University of Twente

Faculty of Electrical Engineering, Mathematics, and
Computer Science

Master Thesis

Context Discovery Adapter (CDA) Protocol

Author:
Hanga Boros
Date:
August 14, 2008

Graduation Committee:
Dr. Ir. Geert Heijenk
Fei Liu
Pravin Pawar

Abstract

Context Discovery Adapter or CDA as we call it is designed and implemented as a proof of concept to demonstrate ability to discover and exchange context-specific information across disparate networks. The implementation adopts Mobile Service Platform (MSP) and Ahoy, two context management frameworks with greatly different properties. MSP is a JINI-based middleware capable of supporting connectivity across cellular and fixed networks. Ahoy on the other hand is a lightweight context discovery framework designed for mobile ad-hoc networks and built with the help of Bloom filters. The project demonstrates that heavyweight context interchange frameworks like MSP and lightweight frameworks similar to Ahoy with limited context sensitivity can establish mutual and transparent context exchange. CDA proves not only a possibility for more pervasive context-aware applications, but also provides a platform for further research.

Contents

1 Introduction **1**

1.1 Goal of the Thesis 1

1.2 Research Questions 2

1.3 Requirements 3

1.4 Research Approach 4

1.5 Relevance and Applications 5

1.6 Document Structure 6

2 Related Literature **7**

2.1 Ahoy 7

2.1.1 Attenuated Bloom Filter 7

2.1.2 Framework Components 8

2.1.3 Protocol Specification 9

2.2 Mobile Service Platform 11

2.2.1 Nomadic Mobile Service 12

2.2.2 Framework Components 12

2.2.3 Protocol Specification 14

2.3 Related Work 16

2.3.1 Context Distribution Framework 16

2.3.2 Web Services 16

2.3.3 CORBA 17

2.3.4 JXTA and Peer-to-peer Frameworks 17

2.3.5 EgoSpaces and Mobi-Blog 18

3 Design and Architecture **19**

3.1 Guidelines 19

3.1.1 Bridge-like Design Paradigm 19

3.1.2 Prototype CDA Framework 19

3.2 CDA Architecture 21

3.2.1 Announcing Ahoy Context Sources to MSP 21

3.2.2 Announcing MSP Context Sources to Ahoy 25

3.2.3 Discovery of Ahoy Context Sources by MSP Clients . 29

3.2.4 Discovery of MSP Context Sources by Ahoy Clients . 30

3.2.5 Context Distribution Between Ahoy and MSP 33

3.3 CDA Architecture: Final Conclusions 38

4 Protocol Specification **39**

4.1 Functional Description 39

4.1.1 CDA Device Service 39

4.1.2 CDA Surrogate 42

4.1.3 CDA Service Proxy 43

4.1.4 Broker Service 44

4.1.5 CDA Daemon 46

4.2 Message Types 47

4.2.1 Channel 1: CDA Device Service and CDA Daemon . . 47

4.2.2 Channel 2: HTTP Interconnect Protocol 48

4.2.3 Channel 3: Java RMI in MSP 50

4.2.4 Channel 4: Client-Server Communication 50

5 Implementation Report **51**

5.1 Implementation Steps 51

5.2 Implementation Tools 54

5.3 Test Report 55

5.3.1 Test Scenarios 55

5.3.2 Test Measurements 55

5.3.3 Test Results 57

6 Conclusions **61**

6.1 Contribution 63

6.2 Future Work 63

7 Glossary **65**

1 Introduction

Today we witness increasing demand for adaptive applications that deliver context-sensitive information. For instance, GPS/GPRS based applications like the novel Zipcar project[1] already incorporate location-aware context information across ad-hoc mobile and fixed networks like the Internet. Handheld devices like PDAs and Smartphones are already network-enabled paving way for personalised context-aware applications.

A main challenge these adaptive applications face is the free movement and intermittent connectivity of users and context sources. Ideally, the applications should be able to cross different network transports seamlessly, in a manner transparent to the user. Bridging across network environments of different kinds, however, is not a simple task. The specifics of wired networks (e.g. Internet) are fundamentally different from mobile ad-hoc communication technologies (e.g. Bluetooth) and of wireless (e.g. WIFI) and cellular (e.g. GPRS) transports. There are numerous challenges that context-aware computing has to overcome before an all-pervasive, truly ubiquitous application environment becomes reality. This thesis project acts as the stepping stone toward achieving that.

1.1 Goal of the Thesis

As suggested in the introduction, an important motivation in context-aware computing is to develop increasingly pervasive and adaptive application frameworks. That research goal has been the motivation for the Freeband AWARENESS project[2] and the current thesis work.

The goal of the AWARENESS project has been to research different paradigms for context management frameworks that would be capable of realizing pervasive context discovery and exchange across disparate network environments. A number of frameworks have emerged from the study, two of which have served as inspiration for this Master's thesis: The Mobile Service Platform[3] (MSP) and Ahoy[4]. The two frameworks at hand rely on greatly different communication and context exchange mechanisms. They have been designed to work in different network environments: One is specific to mobile ad-hoc networks, while the other one is designed for fixed/cellular networking environment. Overcoming their differences and interlinking them to enable mutual and seamless context information transfer raises a number of challenges that this thesis work wishes to answer.

MSP is a relatively heavyweight JINI-based service oriented middleware capable of supporting connectivity across cellular and fixed networks. Ahoy

on the other hand is a lightweight context discovery framework designed for mobile ad-hoc networks and built with the help of Bloom filters. The two frameworks have different degrees of context awareness: While MSP supports both the discovery and distribution of context information, the current implementation of Ahoy supports only context discovery and offers no mechanism for context distribution. This property and other aspects of Ahoy make it less suitable for context-aware computing than its counterpart MSP.

This thesis proposes to interlink Ahoy and MSP in such manner that the new framework preserves the characteristics of the underlying networks and application environments. In other words, the new framework is required to keep the lightweight character of Ahoy as much as possible unaltered, yet also interface it with the service-oriented architecture of MSP. We envision the new framework as a bridge between Ahoy and MSP that leaves the two end points as much as possible intact. If the thesis project succeeds in bridging Ahoy and MSP in spite of their fundamental differences, new perspectives and motivation can be brought to context-aware computing. We would get one small step closer to the vision of a fully transparent environment of context-sensitive applications.

The bridge framework or protocol set that this thesis aims to design is called Context Discovery Adapter (CDA).

1.2 Research Questions

Having described the goal of this thesis, in this section we formulate the research questions:

- What should the CDA protocol architecture be like? What design paradigm should be used to keep the modifications to Ahoy and MSP minimal?
- What representation format should be used for context sources in the CDA protocol so that both Ahoy and MSP can read it?
- What modifications are needed to the default Ahoy and MSP context announcement and discovery protocols to enable registration and discovery of CDA context sources across Ahoy and MSP?
- What modifications are needed to the default Ahoy and MSP context exchange (i.e. distribution) protocols to ensure mutual exchange of context information between CDA clients and services across the two frameworks?

- Is it possible to implement a protocol prototype which has feasible real-life performance and resource utilization? What does the prototype tell us about the scalability, modularity and extensibility of the CDA protocol? Does the implementation ensure truly transparent context discovery and exchange?

1.3 Requirements

This section summarizes the list of requirements the CDA protocol is desired to fulfill while attempting to answer the formulated research questions.

- **Context discovery** and **exchange** between Ahoy and MSP should be **mutual**.
- The **communication** between the two frameworks should be **seamless**. In other words, the differences between the two original frameworks should stay hidden from the end user and the context exchange should not be in any way disrupted when crossing framework boundaries.
- The CDA protocol should impose minimal changes and overhead onto the protocols of the two original frameworks. We refer to this property as the **minimal impact** requirement.
- We expect the CDA protocol architecture to be **modular**, **extensible** and **scalable**. Modularity ensures that modifications can be easily integrated and that different functional modules stay grouped together. Extensibility is meant to allow future expansion of the CDA protocol with new frameworks other than Ahoy and MSP. By scalability we mean that the observed properties of CDA should stay unchanged when the components of Ahoy and MSP increase in number, whether statically or during runtime.
- For a successful proof of concept it is expected that the CDA protocol has a feasible **implementation prototype**. The implementation should show realistic **performance, robustness and resource utilization** features. The platform-, hardware- and software-related requirements of the implementation should be also feasible in the technological context of our times.

These requirements play an important role in the verification of the CDA protocol design and test results. They are further elaborated on in the *Conclusions* (Section 6).

1.4 Research Approach

The thesis work has been carried out in a series of steps, as listed below:

- **Study of related literature.** This phase comprises literature research on Ahoy, Jini, MSP and other related topics.
- **Analysis of design alternatives.** In this phase we decide about the architecture and design guidelines and consider a number of alternative solutions for the CDA protocol architecture within the chosen design paradigm.
- **Selecting the final CDA protocol architecture**. The final architecture is selected from the list of alternatives obtained in the previous step. This phase concludes with a detailed specification of the emerged CDA protocol, which includes a list of its components, communication channels and message formats.
- **Implementation of the CDA prototype.** In this phase a prototype of the CDA protocol is implemented. The chosen protocol architecture serves as blueprint. If the blueprint cannot be implemented as is, the protocol architecture is modified accordingly.
- **Test scenarios and test measurements.** In this phase the test scenarios are designed and executed. The obtained test measurements get processed.
- **Evaluation and discussion**. This phase consists of a discussion and evaluation of the properties of the design, the implementation prototype and the obtained test results. The evaluation is performed against the initially formulated research requirements. Requirements that are quantifiable, e.g. resource utilization and performance values, get evaluated based on test results. Requirements criteria that are not quantifiable, like robustness, extensibility, modularity and alike, get evaluated through qualitative discussion. This phase closes with conclusions about the CDA protocol, its contribution, benefits versus drawbacks and suggestion for future work.
- **Thesis report**. The last phase is writing the thesis report, which is the document at hand. The report includes the results of each research phase listed above.

