderings with respect to subsumption. Though the expressiveness of subscriptions is still limited to conjunctive patterns. ELVIN uses the most expressive of the above-mentioned algorithm. It supports first order logic patterns, and regular expressions for selecting strings. The algorithm in the KERYX system is expressive, but not very efficient. It uses a LISP-like filtering language.
Development of mobile devices demands Publish/Subscribe-systems not only to select messages with content-based attributes, but also with attributes based on the client's environment. Attributes that define computing capabilities of devices and the quality of the connection should be considered as well. All systems mentioned above do not support this feature. [8] Proposes a stateful approach that does take this feature into account. The algorithm selects events with the conditions set by the user and the client state. The conditions are compiled at runtime. However the work in [8] does not have the expressive power comparable to the one showed in this paper and does not use priorities.

2.2 Peer-to-Peer

Systems based on the traditional peer-to-peer model [7],[18] allow direct communication and synchronization between all participants, but exhibit scaling problems. Scaling problems exist because they have been implemented by replicating all relevant information to every participant in the system without any discrimination among the participants. For example, BAYOU [7] supports mobility based on the peer-to-peer model. It allows direct any-to-any communication, but it tries to replicate the whole database on every nodes regardless the physical condition of the nodes and the network. ORACLE [16],[7] tried to avoid the scaling problem by using multimaster-slave model. The idea is similar to ours. The assumption is the number of masters will be much smaller than the total number of users. Only replication among masters is peer-to-peer. However, ORACLE's replication algorithm is not 'epidemic', which means that updates can only be propagated by the server that originates those actions. This is based on a too good assumption for network connectivity, which is not so real in wireless environments.
Neither BAYOU nor ORACLE provides selective replication. CODA [14] provides selective replication at the clients, but not at the replicated servers, which are traditional peers. CODA has prioritized replication, but the idea is different to ours. Their prioritized replication means the system has two replication schemas: first-class and second-class. The first-class schema is for replication among servers and the second-class schema is for replication between clients and servers. The whole algorithm is static, when to use which schema is determined at the system start-up time. Because in CODA architecture, servers are connected, the first class replication always assumes a good bandwidth and connectivity. Disconnected operations and

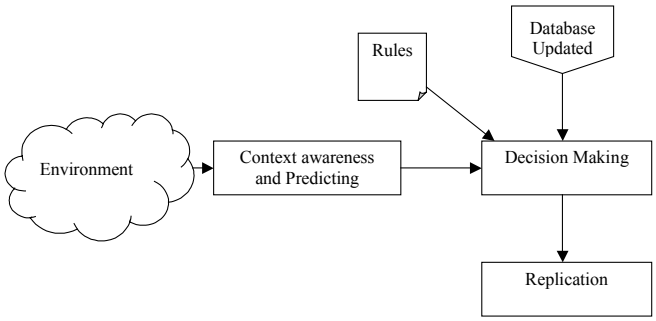


Figure 0-2: Framework for selective replication.

algorithms are only considered in the client-server replication. No priority is considered, however, within the disconnected operation algorithm.

Though some other systems (FICUS [17], RUMOR [11]) have peer-to-peer replication and selective replication features like CODA, some system like Lotus Notes [15] even allow the user to define certain rules for replica selection. None of them provide prioritized replication with dynamic selection, based on network awareness.

2.3 Comparison

In contrast with these systems, the selection algorithm in our work results into priorities, based on current network conditions. These priorities are used for the replication of messages. The advantage of a prioritized approach is that less relevant messages can still be sent when the network is idle. Instead, all systems mentioned above make boolean decisions about whether a message has to be replicated. If a message is not sent because of current conditions, it will never be sent, even when the network conditions improve.

3 SELECTED AND PRIORITIZED REPLICATION

3.1 General Framework

A design of a prioritized algorithm with dynamic selection is composed of several major components. The components and their interrelationship shown in Figure 2 is a generalized picture of how such replication algorithms work. This figure gives a big picture of the process flow after a database-updated event is triggered and how different components interact with each other. A replication algorithm does not necessarily implement all components. In fact, most do not. However, if algorithms follow a well-defined interface standard when programmers developing components, then later another third party or custom made component can be plugged in to enhance the algorithm functionality. All currently existing algorithms can be categorized into four classes, dependent on the components implemented and their functionality.

1: Pure Replication

Pure replication algorithms implement the replication component only. After a database-updated event is triggered, the algorithm replicates the updates to the clients immediately, without any selection or priority taken into consideration. The replication component may use publish/subscribe communication model, peer-to-peer model or both. In essence, pure replication algorithms do not have selective replication at all.

2: Conditional Replication

One step further than pure replication algorithms is to implement a decision-making component with fixed conditions in addition to the replication module. After a database-updated event is triggered, the algorithm first decides whether or not this update should be propagated to the clients and if so, it sets the priority of this update. Then based on the final decision, the replication component will ignore the update, propagate it immediately, or postpone the propagation to a certain time.

3: Rule-Based Replication

Adding rules to the conditional replication algorithm enhances it further compared to the basic conditional replication algorithms. A typical algorithm provides default conditions for decision-making. In addition, it

also allows users to set up the rules themselves. Hence, the decision-making module will receive a rule file as an input, and make decisions based on this rule file the user has provided. Most existing selective replication algorithms fall into this category. Users are provided the opportunity to write a rule file telling the algorithm what kind of data they want to have replicated. The rule file is maintained in the users home server and can be retrieved by other servers before a replication if required. Rule-based replication algorithms are more flexible than the simple conditional replication algorithms in the sense that users can control the replication process at their own will.

4: Dynamic Rule-Based Replication

Although rule-based replication algorithms provide the users nice control over the replication, they still have drawbacks. Because the only input for the decision-making module is the rule file defined by users when they connect in, decisions cannot be made according to the dynamic environmental state. It is important, however, to make different decisions in different states. For example, a user may want some background information to be sent when the network bandwidth is good, but may like to conserve the bandwidth for more important information if connectivity is poor. Thus, we need another input besides the static rule file to reflect the dynamic changing network. Adding another component for environment awareness and prediction does this. This component collects the real time information about the environment (e.g. available network bandwidth and latency, context and location) and provides it to the decision-making module.

This paper presents a dynamic rule-based replication algorithm. All modules of figure 2 are included in our algorithm.

3.2 Decision Making

For the decision-making module of our algorithm, two parts are distinguished: the subscribing part and the selection part. The first part establishes a connection and sets up the subscription for the connection. The selection part handles incoming data objects and distributes it to the subscribed connections.

First we describe how clients subscribe to the server. At the server side a set Φ of atoms is defined of all the atoms that may appear in a subscription. Each atom is a tuple $\phi = (name_\phi, operator_\phi, value_\phi)$, and a subscription is a logical expression of these atoms. When the subscription language supports not only expressions in conjunctive form, but also allow disjunctive logical connectives, the subscription should also describe the form of the logical expression the atoms should be interpreted in. For example the atoms in Table 1 could be of the form: (ϕ_0 OR ϕ_1) AND ϕ_2. We define a form function Γ, as the application of the form to a set of atoms, so $\Gamma(\Phi)$ is the total subscription of all atoms.

Table 2: Example of Φ.