1.5 Relevance and Applications

To grasp better the contribution of the CDA protocol, in this section we consider two plausible real-life applications that would make use of it.

The first scenario is volcano eruption monitoring. Bluetooth or WI-FI devices establish an ad-hoc network and are spread in the endangered area to collect and process data. Some of them offer their own measurements as Ahoy service. Other devices post-process measurements received from others and offer the computed results as Ahoy service. The type of measurements can be temperature, air pressure, shock values and others. A monitoring center can request the different measurements and perform advanced data processing that eventually leads to an escape strategy. Between the endangered area and the monitoring center the measured data traverses a variety of communication channels, among which there could be infrared, WIFI, GPRS, Ethernet or others. The data sent by the devices would have to travel uninterrupted through the various network transports. The MSP framework would be able to cover areas of the fixed and GPRS networks, yet it would not cover the areas of the ad-hoc network. This problem could be surmounted by the CDA protocol. Figure 1 visualizes this scenario.

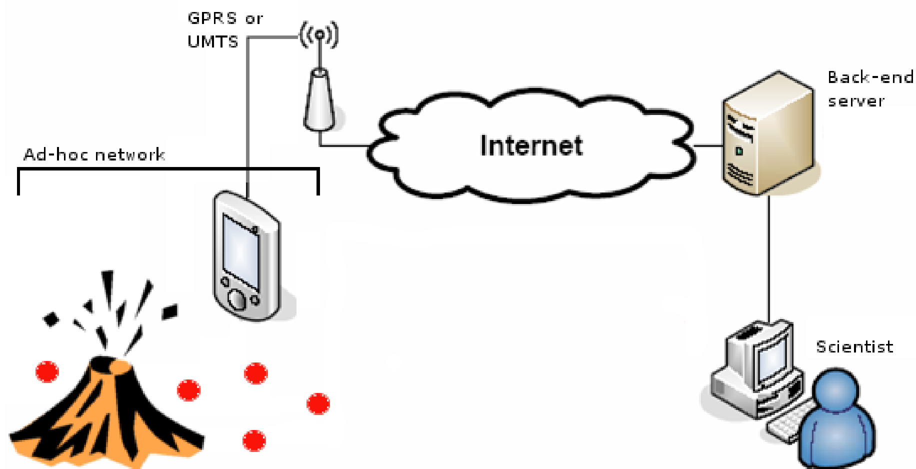

Figure 1: *Vulcano eruption monitoring system.*

In a second scenario the context source could be located at the other end of the network spectrum in the fixed network. In this scenario an ad-hoc network is used to monitor a flood-endangered area. When the sensor ad-hoc nodes detect alarming water levels, they can request strategic advice from a remote monitoring center that post-processes the received measurements. The ad-hoc nodes could be given instructions, for instance, to open up safety

claps and guide the water flow. In this scenario the service we are interested in originates in the fixed network, in an MSP-like environment, and is used to govern ad-hoc clients.

These two calamity monitoring applications have a highly adaptive nature and can perform decision making without human attendance. There are, on the other hand, also numerous human-oriented applications of CDA. Examples would be remote patient or criminal monitoring systems, or conference applications that provide context-specific information to the participants.

1.6 Document Structure

In the current chapter (Chapter 1) the reader got introduced to the research context, goal of the thesis, research questions and adopted research methodology. The rest of the paper is organized as follows:

Chapter 2 presents related literature introducing the Ahoy and MSP frameworks, and it presents other works with similar research interest as the current thesis: Context management frameworks developed by AWARENESS project, some federation projects and state of the art context-aware mobile frameworks.

Chapter 3 describes the design guidelines, presents a list of feasible bridging points between Ahoy and MSPand it lists alternative design considerations for the CDA protocol. The chapter concludes with suggestions for the final CDA protocol architecture model.

Chapter 4 gives an account of the specifications of the CDA protocol architecture. It gives an overview of all the CDA protocol components and describes the communication channels, message types and event flows.

Chapter 5 gives a report of the implementation process involving the CDA prototype. It also presents the design of test scenarios and the obtained test results.

Chapter 6 presents the evaluation of the CDA protocol prototype against the research requirements and it discusses the benefits/drawbacks of the protocol. The chapter closes with conclusions regarding the achieved thesis goals and suggestions for future work.

Finally, there are a number of add-on sections: A glossary which contains a list of abbreviations and bibliographical references.

2 Related Literature

In this section we introduce Ahoy and MSP and discuss several frameworks that offer alternative solutions to the cross-platform context interchange issues this thesis studies.

2.1 Ahoy

Ahoy[4], developed as part of the AWARENESS project, is a lightweight protocol designed to discover services in resource-poor, wireless multi-hop ad-hoc networks. Compared with MSP and other AWARENESS project frameworks Ahoy has reduced context-awareness. Its current implementation has no support for context exchange (i.e. distribution), it supports only the registration and discovery of context sources. Extending Ahoy with an appropriate exchange protocol is certainly possible and this thesis work considers alternatives for that.

2.1.1 Attenuated Bloom Filter

Attenuated Bloom Filter (ABF) is a unique adaptation by Ahoy to represent the availability of context sources. ABF is an array of Bloom filters[5][6]. A Bloom filter provides hash-based, compressed representation for the identifier (i.e. unique name) of context sources. A Bloom filter initially has all its bits set to 0. After hashing the string identifier of the context source, some of the bits get set to 1 in a pattern unique to that identifier. Every context source is associated with a unique hash or filter. Multiple context sources can be mapped to the same filter by combining the bits of their individual filters through a logical "OR" operation. When an Ahoy component computes the ABF, it is done the following way: The context sources that are at the same hop distance from the component at hand get mapped to the same filter through the logical OR operation. Context sources that are at a different hop distance get mapped to a different filter, through the same mechanism. If we stack these Bloom filters together based on the increasing hop distance, we obtain an ABF array. Thanks to the characteristics of the ABF, Ahoy is able to store proximity and routing information about its context sources in a highly efficient way. This property contributes greatly to Ahoy's lightweight nature.

Figure 2 visualizes the computation of an ABF.

A known side-effect of Bloom filters is the occurrence of false positives. False positives happen when the filters of multiple context sources get merged

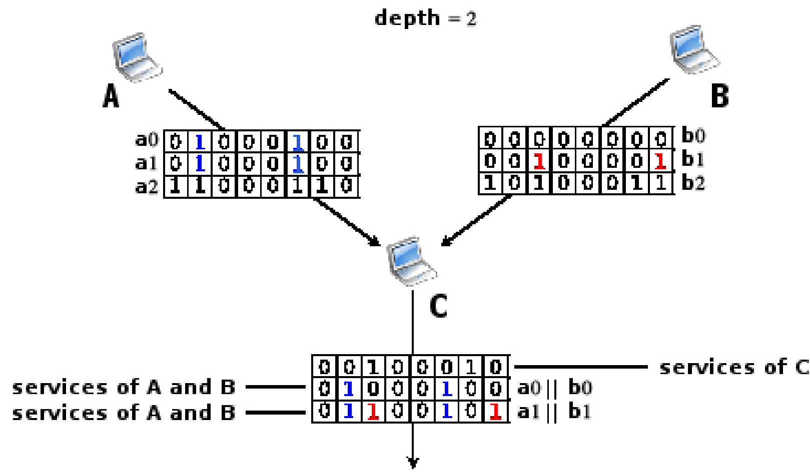

Figure 2: *Computation of a simple ABF. Nodes A and B both have an ABF, which contains service hashes up to two hop distances. For node C the new ABF is calculated. The top layer of the filter contains the hashes of services that the node itself offers. Node C puts in the second layer the services that are offered by its one-hop neighbors (A and B), joining them with a logical OR operation. The bottom layer in node C gets the services that are one-hop away from node A or B, hence two hops away from C. The blue and red color coding helps understand which service hashes comes from which node. The number of stacked filters in one ABF is* depth *+1, where* depth *is a configurable Ahoy parameter. Original image borrowed from[4].*

through the logical "OR" operation to a new filter and accidentally create bit patterns that might overlap with the hash of a service identifier that is not known within the network. If a client queries for that particular context source, it is made to believe that the service exists, while in reality it does not. There is no known remedy to false positives, except trying to minimize their occurrence.

2.1.2 Framework Components

Ahoy has been developed and tested on Unix-like (GNU/Linux) platforms. Its stand-alone modules are written in Ruby programming language[7]. The two distinct architectural components of Ahoy are its clients and the Ahoy daemon.

Ahoy daemon: The core component of Ahoy is the daemon. Each Ahoy node requires one daemon to run to be able to participate in Ahoy discovery. The Ahoy node and daemon are therefore synonyms. The daemon supports two different communication types: It communicates with other Ahoy nodes through UDP sockets and with local clients running on the same node through Unix-specific AF_LOCAL sockets. Due to the usage of the AF_LOCAL sockets, the current implementation of Ahoy runs only on Unix-like systems.

Ahoy client: One or more Ahoy clients can run on an Ahoy node. Their presence is not essential, like that of the daemon. The Ahoy client (also called user) can announce and revoke a service of its own and can initiate query for Ahoy services. Clients never communicate with other Ahoy nodes directly, they do all communication through the local Ahoy daemon.

Figure 3 shows an Ahoy node with its two main components and their functionality.

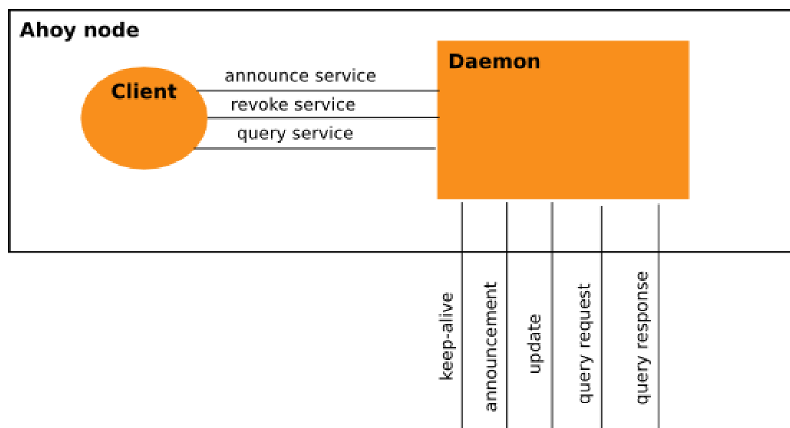

Figure 3: *Main components of the Ahoy protocol. In this figure only one client component is shown. In reality there can be multiple clients running on one node, but never more than one daemon.*

2.1.3 Protocol Specification

Throughout this report we split context management protocols into three distinct functional units. The first unit is *context announcement*. This is the protocol segment which registers context sources. At the end of this step the context sources are published and available to clients. The second functional unit is *context discovery*. This segment enables users (i.e. clients) to query

for context sources and receive their contact information. The third and final functional unit is *context distribution* or *context exchange*. By this we mean the dialog in which a context source transfers its context information to the client(s). For context exchange to happen the client first has to successfully discover the context source.

Occasionally other terms might be used to refer to these three units. By context discovery we frequently mean also context announcement and context exchange is occasionally used to refer to all three functional units together in a more generic sense.

As stated earlier, Ahoy does not support context distribution. The two other units that it supports are described below.

Context Announcement: In Ahoy every node keeps track of its own services announced by the local clients, and of the services offered by the neighbors that are within configurable hop distance defined by the *depth* parameter. The information on all these context sources is encoded in the ABF. Upon any change that effects the list of context sources the Ahoy node re-calculates a new ABF and sends it as announcement to its neighbors one hop away. Those neighbors, subsequently, send their own re-calculated ABF to all the nodes that are one hop away from them. This leads to an incremental announcement distribution, one hop at a time, as shown in Figure 4 in a grid-like network.

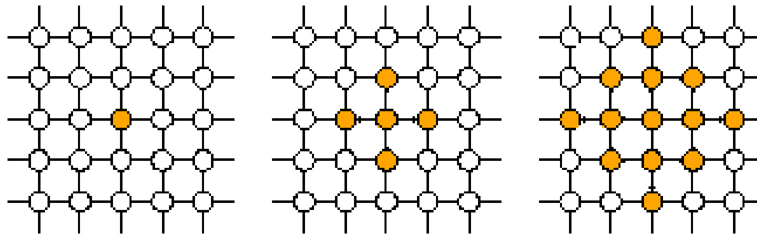

Figure 4: *The distribution pattern of Ahoy announcements in a grid-like network, width* depth*=2. In this scenario the node in the center is sending an announcement. After the first step, the announcement reaches the four direct neighbors. Each of those four nodes calculate their own announcement and send it to their own direct neighbors. After that step all nodes that have received the announcement are colored orange. The non-orange nodes are further than two hops from the original node, they do not get to hear the announcement. Image borrowed from[4].*

Context Discovery: When an Ahoy node wishes to find a context source, it checks first its local service list. If the service is not offered locally, the daemon checks the last ABFs it received from its neighbors to find a match in one of the filters for the context source it is interested in. If matches are found, which might be a false positive, the node sends a query to the neighbors whose ABF showed a match. The neighbors either offer the service and send a query reply or forward the query to their neighbors, the ones whose ABF revealed a match. If the querying node receives a query reply, it discards all other replies arriving after that. The node that offers the service replies to the querying node by sending a query reply through direct unicast. Figures 5 and 6 illustrate the routing of an Ahoy query and its response.

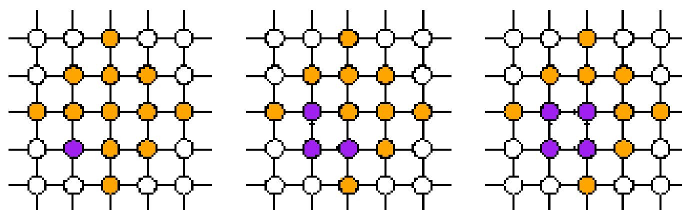

Figure 5: *The routing of Ahoy context query to the node that offers the context service. The nodes marked by purple get to receive the query message. The node offering the service is the one in the middle, the node querying is the purple node on the left-most image. Image borrowed from[4].*

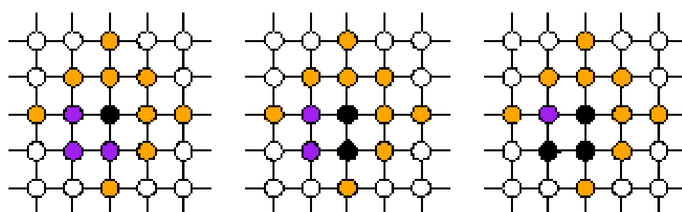

Figure 6: *The routing of Ahoy context query response. The response travels along the nodes colored in black. It is a unicast message targeted to the node where the query initiated from. The route is chosen by the underlying routing protocol and is random when more alternatives exist. Image borrowed from[4].*

2.2 Mobile Service Platform

Mobile Service Platform[3][8] (MSP) is the other context management framework from the AWARENESS project that we study. MSP is a relatively

heavyweight framework designed on top of JINI middleware. JINI[9] is a service oriented, Java-based technology designed to provide a programming framework for large distributed systems consisting of federations of network services and clients. JINI, as well as MSP, are systems of loosely coupled services. JINI uses Java Remote Method Invocation (RMI) as communication layer between its services and clients.

MSP, essentially, is an extension of JINI. It is implemented according to the JINI Surrogate Architecture Specification[10][11] (JSAS), which is an extension of the JINI standards designed to interlink JINI-type fixed networks with mobile environments. Thanks to JSAS and MSP, resource-limited mobile devices can participate in JINI-based applications hosted in the fixed network.

2.2.1 Nomadic Mobile Service

Nomadic Mobile Service (NMS) is how MSP terms the service running on a mobile device that can connect to the JINI network only through a third-party connector module specified by the JSAS standard. NMS is also synonym for the JSAS-based model that MSP uses to interlink cellular/mobile and fixed networks. The type of embedded mobile devices (i.e. PDA, cell-phone) that MSP models as NMS usually communicate with the Internet through wireless networks like Infrared, Bluetooth, WI-FI, GSM or UMTS. Services running on such devices are nomadic because they roam relatively freely from one mobile communication service to another.

We look at the NMS properties more in detail in the overview of the MSP framework components.

2.2.2 Framework Components

Figure 7 shows the main components of MSP.

Surrogate Host: The Surrogate Host is the helper component defined by JSAS as the third-party connector enabling mobile services to connect to JINI networks. MSP reuses Sun's implementation of Surrogate Host known as Madison.

MSP Service: We use the term MSP service to denote any JINI-based service application that participates in an MSP-enabled framework. These

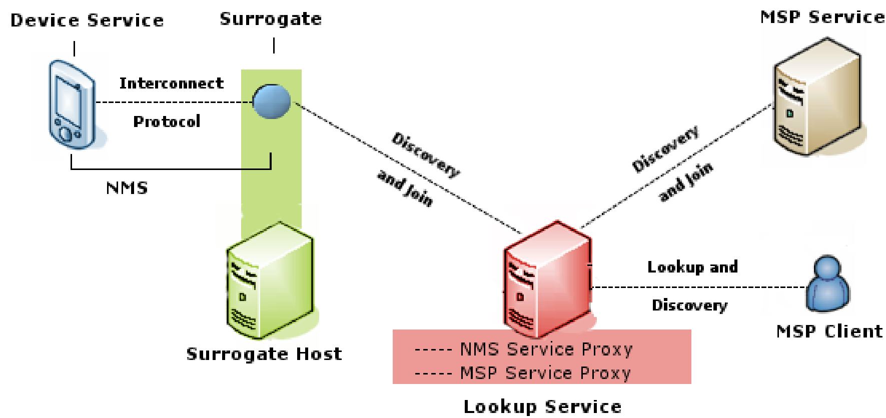

Figure 7: *The main components of MSP: NMS (consisting of Device Service and Surrogate), Surrogate Host, JINI Lookup Service and client. A regular MSP (JINI) service is also included for comparison with the NMS. The difference between the NMS and a regular MSP service is that the NMS consists of two subcomponents and that its registration mechanism contains a few extra steps. The NMS first has to register its Surrogate with the Surrogate Host. Once that done, the NMS Surrogate acts like a regular JINI service and performs the same registration mechanism as JINI services do. The NMS Surrogate is hosted in the fixed network, while the Device Service is hosted on the mobile device. These two communicate with each other through the JSAS Interconnect Protocol.*

are usually regular JINI services except for the NMS, which is a special kind of MSP service.

NMS: Nomadic Mobile Service, or NMS, is the JSAS-based model MSP uses to interlink JINI-based frameworks with mobile/cellular environments. NMS consists of two subcomponents: Device Service (DS) and Surrogate. The DS runs on the mobile device, while the Surrogate is a virtual representation of the DS and runs in the fixed JINI network. The registration of an NMS differs from standard JINI service registration in a few steps. The NMS has to find a Surrogate Host first and register its Surrogate with it. Once that done, the NMS Surrogate behaves like a regular JINI service.

Device Service: The Device Service (DS) is the software component of NMS which runs on the mobile device. Sometimes the Device Service and NMS are used as synonyms.

Service Surrogate: The Service Surrogate is the software component of the NMS which runs in the fixed JINI framework and offers a JINI-compatible representation for the NMS. It acts like a helper for the mobile Device Service, e.g. it offloads the DS by caching and computing data. The NMS Surrogate, once activated and registered with the Surrogate Host, is just like any regular JINI service.

HTTP Interconnect Protocol: The HTTP Interconnect Protocol is an MSP component implemented conform the JSAS Interconnect protocol specification. The NMS Surrogate and Device Service use this protocol to communicate and stay synchronized.