Id	Name	Operator	Value
ϕ_0	company	=	IBM
ϕ_1	company	=	DELL
ϕ_2	price	≤	100

The state diagram in Figure 3 shows the process of subscription. Each connection subscribing to the server will follow these states. The process is context aware; it monitors environmental variables and adapts subscriptions accordingly.

For establishing a connection the connecting party has to send a relevance vector *r*. The relevance vector for a connection specifies how the subscription should be interpreted under different environmental conditions. Each element in the vector, $r(\phi)$, assigns the importance of each atom in Φ for a connection. The possible values for $r(\phi)$ should be in the discrete set ρ. The relevance values for the atoms are used further in the process when we determine the priority values.

The second step is to measure or predict environmental variables. This information makes the method context aware. If we use *M* different environmental variables the state of the connection is defined in a *M*-dimensional context space, called *C*. This space maps the *M*-dimensional state to a context value *c*. The context value should be an element of the discrete set χ of all possible context values, so $c \in \chi$. The context of the connection should be monitored all the time. When a change in the environment state is registered,

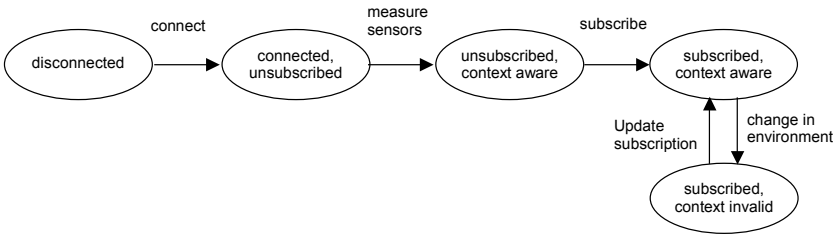


Figure 0-3: State diagram of context aware subscription.

the context the subscription is based on is invalid, and the subscription has to be updated. Now we know the relevance values for each atom and the context value of the state, we can assign subscriptions. For each priority value, a subscription is defined for each connection. So when N different priority levels are considered, we have N subscriptions for each connection, associated with the corresponding priority. Let π be the discrete set of possible priority values. A priority matrix P is used to assign the priority to each atom in Φ. The rows of the matrix represent the elements of χ, and the columns the elements of ρ. Each entry in the matrix is the priority $p \in \pi$, for the particular state and relevance value. The N prioritized subscriptions can now be generated by first determining the set of atoms for each priority level, and then apply Γ to the set. The set of atoms for priority level p and its prioritized subscription are given by the following two functions:

$$\Phi^{p}_{sub}(c) = \{\phi \,|\, P(r(\phi), c) \geq p\} \qquad (1)$$

$$s_p(c) = \Gamma(\Phi^{p}_{sub}(c)) \qquad (2)$$

Atoms that are used in the subscription with a higher priority are also used in the subscriptions with a lower priority, so the latter is less restrictive.

Table 3: example of priority matrix, with *N*=3.

		Relevance value *r*(ϕ)		
		LOW	MEDIUM	HIGH
Context value *c*	LOW	LOW	MEDIUM	HIGH
	MEDIUM	MEDIUM	HIGH	HIGH
	HIGH	HIGH	HIGH	HIGH

The assumption is made that disjunctive operators connect the atoms that differ between the prioritized subscriptions for a connection. The reason behind this lies in the fact that conjunctive operators connect the atoms of different attributes and should all appear in the prioritized subscriptions. It is important that this assumption is kept because it ensures that the higher priority subscription is more restrictive. The relevance vector should enforce this property.

The selection process is much simpler. Updates in the data collection trigger the process to decide if the update should be replicated to other connections, which may be with peers or with clients. The subscriptions select data for the connections, based on the contents of data objects. In this way the server will only replicate data for the interested parties, which saves unnecessary network traffic. The contents of data are represented by notifications. A notification is a conjunction of name value pairs $\alpha = (name_\alpha, value_\alpha)$, that define the content of the data.

Table 4: Example of notification.

Name	**Value**
Company	IBM
Price	85

The process first matches all the atoms with the notification name-value pairs. An atom ϕ in a prioritized subscription $s_p(c)$ is true for a certain notification if and only if $name_\alpha = name_\phi \wedge operator_\phi (value_\alpha, value_\phi)$ [3]. All atoms that do not match any notification name-value pair are false. Next we see if the logical expression of the application of Γ to the atoms results to true. If it does, we say that subscription $s_p(c)$ fires.

When $s_p(c)$ fires, priority p is returned for the data object for the connection. More prioritized subscriptions may fire for a connection; in this case the value representing the highest priority is returned. The priorities for all users are returned as a mapping from user to priority.

3.3 Context Awareness and Prediction

The context awareness and prediction module provides dynamic environment status to the decision-making module. This is especially important for networks that support mobility, where for example network and location awareness influence the way data should be distributed. An important consideration in designing context awareness for pervasive computing is that the cost of acquiring context awareness does not exceed its utility: if not obtained efficiently, context awareness could decrease performance. Several approaches could be used to measure or predict the network status, see for example [6]. A discussion about these approaches goes beyond the scope of this paper.

The purpose of the context awareness module is to supply the state of the environment of a node in the network to the decision-making module. This state is derived of sensor values that measure characteristics

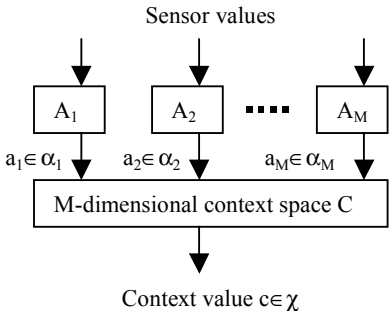


Figure 0-4: Outline of context awareness module.

of the environment. These values are the inputs to the M different awareness modules A_i that reason about the quality of the inputs and return this quality value a_i. For example, awareness module A_1 may be about network awareness and uses the inputs bandwidth and packet round trip time. The possible values for its output may be α_1={HIGH, MEDIUM, LOW}. When bandwidth is high and the variance of the round trip time is low than the value of a_1 is probably HIGH.

All the values a_i are used to find the context value c that represents the current state of the environment. This context value is found in the pre-defined M-dimensional context space C. This space maps each vector of awareness values to a context value. So the context value is given by the following formula.

$$c = C(\underline{a}) \qquad (3)$$

It should be clear that context information is independent of values in the notifications. So if location awareness is used in the context module this means awareness of location information about the node in the network. When notifications contain location aware information this should be handled in the subscriptions.

The reason for applying context awareness in the subscription part instead of the selection part of the decision-making module is performance. In the selection part it is important to reduce the overhead as much as possible, so that data distribution is efficient. By considering the context awareness during generation of the selection rules, in the subscription part, only application of the rules is needed when data arrives at the server. The actual adaptation of the subscription may be done when the server is idle.

3.4 Replication

The replication module is responsible for replicating data between different replicas. Depending on the result it receives from the decision-making module, the replication algorithm may choose one of the following actions: discard the data update, send the update immediately or send the update later. The third action is chosen when the update has a lower priority but not low enough to ignore it. The actual communication mechanism may vary among different systems. For example, we can use publish/subscribe or peer-to-peer.