Service Proxy: When MSP services register with the JINI network they are required to submit a proxy object. This object we call the Service Proxy. The proxy's Java interface serves as the unique identifier of the service in the JINI network. Clients establish connection with a service through its Service Proxy: They obtain a handle on the proxy during service discovery and call remote methods on it, which the proxy relays to the service.

MSP Client: We use this term to denote any JINI-based client application that participates in an MSP-enabled framework. An MSP client queries the same way for regular MSP services and for NMS. An MSP client that queries for NMS-type services we sometimes call an NMS client.

Lookup Service (LS): The LS is a standard JINI component, also used by MSP. The LS is essential for service registration and discovery. It is like a name directory that stores and provides information about registered services.

2.2.3 Protocol Specification

In this section we look at how MSP implements the three context manipulations: *context announcement*, *context discovery* and *context distribution*.

Context Announcement: Regular JINI services register themselves through the following procedure: The service locates available lookup services and sends to them its Service Proxy and optionally, some service attributes. Each LS creates a unique entry for the service.

The registration of NMS-like services has a few extra steps in the beginning: The NMS finds a Surrogate Host and registers its Service Surrogate with it. When doing so, the Surrogate Host downloads the Java byte code of the Surrogate from a public URL and activates the Surrogate. Once the Surrogate is registered, it behaves like a regular JINI service and follows the registration procedure described above, namely it registers its Service Proxy and service attributes with the found lookup services.

Context Discovery: The discovery of context sources in MSP follows the standard JINI discovery procedure: Clients find available lookup services and to each found LS they send a number of service parameters that are meant to identify the service of their query. The service parameters can be of different types: They can be the unique Java type of a service, in which case that specific service is expected in return; They can be the Java type of a class of services, in which case multiple matches are found; They can be a service name or any other service attribute, and they can also be the combination of all of the above. The service attributes that services get filtered on can also be user-defined. JINI discovery returns the matches to the client, which selects the service it wishes to set up connection with. For the selected service the client downloads the Service Proxy and initiates communication. Though this procedure describes the standard JINI discovery, we note that MSP uses the very same discovery mechanism. Clients do not notice any difference between querying for a JINI service and for an NMS whose Surrogate acts like a JINI service.

Context Distribution: MSP, contrary to Ahoy, has support for context distribution. The distribution mechanism is fully JINI-based and uses Java RMI between clients and services. When a client finds a service, it downloads its Service Proxy and starts calling remote methods on the proxy, which the proxy relays to the actual service.

When the service is NMS-like, context distribution is only slightly different. For some methods that the client calls on the Service Proxy of the NMS Surrogate, the Surrogate would have to contact the Device Service. In those instances the Surrogate and Device Service exchange user-defined HTTP messages across the Interconnect protocol until the Surrogate is able to send a reply to the client. To a number of other methods the Surrogate might be able to send a reply itself, offloading the Device Service.

2.3 Related Work

To put the current thesis in research context we discuss a few frameworks that deal with cross-platform context interchange issues of the kind that CDA addresses.

2.3.1 Context Distribution Framework

Context Distribution Framework[12] (CDF) has been developed as a part of the AWARENESS project. CDF is essentially an advanced extension of MSP. It fully integrates MSP architecture and reuses the NMS model for interlinking cellular and fixed networks.

CDF surmounts MSP by enhancing the basic MSP framework with context-awareness and advanced context manipulations. An example is the CDF's context representation system which enables both mobile services and mobile clients to participate in context exchange. CDF is able to differentiates between resource-rich and resource-constrained clients. To resource-poor mobile clients the CDA provides an offload mechanism. Offloading is accomplished by CDF's Context Distribution Service (CDS). In case of resource-poor clients the CDS performs the required context manipulations and returns only the final match to the client. In case of resource-rich clients the CDS performs partial or no context computations at all, allowing the client to take a more active part in the process.

In addition to the context representation system the CDF also offers a classification system for context sources. For that it uses Quality of Context (QoC) principles.

CDF's contribution with regard to interlinking heterogeneous network environments is not much different from that of MSP. Both CDF and MSP use the NMS model to bridge cellular and fixed networks, yet their cross-network coverage stops there.

2.3.2 Web Services

Web services[13][14] are a set of emerging standards that enable interlinking between heterogeneous IT processes and systems. In other words, they offer a framework to link together applications regardless to the underlying network technology. Similarly to JINI, web services are designed according to the SOA paradigm. The similarities with JINI do not end here: Both web services and JINI depend on a number of central networking components like HTTP servers, where remote objects or service definitions can

be downloaded from. Web services seem, however, more flexible and easier to extend to diverse network environments. The reason for that is their platform-independent XML-based messaging mechanism. JINI systems do not have an equally flexible messaging solution, they use Java RMI for client-server communication.

Web services are not being used extensively for mobile networks. The reason for that relies in their high dependence on centralized components and configuration parameters, similar to JINI. Current research is addressing that and of the late attempts have been made to extend web services to the mobile realm. An example for this is the project titled *Cascading Web Services*, which interlinks Bluetooth mobile devices with web services from the fixed network.

2.3.3 CORBA

CORBA[15] (Common Object Request Broker Architecture) is an open protocol standard designed to interlink applications (i.e. clients, services) distributed over a heterogeneous environment. CORBA's so-called broker object (ORB) locates and activates objects, somewhat similar to how the Lookup Service works in JINI, yet the ORB has more complex functionality. The ORB is able to hide platform differences, yet that alone does not make CORBA suitable for interlinking application frameworks that span across wired and mobile environments. CORBA's heavyweight middleware solution is what constitutes a problem for extending it to mobile networks. Of the late a new trend of CORBA-based research has been emerging: Projects investigating real-time and embedded CORBA solutions.

CORBA is comparable in many ways with JINI and MSP: It is relatively heavyweight, it is not fully self-configurable and relies on central networking components. CORBA's ORB uses a technology similar to the Java RMI used by JINI. Given the similarities between CORBA and JINI systems it would be interesting for future work on CDA to compare the design paradigms used in agile CORBA implementations with the paradigms designed in this thesis work.

2.3.4 JXTA and Peer-to-peer Frameworks

JXTA(TM)[16][17] is a Java technology suitable to create large distributed application frameworks based on peer-to-peer (P2P) communication model. JXTA is essentially a set of open protocols which enable devices, whether mobile or fixed, to communicate and collaborate in a P2P manner, forming

self-organized and self-configured peer groups without the need of a centralized management infrastructure. JXTA can incorporate devices with non-ideal network locations, like being behind firewalls, NAT servers, in public vs. private address spaces. The mechanism that makes this possible is dynamic routing built into the JXTA protocols. Routing information is included in the messages, as well as in the local caches of peers. JXTA supports communication between devices that run on different network transports. The JXTA protocols use an open messaging standard, which is usually implemented in XML. The generic message format ensures platform-independence.

JXTA and P2P frameworks offer a radically different alternative to the service oriented infrastructures that we are dealing with in the current project (i.e. MSP, JINI, CDF).

2.3.5 EgoSpaces and Mobi-Blog

We find numerous related projects on service oriented federation systems, yet there is scarcely anything on solutions that study the interlinking of wireless/mobile networks with fixed environments in the field of context aware computing. EgoSpaces and Mobi-Blog are examples for the latter, though both of them give little attention to the interlinking of mobile and fixed frameworks and concentrate more on the context management aspects.

EgoSpaces[18] project has developed a middleware framework that delivers context information in an abstract form to applications running in context-aware mobile networks. The EgoSpaces solution helps programmers to focus on high-level interactions between programs and it helps to employ abstract context specifications in transient and opportunistic mobile network environments. Implementing EgoSpaces on mobile nodes is as simple as extending a couple of API classes. EgoSpaces abstracts away from the actual network topology by using the Source Initiated Context Construction (SICC) protocol and its network abstraction. The EgoSpaces projects refers also to other context-aware middleware solutions applied in mobile networks, e.g. LIME, TuCSoN, GAIA, etc.).

Mobi-Blog[19] represents another example of a highly pervasive, highly context-dependent mobile application framework. Mobi-Blog claims to implement ubiquitous context sharing for applications running on hand-held devices.

Though neither EgoSpaces, nor Mobi-Blog adopt JINI-based service oriented approach like the current project does, it might be valuable to study their context management infrastructure and apply it to enhance CDA, more specifically Ahoy, with context-aware modules.

3 Design and Architecture

In this section we present the design steps and decisions that have guided the development of the CDA architecture. We begin with design guidelines and by making a choice for the design paradigm. Following that we analyze differences and overlaps in Ahoy and MSP regarding the processes of *context announcement*, *context discovery* and *context distribution*. For each of the three processes we decide on a mutual CDA-specific solution, creating in the process the final CDA architecture.

3.1 Guidelines

3.1.1 Bridge-like Design Paradigm

As we have seen in earlier sections, different design paradigms could be adopted to address the framework interoperability challenges dealt with by this thesis. The question remains: Which design paradigm to choose for the CDA protocol?

The nature of differences between Ahoy and MSP make our choice difficult. A CORBA-like federation system would be too much overhead for lightweight Ahoy. The peer-to-peer approach of JXTA does not fit naturally with MSP's middleware design, though the paradigm might hold some potential. A service-oriented design seems most promising since MSP complies with that paradigm already. Ahoy, however, does not seem well suited for service oriented architecture. A strictly SOA-based approach would be too much overhead for Ahoy. Keeping in mind the essential requirement that Ahoy and MSP should be left as much as possible unaltered, we are left with one choice: A hybrid, bridge-like architecture design, which combines MSP's heavyweight SOA-based architecture with Ahoy's lightweight modular programming interface.

The bridge-like design implies that we modify the existing protocols only where absolutely necessary. By doing so we keep the overhead caused by the new CDA protocol minimal.

3.1.2 Prototype CDA Framework

By prototype CDA framework we mean a CDA network consisting of the minimum necessary components capable to model the key properties of CDA. We use the same prototype framework throughout the project with minor extensions added to it in the testing phase.

The CDA prototype framework as we define it contains the following components: A CDA node, a regular Ahoy node, an MSP service, an MSP client, a Surrogate Host and a JINI Lookup Service. Figure 8 illustrates the components of the CDA prototype framework.

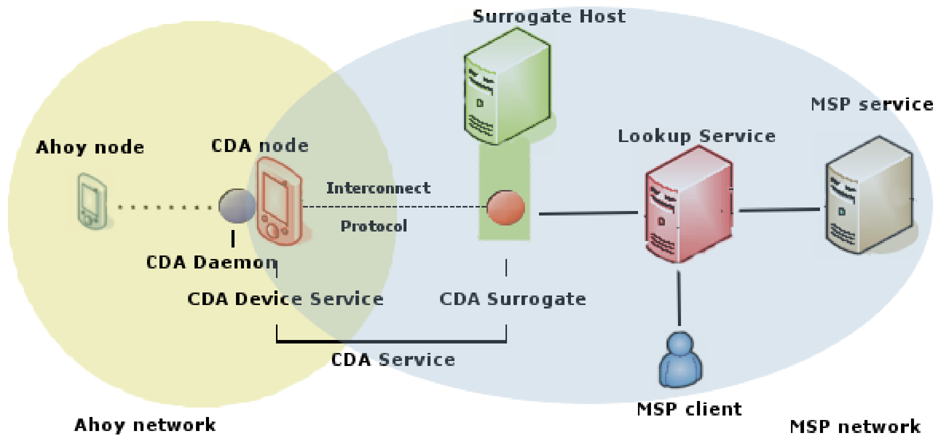

Figure 8: *The components of the CDA prototype network: CDA Service hosted on the CDA node, modeled after NMS; A regular Ahoy node; Surrogate Host; JINI Lookup Service; MSP client and MSP service. The components are described in the text.*

CDA node: The CDA node is the physical bridge between Ahoy and MSP and is part of both frameworks. It interlinks with MSP through an NMS module, the solution ready offered by MSP. The NMS model has been described earlier in Section 7. In Figure 8 we can see its subcomponents: CDA Device Service, CDA Surrogate and the Interconnect protocol. For the CDA node to also interlink with the Ahoy network it must act like an Ahoy node, so it is implied that the CDA node must have an Ahoy daemon running (with optional Ahoy clients). The daemon we call CDA Daemon. The CDA node needs sufficient hardware to support both the NMS and Ahoy bridging modules.

Ahoy node: A regular Ahoy node is also part of the CDA prototype framework. Together with the CDA node they form a minimal two-node Ahoy network on which we can run tests. Both this and the CDA node can host Ahoy clients and services.

MSP client: In the context of CDA by MSP client we mean a client application that queries for context sources located in Ahoy. Regular JINI

clients that participate in CDA we would also call MSP clients, yet they are not relevant for studying CDA mechanisms. A different name for the MSP client is CDA client. Ahoy clients that query for MSP context sources are also CDA clients, so to avoid confusion, the framework of origin is used to refer to each CDA client.

JINI Lookup Service: The JINI Lookup Service is essential to context announcement and discovery in both JINI and MSP, and also in CDA.

MSP service: In the context of CDA by MSP service we mean a JINI service that can be discovered by Ahoy clients. With this new definition we do not consider the NMS an MSP service. Or rather, the NMS is a very special kind of MSP service. We also do not use the term MSP service for regular JINI services that are not meant to be discovered by Ahoy clients.

Surrogate Host: The Surrogate Host is a necessary helper component. Its properties have been described earlier.

3.2 CDA Architecture

In this section we study differences and overlaps in the context manipulation mechanisms of Ahoy and MSP. The goal is to find mutually feasible CDA mechanisms that can replace or complement the protocols of the two underlying frameworks. The context manipulation mechanisms that we consider are *context announcement*, *context updates*, *context discovery*, and *context distribution*. Our study is guided by the requirements and research questions as defined in the introduction.

3.2.1 Announcing Ahoy Context Sources to MSP

In this section we concentrate on Ahoy context sources and alternative solutions for how to register them with the MSP framework.

3.2.1.1 Can we use MSP's announcement mechanism to register Ahoy context sources with the MSP framework?

Services in MSP are required to provide their Java interface upon registration. If we wanted to register Ahoy context sources the same way, they would need to have Java interface definitions. Regular Ahoy services do not

have Java interfaces and even if they had, the current Ahoy implementation does not have a message format to transport the interface definitions. CDA nodes, however, have a ready made solution for this: They run an NMS module which interlinks them with MSP.

The NMS Surrogate allows CDA nodes to register with MSP like any regular JINI service would. If the NMS model solves the problem of registration, why not use it as solution for all Ahoy nodes? The answer is simple, yet it leads to further questions: NMS technology is too heavyweight for the resource-constrained ad-hoc devices, only few nodes with enhanced hardware can host it. Those few NMS-compatible hosts are the CDA nodes. That being the case, would it suffice to register CDA nodes only and have MSP clients query for Ahoy services through the registered CDA services as bridges? Without a CDA Service storing explicit references to individual Ahoy services the MSP clients cannot know which Ahoy service they can access through which CDA Service and that would lead to highly inefficient context discovery. We conclude therefore that each CDA Service should store explicit references to the individual Ahoy services they know about.

We now face the questions: How can the CDA services collect information about the individual Ahoy services and how can they register that information with MSP?

Each CDA node has an Ahoy daemon running locally. The daemon stores the latest Ahoy announcement. What we need to do is to transfer that knowledge from the Ahoy daemon to the locally running NMS module, which is the CDA Device Service. Initially, however, the Ahoy daemon and CDA Device Service cannot communicate with each other. How can we interlink them? We know that the Ahoy daemon uses two types of communication: It uses local Unix sockets to communicate with the Ahoy clients and uses UDP ports to communicate with other Ahoy daemons residing on different nodes. The Java-based NMS implementation does not support communication over Unix sockets, so we are left to attempt communication through the UDP sockets. As solution to this we propose that the CDA Device Service and the Ahoy daemon communicate with each other through UDP sockets across the loopback network interface. That way we keep the Ahoy daemon mostly unaltered and the messaging between the two stays local, not affecting other segments of the Ahoy network.

The other question we have not answered yet: How to register the information about individual Ahoy services with the MSP framework? The Surrogate of the CDA Service registers with MSP like regular JINI services would do, through the JINI Lookup Service. When it registers, it can provide a number of optional service attributes that get registered in its service entry. It can store the references to Ahoy services in one such attribute.

JINI mechanisms exist to keep the service attributes up-to-date. The CDA Surrogate, however, first needs to obtain the Ahoy service references from the CDA Device Service, which at its turn received it from the Ahoy daemon running on the CDA node. Figure 9 illustrates the complete flow of announcing Ahoy context sources in MSP.

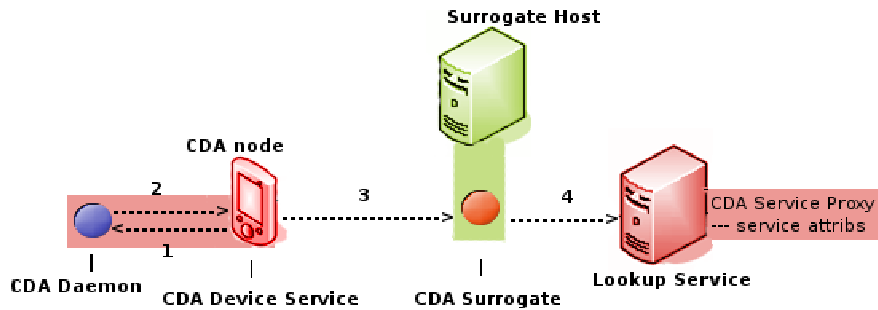

Figure 9: *Announcing Ahoy context sources with MSP. 1: When initializing, CDA Device Service sends an Ahoy update message to the CDA Daemon. 2: CDA Daemon replies with a response message containing the latest announcement. 3: CDA Device Service registers its Surrogate and sends the Ahoy service information. 4: CDA Surrogate registers with the JINI Lookup Service the CDA Service Proxy and service attributes, among which the information on Ahoy services.*

3.2.1.2 In what format should the Ahoy service information be transferred to the MSP framework?

In the previous section we have seen that the announcement of Ahoy context sources starts with a dialog between the CDA Device Service and Ahoy daemon. As result of this dialog the CDA Device Service obtains the latest announcement from the Ahoy daemon. The latest announcement is an ABF array. Knowing that, we need to consider two options: Either the CDA Surrogate registers with the JINI Lookup Service that ABF array as service attribute, or, somewhere on the path between the CDA Device Service and the JINI Lookup Service the ABF array gets converted to another format.

Registering the ABF seems to be the simplest solution, since it would require no extra conversion or computation. The MSP framework, however, would have to deal with the ABF format when handling queries for Ahoy context sources. The ABF representation format not being native to MSP, we ex-

pect this to cause numerous difficulties. The consequences are discussed in Section 3.2.3).

If we chose the other alternative and wanted to convert the ABF to another format, only one choice would be rewarding: If we converted the ABF to a format that MSP supports. When MSP clients discover context sources they commonly search for services by their string-based name. If we could convert the ABF to a list of the Ahoy service names that it represents, we are sure that MSP client queries could deal with the new format. The conversion is however not possible. The computation of ABF involves one-way hashing, which cannot be reversed to obtain back the initial string-based service names. In lack of this option we are left with the alternative to register the ABF itself with the JINI Lookup Service.

3.2.1.3 How to keep the registered Ahoy service information up-to-date in MSP?

The registered Ahoy service references need to be kept updated. Every time the latest announcement of the Ahoy daemon on the CDA node changes, the MSP framework needs to be notified. We assume that the Ahoy service references are kept in ABF format in the Lookup Service.