The peer-to-peer model does not have the single point failure problem as the publish/subscribe model does. However, the tradeoff is that it is less scalable than the publish/subscribe model. Because temporary disconnection may occur, each node has to maintain a list per peer to remember the updates needed to

inform that peer. To use the prioritized replication, these lists need to provide some way to support retrieving by order of priority. Epidemic and optimistic algorithms should be used in peer-to-peer replication in order to remove the single point failure problem and increase performance [19]. Conflict resolution method, either provided by the system or by the user, is called during the replication when two disconnect nodes update the same data separately and both try to propagate the updates. Disconnect operations are needed on each node to support the ad-hoc, partitioned network environment.
The publish/subscribe communication paradigm allows suppliers of data objects and receivers thereof to communicate asynchronously. Both publishers and subscribers do not have to know anything about the other party. When the server receives a data object from the publisher it decides to which subscribers it will forward the message. The decision is based on the subscriptions the possible receivers have defined. A user subscribes to the server by sending the relevance vector. Next the server sets up the prioritized subscriptions for the user as described above. The server updates the local data repository of the client by sending all data that are selected by the user's subscriptions.

4 IMPLEMENTATION

In this section we discuss how the algorithm described above is implemented in our example system. We show how the subscription information is stored, how the decision-making agent works, and how data is replicated.

4.1 XML Subscription

The subscription information is defined in an XML file at the server side. An example of such a file is shown in Listing 1. The subscription is structured like a tree with logical operators as nodes and atomic expression as leafs. The tree-structure of logical operators is the Γ function, introduced in the section about decision-making, and the atom elements form the set Φ of atoms. Thus the total subscription file represents $\Gamma(\Phi)$. This is the structure users may subscribe to by sending relevance vectors r. The size of the vector must equal the number of atom elements in the subscription XML file. Each element in the relevance vector describes the importance of an atom for the user. When the value of relevance vector is equal to zero the user is not interested in the atom at all and it will not appear in the user's prioritized subscriptions.

Listing 1: Example of the XML subscription file.

```
<SUBSCRIPTION>
  <LOGICAL_OPERATOR value="OR">
    <LOGICAL_OPERATOR value="OR">
      <ATOM name="type" operator="=" value="document"/>
      <ATOM name="type" operator="=" value="time"/>
      <ATOM name="type" operator="=" value="overlay"/>
      <ATOM name="type" operator="=" value="chat"/>
    </LOGICAL_OPERATOR>
    <LOGICAL_OPERATOR value="AND">
      <ATOM name="type" operator="=" value="unit"/>
      <LOGICAL_OPERATOR value="OR">
        <ATOM name="affiliation" operator="=" value="H"/>
        <ATOM name="affiliation" operator="=" value="F"/>
        <ATOM name="affiliation" operator="=" value="N"/>
      </LOGICAL_OPERATOR>
    </LOGICAL_OPERATOR>
  </LOGICAL_OPERATOR>
</SUBSCRIPTION>
```

Suppose we use the subscription of Listing 1, and the priority matrix of Table 2. When the context value is low and we summit the relevance vector (`H M M L H H M L`), the server will setup the following prioritized subscriptions for the user.

Table 5: Example prioritized subscriptions.

Priority	Subscription
LOW	type $\in$ {document, time, overlay, chat} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H, F, N})
MEDIUM	type $\in$ {document, time, overlay} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H, F})
HIGH	type $\in$ {document} $\vee$ (type=unit $\wedge$ affiliation $\in$ {H})

When the quality of the connection or the relevance vector changes over time the prioritized subscriptions for the connection are adjusted to the new values.
As the users subscribe, a lot of subscriptions will overlap. For performance reasons, connections share subscriptions. The sharing connections with according priorities are registered for each subscription. In this way the system needs to evaluate less subscriptions.

4.2 Rule-Based Distribution Agent

A rule-based distribution agent matches incoming data objects to the prioritized subscriptions. The agent utilizes an embedded expert system for the matching process. We use the Jess Expert system, because it is easily integrated in the existing Java software and it uses the efficient Rete algorithm for rule matching [8]. Data objects are the input of the expert system and a mapping of connections to priorities are the output. The subscriptions are the rules the inference engine of the expert system uses to decide to whom and with what priorities a data object should be forwarded.
When a server is started it initiates two distribution agents: one for the clients connected to the server, and one for the neighboring servers. We distinguish these two agents because the replication for clients and the replication for peers are executed by two different threads. A database update triggers both agents to decide to which connections the update has to be replicated to.

4.3 Replication Algorithm

In our example system replications among servers are done in peer-to-peer mode. Upon starting, each server will receive configuration information from a configuration file, including the relevance value for each condition described in the subscription file. Currently, neighbor information like neighbor server address and port number is retrieved from server configuration file too. However, a slight modification can make the server dynamically recognize neighbors by using multicasting messages. The server will open a channel listening for data updates propagated to it from other servers. When an update is made on the local repository, either by a client connected to the server or by another neighbor server, the update will be put into dirty queues for each neighbor server and separate threads are waken up trying to propagate the update. Those threads will ask the Jess expert system for the priority of a certain update and then decide how to handle it. As we discussed before, the update will be discarded, sent as soon as possible or postponed to send later. If an update should be sent later, it is moved into some secondary dirty queue based on its priority. When all updates with high priorities are sent, updates in the secondary dirty queue will be sent if the network connection allows it. Below is an example code for the propagation method:

Listing 2: Example code for replication algorithm.

```
for (int priority=0; priority<numberOfPriorities; priority++) {
        for (int i=0; i<dirtyQueue[priority].size()) {
                Update update = dirtyQueue[priority].next();
                //determine the uform sending priority, if it's 0 send immediately
                if (priority ==0) {//initially, all dirty uforms are in priority 0 queue
                        int newPriority = getUFormSendingPriority(update);
                        if (newPriority != 0) {
                                dirtyQueue[priority].remove(update);
                                if (newPriority > 0) //-1 is no replicate
                                        dirtyQueue[newPriority].add(update);
                                continue;
                        }
                }
                // compare the neighborVector with update.version,
                // send if update version is later than the neighbor's
                if (newerThan(update.getVersionVector(), neighborVersionVector)) {
                        con.send(update);
                }
         }
}
```

The interesting part is that the propagation method in listing 2 maintains an array of lists. Each list corresponds to a dirty queue with a specific priority. Initially, all updates are in the priority 0 queue. Before sending, this thread asks the decision-making module to decide the priority of the update and acts

accordingly. Upon environmental change, the decision-making module will re-evaluate all dirty updates again. In that case, the sending thread will get notified. The current loop will start all over to send updates from the highest priority queue again.

The replication algorithm used in the peer-to-peer model is epidemic and optimistic. Each server will try to propagate an update to other servers but the server that it gets the update from. There is no single point failure problem. The tradeoff is that it may incur unnecessary network traffic. In order to reduce the network traffic as much as possible, a global version vector is used for each server that records the updates seen by the server as a whole and propagates updates that a neighbor server has not yet seen. The global version vector is only updated when all updates are received from a neighbor server. Working perfectly well in good network condition, it still cannot totally eliminate redundant traffic in poor network condition unfortunately.

Optimistic replication algorithms apply weak-consistency restrictions and allow access to replicas and propagate updates to other replicas in the background. By contrast, a conservative or pessimistic algorithm does not allow multiple writers when the writers cannot directly communicate to serialize the order of operations. Thus, an optimistic approach can greatly improve the data availability in wireless environments, where unreliable connection and low bandwidth are common cases and somewhat stale replicas are more acceptable by users than frequently long delays in data access. Version vectors at data item level are used to decide update sequences and detect conflicts. Each client is assigned a priority level and he or she will stamp the priority level on all updates he or she makes. Upon a confliction, the update with higher priority wins. This is a default conflict resolution approach. If the user likes, he or she can also plug in a user-defined conflict resolution method as long as the method follows the same interface. Strategy design pattern [9] is used here so that the new conflict resolution method can be plugged in without modifying the rest of the code.

In the publish/subscribe model the replication algorithm does not have to worry about any version vectors. Clients connect to the server by sending a relevance vector, and all objects in the server's repository that are selected by a client's subscription are replicated to its local repository. As long as the client is connected to the server, it receives all data updates at the server that it subscribed to.

5 MEASUREMENTS

In this section we discuss the results of measurements we have taken with our algorithm. We show the overhead the scalability and the performance gain of both the peer-to-peer model as the publish/subscribe model.

5.1 Overhead

We have measured the overhead of prioritized replication in peer-to-peer communication model. In this measurement, we recorded the time used for data to be transmitted from one server to another server. The experiment is done on a LAN with two Pentium III severs. Two sets of experiments were done. First we transmitted data without any priority decision. Then we determined the data priorities and transmitted data according to their priorities. We can see in figure 5 that the transmission times go linearly in both cases. The overheads for the priority decision go linearly to the number of data items we sent. The reason is clear: since each data item needs to be assigned a priority, the more data items we want to send, the more time we spend in the decision-making module. However, the overheads will be generally within 10% of the total time we need. Interestingly, the overheads for the first two columns were negative. We believe this is because the overhead in these cases is small enough to be flatted out by the measurement error under this number of data items.

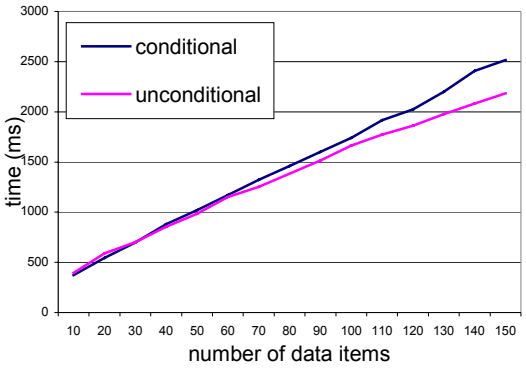


Figure 41: Overhead of conditional replication for the peer-to-peer model.

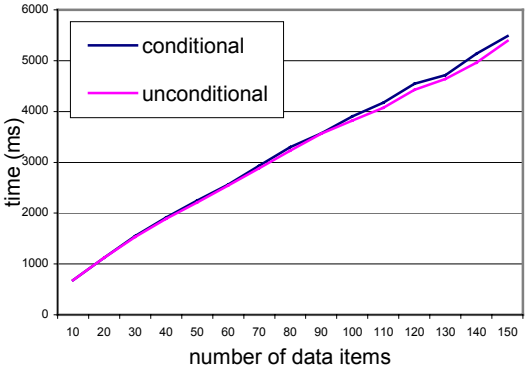


Figure 41: Overhead of conditional replication for publish/subscribe model.

Also for the publish/subscribe model the overhead is measured, see figure 6. The time measured for the conditional and unconditional replication is the time between the moment the data items are submitted and the time a fixed number of clients receive them. The linearity of both the conditional and the unconditional is clear. The overhead is constant at approximately 1.5%.

5.2 Scalability

Figure 7 shows the scalability for the publish/subscribe model over an increasing number of users. We again measure the time it takes between sending a data item and receiving it by all other users. The measurements are done for one, two and three servers. The graphs for all three are linear, though the slope decreases with an increasing number of servers. The graphs show us that for less than 25 users two servers are preferred and for more than 25 users a configuration of three servers performs better. It is to be expected that for a high number of servers the graph will be almost flat. This means the increase of users does not influence the time when we use many servers.
In Figure 8, we measured the time used to replicate updates among servers. With the number of servers increasing, the total time needed goes almost linearly. Figure 8 shows that the scalability of our system over increasing number of servers is satisfactory, at least for ten servers.

In Figure 9 we measured the benefits of prioritized replication. The scenario is as follows: first we have 60 data item updates, among which an increasing percentage of the items should have high priority. The time measured starts when we send the first data item and ends when the last high priority data item is received. The order in which high and low priority updates are sent is random, but obviously we have best and worst case. The best case is that all high-priority updates are in the very front and if one of them appears as the last update, it becomes the worst case. If we are using unconditional sending, all updates will have the same priority and send by time order, this is shown by the yellow line. The time measured for the unconditional replication starts when the first data item is sent and ends when the last data item arrives. We can see from the figure that if high priority updates are about less than 75%, we will have a performance gain. The performance gain goes down dramatically when the high priority updates are under 50%.

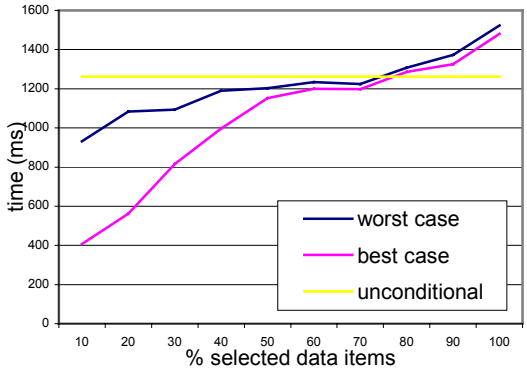


Figure 41: Performance gain of conditional replication for peer-to-peer.

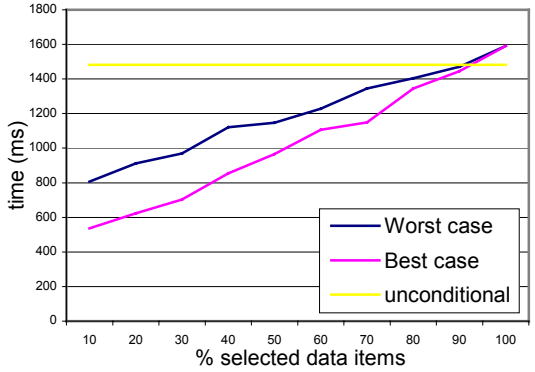


Figure 41: Performance gain of conditional replication for publish/subscribe.

The same measurements are performed for the publish/subscribe model. The results, shown in figure 10 are comparable with the results for the peer-to-peer model. Though, the slopes of the conditional graphs are more linear. The conditional replication algorithm performs better for situations where less than 90% of the data items are selected.

6 CONCLUSION AND FUTURE WORK

In this paper we presented an algorithm for selective and prioritized replication with dynamic selection. The results of the measurements show us that the algorithm is scalable and produces little overhead. The fact that the algorithm is context aware makes it more suitable to networks with mobile nodes than traditional methods of replication. Prioritized subscriptions extend traditional Boolean subscriptions and give subscribers more control over the way the want to receive data.