The update flow of Ahoy services in MSP is illustrated in Figure 10.

A second update mechanism is required to make sure that the CDA Service is not removed from the JINI Lookup Service. The mechanism is implemented by MSP's NMS model and it involves continuous keep-alive dialog between CDA Device Service and CDA Surrogate. If the keep-alive dialog is disrupted, the Surrogate unregisters with the MSP framework.

3.2.1.4 Should we apply filtering on the Ahoy services that get registered with the MSP framework? If yes, what mechanism should we use?

A concern why we would not want to have too many CDA services registered with the MSP framework is that the MSP clients querying for Ahoy services would have a long list of CDA services to iterate through, slowing down the query process. No mechanism is in place for this, yet it should be considered for future research. The regulation can be done in two places: Either during registration of CDA services in the JINI Lookup Service or during the response to an MSP client query.

Changing the number of regular non-CDA Ahoy services, however, does not have any drastic affect. An increase in the number of Ahoy services could

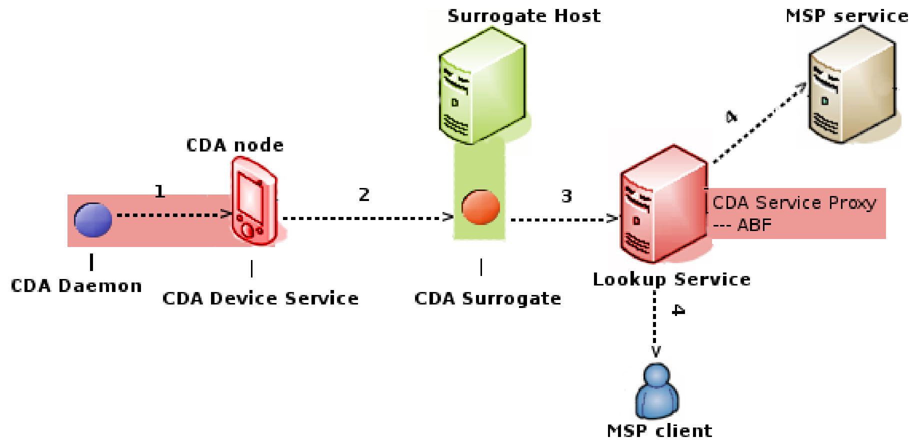

Figure 10: *The update flow of Ahoy services registered with MSP. 1: CDA daemon broadcasts its latest announcement. 2: CDA Device Service receives the latest announcement and sends an HTTP Interconnect message to CDA Surrogate containing the received announcement. 3: CDA Surrogate caches the new latest announcement and updates its service entry in the Lookup Service through built-in JINI mechanism. 4: JINI lookup service notifies all listeners of the CDA Service about the state change through built-in JINI notification mechanism.*

cause an increase in the probability of false positives, since the ABF array would be more densely packed. It is Ahoy's responsibility to deal with this phenomenon.

Finally, we consider the filtering capabilities of Ahoy. The question is whether Ahoy supports any mechanism for classifying its context sources. That is not the case. The current implementation of Ahoy offers no support for context classification, with one exception: The ABF array, by its design, allows to represent the approximate distance of services. Ahoy has therefore a built-in property that can be used to qualify services in function of their proximity values. For Ahoy to become more context aware, it would require a number of extensions.

3.2.2 Announcing MSP Context Sources to Ahoy

Having looked at the announcement flow of Ahoy services we now consider the registration of MSP context sources with the Ahoy framework. Our study is guided by the same questions that we had for the Ahoy context sources.

3.2.2.1 Can we use Ahoy's announcement mechanism to register MSP context sources with the Ahoy framework?

In Ahoy the announcement of context sources happens in two ways, at two different levels: At a local level an Ahoy client announces its service to the daemon. At the network level a node sends its latest ABF to the one-hop neighbors, announcing them about a change.

In analogy with the local announcement the CDA Device Service could *pretend* to be a client and announce MSP context sources one by one, sending their service identifier to the local daemon. The problem with this solution is that the Ahoy client queries would fail to find the MSP services. The daemon would return its own IP address to the query, since it registered the MSP services as offered locally. For the queries to work we would need to modify the Ahoy daemon interface significantly. Given these drawbacks we believe that the MSP context source identifiers should be converted to ABF before they reach the daemon. That being the case, the CDA Device Service can *pretend* to be a neighbor of the local Ahoy daemon and announce the MSP context sources in ABF format through the local network interface. One implication is that MSP context sources must each have a unique service name or else they cannot be represented in ABF form. Another implication is that somewhere along the announcement path the MSP service names have to be converted to ABF.

In the hope that simpler alternatives exist, we consider two options when we do not convert MSP context information to ABF. For instance, we can make the CDA node *pretend* to offer all services and have the Ahoy daemon forward incoming Ahoy queries to the CDA Device Service, which can decide alone or with its Surrogate whether an MSP context source is available. It is not difficult to see that this solution would create many redundant queries and unnecessary waste of resources. Another drawback is that we purposefully generate false positives. Another option is if the CDA Service announces itself in Ahoy as a special service through which Ahoy clients can gain access to MSP context sources. This would imply, however, that Ahoy clients discover Ahoy and MSP context sources differently, introducing lack of transparency, which is unwanted.

Not having a better alternative we settle with the option to register MSP context sources with Ahoy in ABF format.

3.2.2.2 In what format should the MSP service information be transferred to Ahoy?

It has become clear from the previous section that MSP context sources should be represented in ABF form and that the conversion from the service

identifiers to ABF should happen before the ABF reaches the Ahoy daemon. In this section we study each component along the announcement path and choose a location for ABF conversion. The components we consider are: JINI Lookup Service, CDA Surrogate and CDA Device Service.

It is not feasible to have the JINI Lookup Service perform the ABF conversion. The LS is a core element of JINI and should stay unaltered. Making the CDA Device Service perform the conversion is also not an ideal solution, since the computations would likely exhaust the resource-constrained device on which the DS is running. The component we are left with is the CDA Surrogate. The Surrogate is indeed quite suited for the work. It is not affected by resource constraints like the CDA Device Service is and we can extend its functionality without affecting Ahoy or MSP. In the current CDA architecture the CDA Surrogate is best suited for this purpose.

The mechanism of announcing MSP context sources has been cleared down to its details. Figure 11 illustrates the complete announcement flow.

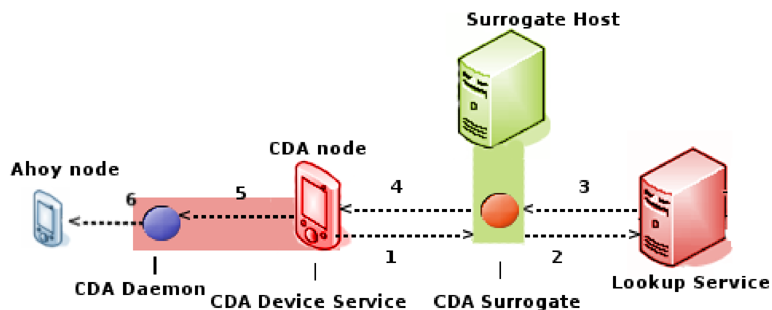

Figure 11: *Announcing MSP context sources with Ahoy. 1: CDA Device Service activates and registers its Surrogate. 2: CDA Surrogate registers its Service Proxy with the LS and initiates a query for MSP context sources. 3: CDA Surrogate receives the MSP context source references from the LS and it creates a local cache to store them. 4: CDA Surrogate sends the computed ABF to the CDA Device Service as an HTTP Interconnect message. 5: CDA Device Service announces the ABF to the CDA daemon through the local network interface. 6: CDA daemon recomputes its latest announcement and broadcasts it to its neighbors.*

3.2.2.3 How to keep the registered MSP service information up-to-date in Ahoy?

We have assumed in the previous section that the CDA Surrogate queries for MSP context sources and stores them in a local cache. That solution would not allow to keep the list of MSP services up-to-date in real-life. JINI offers a better solution instead. JINI components can register as service listeners and receive change events about the services of their interest. We believe this mechanism should be used by the CDA Surrogate. The Surrogate can register as service listener for MSP context sources and receive events from the JINI LS every time such a service has been added, removed or modified. This change affects one step in the announcement flow of MSP services: When the CDA Surrogate registers with the LS it also registers as service listener to MSP context sources. The update flow of MSP services is essentially the same as their announcement flow (steps 3-6 in Figure 11), except for one detail: CDA Device Service is required to send regular Ahoy keep-alive messages to the local daemon or else the registered MSP context sources get removed. In addition we note that an update flow is initiated when the CDA Surrogate receives a service listener event from the LS. The same event makes the Surrogate update its local service cache.

3.2.2.4 Should we apply filtering on the MSP services that get registered with the Ahoy framework? If yes, what mechanism should we use?

When the number of MSP services registered in Ahoy is high we face a problem: The MSP services would all get mapped to the same filter in the ABF array leading to increased occurrence of false positives. The reason why MSP context sources get mapped to the same filter is because they are all considered to be one hop distance away from the Ahoy network originating from the fake Ahoy neighbor which is the CDA Device Service.

One solution is to limit the number of MSP services that get cached in the CDA Surrogate. We can choose between at least two options. One option is to filter all eligible MSP context sources by some of their service attributes. Another option is to use an adjustable threshold value beyond which the Surrogate does not accept new MSP context sources. The threshold would need to be a function of the Ahoy *filter size* configuration parameter, since it is known that the larger the filter and the less services mapped to it, the less the probability of false positives. Implementing these ideas is future work.

For the CDA architecture our choice went for a different filtering solution. We screen MSP services for Quality of Context (QoC) parameters and in function of those parameters the services get mapped to different rows in the ABF. Less desired QoC values get mapped to lower rows, while more desired QoC values map to higher rows. This way we translate QoC values

to virtual proximity of services. The mapping algorithm is not fine-tuned in this project, yet the capability to introduce this kind of context awareness has been integrated with CDA.