The algorithm presented addresses all three dimensions important to data adaptation mentioned in the introduction: relevance, fidelity, and timeliness. Relevance of data is explicitly expressed in the relevance vector. Subscriptions may contain conditions that select certain levels of detail. The timeliness is addressed by the sending of data in order of priority.

In our current work we focused on the network awareness. For future work we will do more research in location awareness. Processing both geographical location of the user as well as location information in data objects will be of interest to our work. Another part of future work will be in the expressive power of the subscriptions in out system. Currently the subscription language supports disjunctions and conjunctions, but other logical operators, such as negation, will be useful.

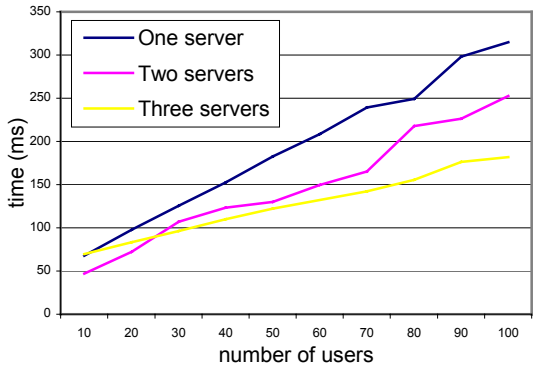


Figure 41: Scalability of publish/subscribe over an increasing number of users.

Figure 4: Scalability of peer-to-peer over increasing number of servers.

7 ACKNOWLEDGMENTS

The research reported here is supported in part by NSF Grant No. ANI-01-23910, US Army CECOM Contract No. DAAB07-02-C-P301 and by the Rutgers Center for Advanced Information Processing (CAIP).

8 REFERENCES

[1] M. K. Aguilera, R.E. Storm, D.C. Sturman, M. Astley, T.Chandra, "Matching events in a content-based subscription system," *Proc. 18th ACM Symp. Principles of Distributing Computing (PODC)*, 1999.

[2] M. Altinel and M. J. Franklin ,"Efficient filtering of XML documents for selective dissemination of information," *Proc. 26th VLDB Conference*, 2000.

[3] S. Brandt and A. Kristensen, "Web push as an Internet notification service," *Proc. W3C Workshop on Push Technology*, Boston, MA, Sep. 1997

[4] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision diagrams," *Proc. Int'l Conf. Software Engineering (ICSE)*, Toronto, Canada, 2001.

[5] A. Carzaniga, D. S. Rosemblum, and A.L. Wolf" "Achieving scalability and expressiveness in an internet-scale event notification service," *Proc. 19th ACM Symp. Principles of Distributing Computing (PODC)*, 2000.

[6] L. Cheng and I. Marsic, "Piecewise network awareness service for wireless/mobile pervasive computing." *MONET*, 7(4):269-278, 2002.

[7] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch, "The Bayou architecture: Support for data sharing among mobile users," *Proc. Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, Dec. 1994.

[8] E. Friedman-Hill, "JESS: The Java expert system shell," Sandia National Laboratories, Livermore, CA. Online at: *http://herzberg1.ca.sandia.gov/jess/*

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

[10] J. Gough and G. Smith, "Efficient recognition of events in a distributed system," *Proc. ACSC-18*, Adelaide, Australia, 1995.

[11] R. Guy, P. Reicher, D. Ratner, M. Gunter, W. Ma, and G. Popek, "Rumor: Mobile data access through optimistic peer-to-peer replication," *Proc ER'98 Workshop on Mobile Data Access*, 1998.

[12] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Proc. 2nd ACM Int'l Workshop on Data Engineering for Wireless and Mobile Access (MobiDe '01)*, Santa Barbara, CA, pp.27-34, 2001.

[13] M. Ionescu and I. Marsic, "A stateful approach for publish-subscribe systems in mobile environments," Submitted for publication.

[14] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, 10(1):3, February 1992.

[15] Lotus Development Corporation. "Lotus Notes: Essential software for group communications." Lotus Notes Technical Series Vol. 1, December 6 1989.

[16] Oracle 7 Distributed Database Technology and Symmetric Replication. Oracle White Paper, April 1996.

[17] T. W. Page, Jr., R. G. Guy, G. J. Popek, J. S. Heidemann, W. Mak, and D. Rothmeier. "Management of replicated volume location data in the Ficus replicated file system." *Proc. USENIX Conf.*, Los Angeles, CA, pp. 17-29, June 1991.

[18] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner, "Peer-to-peer reconciliation based replication for mobile computers," *Proc. ECOOP Workshop on Mobility and Replication*, July 1996.

[19] Y. Saito, "Optimistic replication algorithms," Tech. report, University of Washington, August 2000.

[20] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proc. Australian UNIX and Open Systems User Group Conference*, 1997.

APPENDIX B: RULEBASE

```
; author  : Bart van der Poel
; date    : 04/09/2002
; descr.  : Rule-base for adaptive message distribution agent. Return mapping of users to
;           priorities.

; #### TEMPLATES ##############################################################################

(deftemplate bandwidth (slot value))
(deftemplate relevance (slot id) (slot value))
(deftemplate connection (slot connId))
(deftemplate form_function (slot id) (multislot nodes) (multislot branches)
          (multislot connections))
(deftemplate atom (slot id) (slot name) (slot operator) (slot value) (slot include)
          (slot satisfied))
(deftemplate message (slot name) (slot value))
(deftemplate relevance_quality (slot id) (slot value))
(deftemplate bandwidth_quality (slot value))
(deftemplate new_subscription (multislot atoms) (slot connection) (slot priority))
(deftemplate subscription (slot id) (multislot atoms) (multislot connections) (multislot priorities)
          (slot form_id))
(deftemplate result (multislot connections) (multislot priorities))
(deftemplate sub_count (slot value))
(deftemplate pred_count (slot value))
(deftemplate number_of_priorities (slot value))
(deftemplate formfunction_connection (slot formfunction_id) (slot connection_id))
(deftemplate uform (slot id) (slot type) (multislot keys) (multislot values))
(deftemplate parameter (slot id) (slot name) (multislot attributes) (multislot values))
(deftemplate location (slot id) (slot x) (slot y) (slot z))
(deftemplate distance (slot id) (slot param1) (slot param2))
(deftemplate count (slot id) (slot param))

; #### SYSTEM VARIABLES ########################################################################

(bind $?nodes (create$ ))
(bind $?branches (create$ ))

; #### INITIALIZATION ##########################################################################

(assert (sub_count (value 0)))

; #### SUBSCRIPTION MANAGEMENT #################################################################

(defrule addSubscription
    "If the new subscription does not yet exist, then create the new subscription and add the
     connection with its priority to it"
    ?new <- (new_subscription (atoms $?atoms) (connection ?connId) (priority ?pri))
    ?sub <- (sub_count (value ?count))
    (form_function (id ?formId) (nodes $?n) (branches $?b) (connections $?connections))
    (not (subscription (atoms $?atoms) (form_id ?formId)))
    (test (neq (member$ ?connId $?connections) FALSE))
    =>
    (assert (subscription (id ?count) (atoms $?atoms) (connections (create$ ?connId))
        (priorities ?pri) (form_id ?formId)))
    (bind $?nodes $?n)
    (bind $?branches $?b)
    (createRule ?count $?atoms)
    (modify ?sub (value (+ ?count 1)))
    (retract ?new)
)
```

```
(defrule keepConnectionWithSubscription
    "If the new subscription already exists with the connection and the same priority, then no
     adaptation is needed"
    ?new <- (new_subscription (atoms $?pred) (connection ?connId) (priority ?pri))
    ?sub <- (subscription (id ?index) (atoms $?pred) (connections $?subscribers)
        (priorities $?priorities) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq (member$ ?connId $?subscribers) FALSE) (eq (nth$ (member$ ?connId $?subscribers)
        $?priorities) ?pri) (neq (member$ ?connId $?connections) FALSE)))
    =>
    (retract ?new)
)

(defrule addConnectionToSubscription
    "If the new subscription exisits but the connection is not registered to it, then add connection to
     subscription"
    ?new <- (new_subscription (atoms $?pred) (connection ?connId) (priority ?pri))
    ?sub <- (subscription (id ?index) (atoms $?pred) (connections $?subscribers)
        (priorities $?priorities) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (eq (member$ ?connId $?subscribers) FALSE) (neq (member$ ?connId $?connections) FALSE)))
    =>
    (modify ?sub (connections (create$ ?connId $?subscribers)) (priorities (create$ ?pri $?priorities)))
    (retract ?new)
)

(defrule removeConnectionFromWrongSubscription
    "If the connection with its priority is assigned to the wrong subscription, then delete the
     connection from this subscription"
    (declare (salience 5))
    ?new <- (new_subscription (atoms $?new_pred) (connection ?connId) (priority ?pri))
    ?sub <- (subscription (id ?index) (atoms $?sub_pred) (connections $?subscribers)
        (priorities $?priorities) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq $?new_pred $?sub_pred) (neq (member$ ?connId $?subscribers) FALSE)
        (eq (nth$ (member$ ?connId $?subscribers) $?priorities) ?pri) (neq (member$
        ?connId $?connections) FALSE)))
    =>
    (bind ?pos (member$ ?connId $?subscribers))
    (modify ?sub (connections (delete$ $?subscribers ?pos ?pos)) (priorities (delete$
        $?priorities ?pos ?pos)))
)

(defrule removeConnectionNewPriorityIsHigher
    "If the connection is registered to the right subscription but the new priority is higher, then
     remove the connection from the subscription"
    (declare (salience 5))
    ?new <- (new_subscription (atoms $?pred) (connection ?connId) (priority ?pri))
    ?sub <- (subscription (id ?index) (atoms $?pred) (connections $?subscribers)
        (priorities $?priorities) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq (member$ ?connId $?subscribers) FALSE) (> (nth$ (member$ ?connId $?subscribers)
        $?priorities) ?pri) (neq (member$ ?connId $?connections) FALSE)))
    =>
    (bind ?pos (member$ ?connId $?subscribers))
    (modify ?sub (connections (delete$ $?subscribers ?pos ?pos)) (priorities (delete$
        $?priorities ?pos ?pos)))
)

(defrule ignoreConnectionWithNewPriorityIsLower
    "If the connection is registered to the right subscription and the new priority is
     lower, then ignore the new priority"
    (declare (salience 5))
    ?new <- (new_subscription (atoms $?pred) (connection ?connId) (priority ?pri))
    ?sub <- (subscription (id ?index) (atoms $?pred) (connections $?subscribers)
        (priorities $?priorities) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq (member$ ?connId $?subscribers) FALSE) (< (nth$ (member$ ?connId
        $?subscribers) $?priorities) ?pri) (neq (member$ ?connId $?connections) FALSE)))
    =>
    (retract ?new)
)
```

```
(defrule removeConnectionFromWrongSubscriptionWithRightPriority
    "If the connection is assigned to the wrong subscription with the right priority and
     assigned to the right subscription with the wrong priority, then remove the connection
     from the wrong subscription"
    (declare (salience 5))
    ?new <- (new_subscription (atoms $?pred1) (connection ?connId) (priority ?pri))
    ?sub1 <- (subscription (id ?index1) (atoms $?pred1) (connections $?users1) (priorities
        $?priorities1) (form_id ?formId))
    ?sub2 <- (subscription (id ?index2) (atoms $?pred2) (connections $?users2) (priorities
        $?priorities2) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq ?index1 ?index2) (neq (member$ ?connId $?users1) FALSE) (neq (nth$
        (member$ ?connId $?users1) $?priorities1) ?pri) (neq (member$ ?connId $?users2) FALSE)
        (eq (nth$ (member$ ?connId $?users2) $?priorities2) ?pri) (neq (member$ ?connId
        $?connections) FALSE)))
    =>
    (bind ?pos (member$ ?connId $?users2))
    (modify ?sub2 (connections (delete$ $?users2 ?pos ?pos)) (priorities (delete$
        $?priorities2 ?pos ?pos)))
)

(defrule removeSubscriptionFromWrongSubscriptionNotRightlySubscription
    "If the connection if assigned to the wrong subscription with the right priority and not
     assigned to the right subscription, then remove the connection from the wrong subscription"
    (declare (salience 5))
    ?new <- (new_subscription (atoms $?pred1) (connection ?connId) (priority ?pri))
    ?sub1 <- (subscription (id ?index1) (atoms $?pred1) (connections $?users1) (priorities
         $?priorities1) (form_id ?formId))
    ?sub2 <- (subscription (id ?index2) (atoms $?pred2) (connections $?users2) (priorities
        $?priorities2) (form_id ?formId))
    (form_function (id ?formId) (connections $?connections))
    (test (and (neq ?index1 ?index2) (eq (member$ ?connId $?users1) FALSE) (neq (member$ ?connId
        $?users2) FALSE) (eq (nth$ (member$ ?connId $?users2) $?priorities2) ?pri) (neq (member$
        ?connId $?connections) FALSE)))
    =>
    (bind ?pos (member$ ?connId $?users2))
    (modify ?sub2 (users (delete$ $?users2 ?pos ?pos)) (priorities (delete$ $?priorities2 ?pos
        ?pos)))
)

; #### LOCATION AWARENESS ###############################################################################

(defrule setSelfLocation
    "Set the location of the new uform in the knowledge-base"
    (parameter (id ?id) (name this))
    (message (name pos_x) (value ?x))
    (message (name pos_y) (value ?y))
    (message (name pos_z) (value ?z))
    (not (location (id ?id)))
    =>
    (assert (location (id ?id) (x ?x) (y ?y) (z ?z)))
)

(defrule setUserLocation
    "Set the location of a user in the knowledge-base"
    ?xyz <- (user_pos ?u ?x ?y ?z)
    (parameter (id ?id) (name user) (attributes $?attributes) (values $?values))
    (not (location (id ?id)))
    (test (eq ?u (nth$ 1 $?values)))
    =>
    (assert (location (id ?id) (x ?x) (y ?y) (z ?z)))
    (retract ?xyz)
)
```

```
(defrule changeUserLocation
    "Update the location of the user indicated by the parameter"
    ?xyz <- (user_pos ?u ?x ?y ?z)
    (parameter (id ?id) (name user) (attributes ?attribtues) (values $?values))
    ?loc <- (location (id ?id))
    (test (eq ?u (nth$ 1 $?values)))
    =>
    (modify ?loc (x ?x) (y ?y) (z ?z))
    (retract ?xyz)
)

(defrule setUFormLocation
    "Lookup the location of the uform indicated by the parameter and set it in the knowledge-base"
    (parameter (id ?id) (name uform) (attributes $?attributes) (values $?values))
    (uform (type unit) (keys $?keys) (values $?ufvalues))
    (test (testCondition (create$ (length$ $?attributes) $?attributes $?values (length$ $?keys)
        $?keys $?ufvalues)))
    =>
    (bind ?indexX (member$ x $?keys))
    (bind ?indexY (member$ y $?keys))
    (bind ?indexZ (member$ z $?keys))
    (assert (location (id ?id) (x (nth$ ?indexX $?ufvalues)) (y (nth$ ?indexY $?ufvalues))
        (z (nth$ ?indexZ $?ufvalues))))
)

(defrule setDistance
    "Calculate the distance of two locations indicated by the parameters"