3.2.3 Discovery of Ahoy Context Sources by MSP Clients

Having dealt with the announcement of context sources until now, in this section we turn our attention to the discovery of context sources. In particular, we consider alternative CDA mechanisms for the discovery of Ahoy context sources by clients in the MSP framework.

3.2.3.1 What discovery mechanisms fit the established announcement of Ahoy context sources in CDA?

It has been concluded in earlier sections that Ahoy context sources should be represented in the MSP framework in ABF format. The ABF is stored as service attribute in the Lookup Service.

Problems arise when the LS receives an MSP client query. The MSP client, being in search for a specific Ahoy service, discovers registered CDA services and when it finds one, it checks its ABF service attribute to locate whether the service it is looking for is known to that CDA Service or not. To perform the last step the client must compute the Bloom filter of the queried service name and match the computed filter against the found ABF. A match indicates that the CDA Service has knowledge about the contact details of the specific Ahoy service. It is a concern for us that every MSP client has to implement the Bloom filter computations and perform those computations during each query. The solution seems to lack modularity and scalability, and it would exhaust resource-poor clients. We therefore consider alternatives.

The Lookup Service and CDA Device Service cannot perform the Bloom filter computations for the same reason why they were not suitable to do the conversion of MSP service identifiers to ABF. Regarding the LS it is not desirable to alter its interface, while the CDA Device Service has limited resources. The same arguments that made us choose for the CDA Surrogate to compute the ABF from MSP service identifiers also hold here. Hence, the best candidate to perform Bloom filter conversion and query string matching during the discovery of Ahoy services is the CDA Surrogate. There is one disadvantage of this solution: When MSP clients discover the CDA services they do not know which of them store reference to the Ahoy context source the client is interested in. To find out which CDA service offers the chosen Ahoy context source the Surrogate must compute the Bloom filter of the

query string and match it against the stored ABF. That means for all CDA services that do not offer the Ahoy context source the computations are redundant.

In the hope to find a more efficient solution, we consider a radical alternative. We design a new JINI service from scratch with the capability to offload both the CDA Surrogate and the MSP clients. The new component has much functionality in common with the Context Distribution Service (CDS) offered by the CDF framework as described in Section 2.3.1. We call this component the CDA Broker service. The Broker service takes over the functions of the CDA Surrogate that play a role in the discovery of Ahoy context sources. Hence, instead of the CDA Surrogate the Broker will handle incoming queries from MSP clients. The change implies that MSP clients, when initiating a query, first have to discover the Broker service. When found, the client calls a remote method on the Broker service which returns the contact information of the queried Ahoy service. The client does not need to know about CDA services, neither about Bloom filter computation, since all steps are performed by the Broker and the client only gets the end result. The Broker keeps a local cache of CDA services, so that when it receives a query, it can look through the list and match the query string's Bloom filter with the stored ABF. For finding a match the Broker service should not make any remote calls to the CDA services, it should rely instead on the cached list of services and their attributes. The Broker registers with the Lookup Service as a listener to CDA services. Every time when a listener event is received, the cached list gets updated.

With the Broker Service added the previously established mechanisms of Ahoy service announcement and update require modifications. Figures 12, 13 and 14 illustrate the announcement, update and discovery flows, respectively, after having added the Broker Service to the architecture. The designs shown in these figures have been obtained after we created the implementation prototype and found out that the user-defined service entries were not recognized. The announcement and update flows have been ulteriorly modified to be in line with the implementation. Without the user-defined service attributes the Ahoy announcements and updates reach the Broker Service through a callback interface.

3.2.4 Discovery of MSP Context Sources by Ahoy Clients

Having considered in the previous chapter the discovery of Ahoy context sources, in this section we concentrate on how MSP context sources are discovered by Ahoy clients.

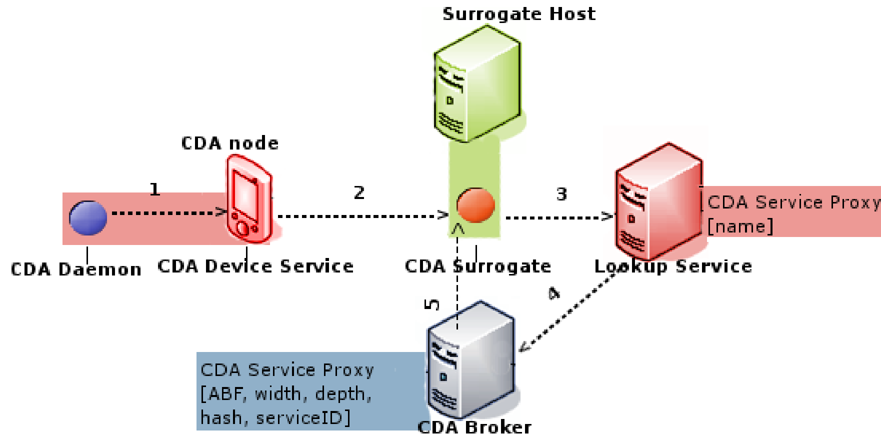

Figure 12: *Announcement flow of Ahoy services, with CDA Broker. 1: CDA daemon broadcasts its latest announcement. 2: CDA Device Service sends the ABF to CDA Surrogate over the Interconnect protocol. 3: CDA Surrogate registers itself with the Lookup Service. 4: The Lookup Service sends service added event to the Broker Service and the Broker adds the new CDA Service to its cache. 5: Broker Service registers a callback interface with the CDA Surrogate.*

3.2.4.1 What discovery mechanisms fit the established announcement of MSP context sources in CDA?

We concluded in earlier sections that MSP context sources should be represented in Ahoy in ABF format and that the best suited component to convert the MSP service information to ABF is the CDA Surrogate. Now we follow the path how Ahoy clients discover the registered MSP context sources.

The Ahoy client query originates in the Ahoy framework and eventually reaches the Ahoy daemon that runs on the CDA node. Thanks to how the MSP service references get registered, the daemon believes that the MSP services are offered by a direct neighbor, one that has the local network interface as address. Hence, the daemon forwards the query to this neighbor of his, which is how the query reaches the CDA Device Service. From there the query travels to the Surrogate. Since the Surrogate has a list of cached MSP context sources, it seems logical that the Surrogate answers the query. This can be done in the following way: The Surrogate iterates through the list of MSP context sources and matches the received query string against their names. When a match is found, the Surrogate calls a remote method on the respective MSP context source and obtains its contact details. The matching should not involve any remote calls except for obtaining the contact details.

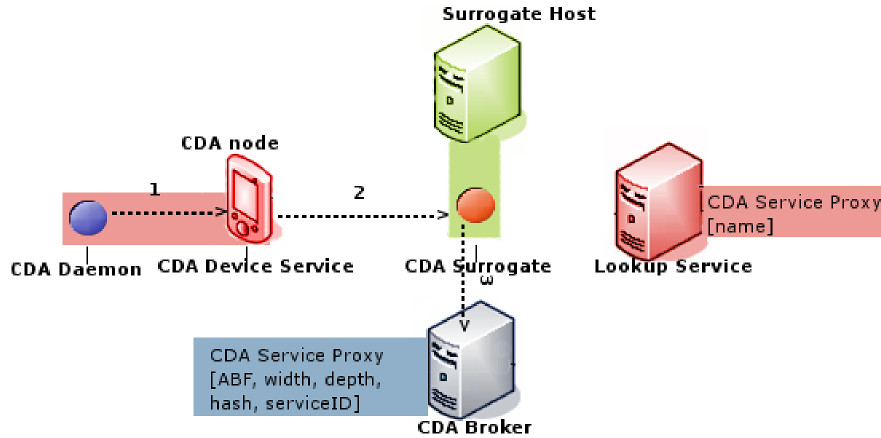

Figure 13: *Update flow of Ahoy services, with CDA Broker. 1: CDA daemon broadcasts its latest announcement. 2: CDA Device Service sends the ABF to CDA Surrogate over the Interconnect protocol. 3: CDA Surrogate updates the Broker Service through its callback interface.*

The Surrogate seems to handle the discovery of MSP context sources well. We would not look any further for alternatives if we had not introduced a new component while we were optimizing the discovery mechanism of Ahoy services. This new component is the Broker Service. We believe that the Broker can also prove useful in the discovery of MSP context sources. It is required for the discovery of Ahoy context sources that the Broker registers as listener to CDA services and keeps a cached CDA Service list. It is also required that the Broker performs Bloom filter computation for incoming MSP queries, matching the computed filter against ABF arrays stored with the CDA services. The CDA Surrogate performs mostly the same functions for MSP context sources. In addition it also computes the ABF for MSP service representation. We believe that implementing the same functionality in two distinct components is redundant and error-prone. Also, it affects the scalability of the CDA protocol and is unwelcome for code maintenance. We believe that the Broker component should implement all functions and handle the discovery of both Ahoy and MSP context sources. Another advantage of this design is that the CDA Broker, by having access to both CDA and MSP service lists, can enhance query results with context classification. In other words, the Broker could perform ranking and selection among multiple matches and return always the optimal one.