    (distance (id ?id) (param1 ?p1) (param2 ?p2))
    (location (id ?p1) (x ?x1) (y ?y1) (z ?z1))
    (location (id ?p2) (x ?x2) (y ?y2) (z ?z2))
    (newUform)
    =>
    (bind ?distance (sqrt (+ (** (- ?x1 ?x2) 2) (** (- ?y1 ?y2) 2) (** (- ?z1 ?z2) 2))))
    (assert (message (name ?id) (value ?distance)))
)

; #### SETTING VALUES FOR NEW SUBSCRIPTION ##########################################################

(defrule setNewPrioritizedSubscriptionForConnection
    "If we know the priorities of all atoms, then we setup the prioritized subscription for the
     connection"
    ?u <- (connection (connId ?connId))
    ?c <- (bandwidth_quality)
    (number_of_priorities (value ?priorities))
    =>
    (bind ?i 0)
    (while (< ?i ?priorities) do
        (bind ?priority (- ?priorities ?i 1))
            (eval (str-cat "(assert (new_subscription (atoms $?atoms_" ?i ") (connection " ?connId ")
                (priority " ?priority ")))"))
            (eval (str-cat "(bind $?atoms_" ?i " (create$ ))"))
            (bind ?i (+ ?i 1))
    )
    (retract ?u)
    (retract ?c)
)

(defrule priorityMatrix
    "If we know the quality of the relevance and of the bandwidth, determine the priority of the atom"
    (declare (salience 5))
    ?rq <- (relevance_quality (id ?r_id) (value ?r_value))
    ?co <- (bandwidth_quality (value ?b_value))
    (number_of_priorities (value ?priorities))
    =>
    (bind ?p (- ?priorities 1))
    (bind ?priority (min ?p (+ ?r_value ?b_value)))
    (while (> ?priority 0)
        (bind ?var_name (str-cat "$?atoms_" ?priority))
        (eval (str-cat "(bind " ?var_name " (create$ " ?r_id " " ?var_name "))"))
        (bind ?priority (- ?priority 1))
    )
    (retract ?rq)
)
```

```
(defrule addConnectionToFormfunction
    "Add the connection to the formfunction"
    (declare (salience 5))
    ?fc <- (formfunction_connection (formfunction_id ?fid) (connection_id ?cid))
    ?ff <- (form_function (id ?fid) (connections $?connections))
    (test (eq (member$ ?cid $?connections) FALSE))
    =>
    (modify ?ff (connections (create$ $?connections ?cid)))
    (retract ?fc)
)

; #### MATCHING RULES ###########################################################################

(defrule newUformReset
    "After the matching is done, delete the newUform fact"
    (declare (salience -10))
    ?nu <- (newUform)
    =>
    (retract ?nu)
)

(defrule atomReset
    "After the matching is done, reset the satisfied field of all atoms to FALSE"
    (declare (salience -10))
    ?fp <- (atom (id ?id) (satisfied TRUE))
    =>
    (modify ?fp (satisfied FALSE))
)

(defrule messageReset
    "After the matching is done, delete all remaining message facts"
    (declare (salience -10))
    ?fm <- (message)
    =>
    (retract ?fm)
)

(defrule matchAtom
    "If the operator of the atom applied to the values of the message and atom is true and the names of
     the message and atom are equal then the atom is matched"
    ?mf <- (message (name ?mes_name) (value ?mes_value))
    ?fs <- (atom (id ?sub_id) (name ?sub_name) (operator ?sub_operator) (value ?sub_value)
        (satisfied ?sub_satisfied))
    (test (and (eq ?mes_name ?sub_name) (eq ?sub_satisfied FALSE) (eval (str-cat "(" ?sub_operator " "
        ?mes_value " " ?sub_value ")"))))
    =>
    (modify ?fs (satisfied TRUE))
    (retract ?mf)
)

; #### QUERIES ##################################################################################

(defrule countUFormFacts
    "Perform the counting op the given parameter"
    (count (id ?id) (param ?param))
    ?nu <- (newUform)
    =>
    (bind ?count (count-query-results countUforms ?param))
    (assert (message (name ?id) (value ?count)))
)

(defquery countUforms
    "Count all uforms that satisfy the conditions of the given parameter"
    (declare (variables ?param))
    (parameter (id ?param) (name uform)  (attributes $?attributes) (values $?values))
    (uform (type unit) (keys $?keys) (values $?ufvalues))
    (test (testCondition (create$ (length$ $?attributes) $?attributes $?values (length$ $?keys) $?keys
        $?ufvalues)))
)
```

```
(defrule countDistanceFacts
    "Perform the counting of uforms within the distance indicated by the distance fact"
    (distance (id ?id))
    (parameter (id ?pid) (name ?id) (attributes value) (values ?value))
    ?nu <- (newUform)
    =>
    (bind ?count (count-query-results countDistances ?id ?value))
    (assert (message (name (eval (str-cat count ?id))) (value ?count)))
)

(defquery countDistances
    "Count all the distance facts with the given id"
    (declare (variables ?id ?value))
    (message (name ?id) (value ?mesValue))
    (test (<= ?mesValue ?value))
)

; #### FUNCTIONS ##############################################################################################

(deffunction createRule (?index $?atom_list)
    "This function dreates a new subscription rule"
    (bind ?rule_name (str-cat "subscription" ?index))
    (bind ?antecedent "")
    (foreach ?p $?atom_list
        (bind ?antecedent (str-cat "(atom (id " ?p ") (satisfied ?" ?p ")) " ?antecedent))
    )
    (bind ?antecedent (str-cat ?antecedent " (subscription (atoms " $?atom_list ")
        (connections $?user_list) (priorities $?priorities))"))
    (bind ?antecedent (str-cat ?antecedent "(test " (getSubscriptionString $?atom_list)
        ")"))
    (bind ?rule_body (str-cat ?antecedent " => (assert (result (connections $?user_list)
        (priorities $?priorities)))"))
    (eval (str-cat "(defrule " ?rule_name ?rule_body ")"))
)

(deffunction hasPredicates ($?predicate_list)
    "Tests if a subscription contains atoms"
    (return (> (length$ (complement$ (create$ and or not nil) $?predicate_list)) 0))
)

(deffunction equal_to_first (?item $?list)
    "Tests if the first item in the list is equal to the given item"
    (if (> (length$ $?list) 0) then
        (return (eq ?item (nth$ 1 $?list)))
    else
        (return FALSE)
    )
)

(deffunction testCondition ($?parameters)
    "Test the conditions stated in the parameter, the result is a boolean value indicating if the
     conditions are met"
    (bind ?lengthParameters (nth$ 1 $?parameters))
    (bind $?attributes (subseq$ $?parameters 2 (+ ?lengthParameters 1)))
    (bind $?values (subseq$ $?parameters (+ ?lengthParameters  2) (+ (* ?lengthParameters 2) 1)))
    (bind ?lengthUform (nth$ (+ (* ?lengthParameters 2) 2) $?parameters))
    (bind ?indexUform (+ (* ?lengthParameters 2) 2))
    (bind $?keys (subseq$ $?parameters (+ ?indexUform 1) (+ ?indexUform ?lengthUform)))
    (bind $?ufvalues (subseq$ $?parameters (+ ?indexUform ?lengthUform 1) (+ ?indexUform (* ?lengthUform
        2))))
    (bind ?result TRUE)
    (bind ?j 1)
    (foreach ?attribute $?attributes
        (bind ?subresult (and (neq (bind ?i (member$ ?attribute $?keys)) FALSE) (eq (nth$ ?i $?ufvalues)
            (nth$ ?j $?values))))
        (bind ?result (and ?result ?subresult))
        (bind ?j (+ ?j 1))
    )
    (return ?result)
)
```

```
(deffunction getSubscriptionString ($?predicate_list)
    (bind $?include_list (create$ and or not $?predicate_list))
    (bind $?current_nodes (create$ ))

    (foreach ?n $?nodes
        (if (neq FALSE (member$ ?n $?include_list)) then
            (bind $?current_nodes (create$ $?current_nodes ?n))
        else
            (bind $?current_nodes (create$ $?current_nodes nil))
        )
    )

    (bind ?str "")
    (bind $?countPredicates (create$ 0))
    (bind $?logicList (create$ ))
    (bind $?visited_nodes (create$ ))
    (bind $?stack (create$ (first$ $?branches)))

    (if (eq (hasPredicates $?current_nodes) TRUE) then
        (while (> (length$ $?stack) 0)
            (bind ?position (member$ (nth$ 1 $?stack) $?branches))
            (bind ?n (nth$ (nth$ 1 $?stack) $?current_nodes))
            (if (eq (member$ (nth$ 1 $?stack) $?visited_nodes) FALSE) then
               (if (or (eq ?n and) (eq ?n or) (eq ?n not)) then
                       (bind ?str (str-cat ?str "(" ?n " "))
                       (bind $?countPredicates (create$ (+ (nth$ 1 $?countPredicates) 1)
                              (rest$ $?countPredicates)))
                       (bind $?countPredicates (create$ 0 $?countPredicates))
                       (bind $?logicList (create$ ?n $?logicList))
               else
                       (if (neq ?n nil) then
                              (bind ?str (str-cat ?str "?" ?n " "))
                              (bind $?countPredicates (create$ (+ (nth$ 1 $?countPredicates) 1)
                                     (rest$ $?countPredicates)))
                       )
               )
               (bind $?visited_nodes (create$ $?visited_nodes (nth$ 1 $?stack)))
            else
               (if (eq ?position FALSE) then
                       (if (eq (nth$ 1 $?countPredicates) 0) then
                              (if (>= (length$ $?logicList) 2) then
                                     (if (eq (nth$ 2 $?logicList) and) then (bind ?str (str-cat ?str
                                            "TRUE")))
                                     (if (eq (nth$ 2 $?logicList) or) then (bind ?str (str-cat ?str
                                            "FALSE")))
                                     (if (eq (nth$ 2 $?logicList) not) then
                                            (if (eq (nth$ 3 $?logicList) and) then (bind ?str (str-cat
                                                   ?str "TRUE")))
                                            (if (eq (nth$ 3 $?logicList) or) then (bind ?str (str-cat
                                                   ?str "FALSE")))
                                     )
                              else
                                     (bind ?str "(eq TRUE FALSE)")
                              )
                       )
                       (bind ?str (str-cat ?str ") "))
                       (bind $?countPredicates (rest$ $?countPredicates))
                       (bind $?logicList (rest$ $?logicList))
               )
            )
            (if (neq ?position FALSE) then
               (bind $?stack (create$ (nth$ (+ ?position 1) $?branches) $?stack))
               (bind $?branches (delete$ $?branches ?position (+ ?position 1)))
            else
               (bind $?stack (rest$ $?stack))
            )
        )
    else
        (bind ?str "(eq TRUE FALSE)")
    )

    (return ?str)
)
```

APPENDIX C: DATADISTRIBUTIONAGENT CLASS

```
/**
 * title        :   Date Distribution Agent
 * description  :   The interface of the data distribution agent
 * @author      :   B.P.I. van der Poel
 * @since       :   07/14/2002
 * @version     :   %Version%
 */

package disciple.agents;

import disciple.repository.UForm;
import disciple.repository.Repository;
import disciple.bandwidth.Bandwidth;

public abstract class DataDistributionAgent {

    /**
     * The repository of the server
     */
    protected Repository repository_;

    /**
     * This constructor initiates a new data distribution agent
     * @param repository The repository of the server
     */
    public DataDistributionAgent(Repository repository) {
        repository_ = repository;
    }

    /**
     * This method sets the repository of the agent
     * @param repository repository of agent, narmally points to global repository of server
     */
    public void setRepository(Repository repository) {
        repository_ = repository;
    }

    /**
     * This method returns the repository of the agent.
     * @return repository of the agent, normally points to global repository of server
     */
    public Repository getRepository() {
        return repository_;
    }

    /**
     * This method will be overridden in most cases. An agent is instantiated.
     * @param repository The repository of the server
     * @return The instantiated agent
     */
    public static DataDistributionAgent loadAgent(Repository repository) {
        DataDistributionAgent agent = null;
        return agent;
    }

    /**
     * This method sets up the rules for a connection.
     * @param subscription The text that define the rules
     * @param connId The id of the connection
     */
    public abstract void setRules(String subscription, String connId);
```

```
    /**
     * Test to which connections a uform has to be forwarded.
     * @param uf The uform that has to be matched with the rules
     * @return Mapping of connections to priorities
     */
    public abstract PrioritizedConnectionList applyRules(UForm uf);

    /**
     * Sets the bandwidth of a connection for the data distribution agent.
     * @param id The id of the connection
     * @param bw The bandwidth of the connection
     */
    public abstract void setQoSParameters(String id, Bandwidth bw);

    /**
     * Determines the list of uforms needed to refresh the repository of a user.
     * @id The id of the connection requesting the update
     * @return The list of uforms
     */
    public abstract UForm[] refresh(long id);

}
```