Figures 15 and 16 show the announcement and discovery flows of MSP services after having added the Broker Service to the CDA architecture. The update flow of MSP services is same as the announcement flow, only

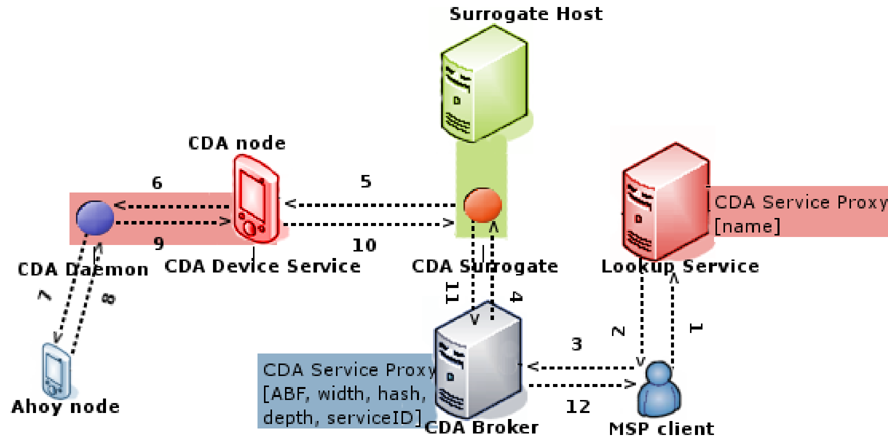

Figure 14: *Discovery flow of Ahoy services by MSP clients, with CDA Broker. 1: MSP client queries the JINI LS for CDA Broker service. 2: MSP client receives the proxy of CDA Broker service. 3: MSP client starts query. 4: Broker service iterates through its cached CDA Service list and for each service matches the query string's Bloom filter against the found ABF. 4: If a match is found, CDA Broker contacts the CDA Surrogate and passes on the query string. 5: The query string reaches the Device Service over the Interconnect protocol. 6: Device Service sends query to the CDA daemon. 7: CDA daemon forwards the query to knowledgeable neighbors. 8: CDA daemon receives a query response. 9: CDA daemon sends query response to the CDA Device Service. The response travels back the same road where it came until it reaches the MSP client.*

the Broker receives a service change event from the Lookup Service instead of a service added event. It has been assumed that MSP services do not require user-defined entries. If they were to require them, either we need to make sure that user-defined entries are integrated with CDA or, the Broker Service would need to register a callback interface with the MSP services the same way how it does it with the CDA services.

3.2.5 Context Distribution Between Ahoy and MSP

In previous sections we have considered each of the CDA announcement, update and discovery mechanisms. The topic of the current section is CDA context distribution.

We take as starting point the context distribution mechanisms of Ahoy and MSP. The current Ahoy implementation does not support context distribution. Ahoy discovery returns the IP address and port number of the queried

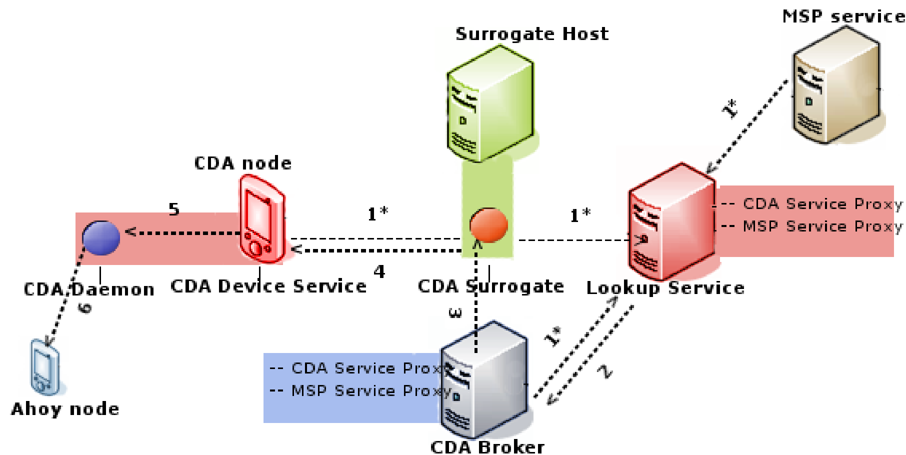

Figure 15: *Announcement of MSP services in Ahoy, with CDA Broker. 1*: MSP services register with LS. 1*: CDA service registers with LS. 1*: Broker registers with LS and registers as listener for both CDA and MSP services. 2: LS sends service added event to Broker. 3: Broker Service updates its caches, registers a callback interface with all active CDA services, computes the ABF representation of MSP services and sends the ABF to CDA Surrogate. If we were also interested in receiving updates from the MSP services, in this step the Broker Service would have to register a callback interface with them. That is required because the user-defined service attributes are not recognized. 4: CDA Surrogate sends ABF-based MSP announcement to Device Service through Interconnect protocol. 5: Device Service announces ABF of MSP services to CDA daemon. 6: CDA Daemon recomputes its latest announcement and sends it to neighbors. The steps marked with asterisk (*) happen independent from each other.*

service and stops there. MSP on the other hand has Java RMI-based distribution mechanism. As return value to service discovery an MSP client receives the Java interface (i.e. proxy object) of a remote server. The client can set up a dialog with the server by calling remote methods on the proxy object. RMI offers a flexible solution for context distribution. Knowing that Ahoy context distribution must be built from ground, it seems like a practical solution to have Ahoy adopt the mechanism used by MSP, namely Java RMI.

Implementing Java RMI on top of Ahoy raises numerous challenges. Firstly, relatively heavyweight Java RMI modules would need to run on each node putting the resources of the mobile device at risk. Secondly, the Ahoy protocol would need to be modified to transfer serialized objects or the reference to a public URL where the RMI service proxy can be dynamically downloaded from. Thirdly, the transfer of serialized objects would cause increased

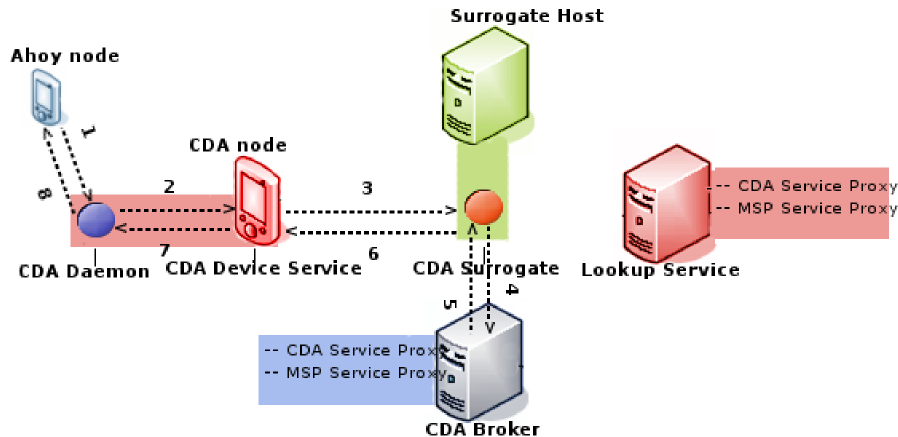

Figure 16: *Discovery flow of MSP services by Ahoy clients. 1: Ahoy client sends query for MSP service that the daemon knows about. 2: CDA daemon forwards the query to CDA Device Service. 3: Device Service sends the query to Surrogate through Interconnect protocol. 4: Surrogate calls remote method on CDA Broker callback interface. 5: CDA Broker checks the query string against cached MSP services and sends response. 6: Surrogate forwards the query response to Device Service through Interconnect protocol. 7: Device Service sends the response to CDA daemon. 8: CDA daemon forwards the response to Ahoy client.*

network traffic which is also a potential hazard to mobile devices with limited resources. Furthermore, if RMI was used, MSP clients would require to know the Java interface definition of every Ahoy service to be able to find them. In other words, the Java interface definition of the Ahoy services would have to be compiled into the MSP client modules. A work-around to this problem would be to implement all Ahoy services by extending one generic Java interface, in which case only that interface needs to be compiled into the MSP client modules. There are some more issues that make Java RMI little suited for Ahoy. RMI sessions are sensitive to end-point mobility, so they are expected to drop frequently in a mobile environment. Also, it is known that RMI-based communication has high latencies, an undesired property in mobile networks where quick adaptation is required. Given the list of these issues, we are confident that standard Java RMI should not be considered for CDA context distribution.

An alternative is to use lightweight RMI technologies. We found only few promising projects in this area (see [21] and [22]). Since none of the projects wished to disclose the source code of their solution, we have also disregarded this option.

A final alternative that we considered is Remote Procedure Call (RPC).

RPC is in fact the precursor of RMI, a lightweight, non object-oriented communication technology. The simplest and most lightweight RPC design is XML-RPC[23]. Of the numerous implementations we selected Apache XML-RPC [24] with the intention to integrate it with CDA. An XML-RPC client requires the following parameters to establish connection with a server: The name of the remote method(s) that the server offers, the URL location of the host where the server is running and the port number on which the server is listening. The last two parameters, an IP address and port number, are the return value of an Ahoy query. As long as the RPC server runs in the document root, it can be found based on the IP address only. If, however, it runs in a subdirectory of the web server, then the URL location would require text elements, which value can currently not be transferred by Ahoy. To avoid having to change the Ahoy discovery mechanism, we prescribe for CDA services that the RPC server module runs in the document root and that a generic method name is used for all RPC servers. The method name is *getContext*. With these assumptions neither Ahoy, nor MSP discovery need to be modified. The CDA discovery mechanism returns to both Ahoy and MSP client queries an IP address and a port number, which are sufficient to establish connection between an RPC client and server. It should be noted that Ahoy supports only IPv6 addresses and so does CDA. The returned IP address can refer to some central web server or to a specific MSP or Ahoy node. Without further modification Ahoy nodes can return only their own IP address. MSP services can be designed freely to return the IP of some central server, yet for them it is of no relevance. The Ahoy nodes would benefit more from returning the address of a central web server, since it would mean they themselves need not run web servers locally. In the current setup, if an Ahoy service wants to run an RPC server, it must have a web server running. XML-RPC implements a web server which is light enough to run on a mobile device, yet more test results are needed before we can surely state that. In the current project the RPC server and client templates have been developed and tested independently, yet they have not been integrated with CDA due to lack of time and unforeseen implementation difficulties. Due to that we cannot say anything conclusive about how the XML-RPC web server performs on a mobile node.

3.2.5.1 Routing Challenges

Both Java RMI and Java RPC are one-to-one communication models. The client and server communicate directly, without intermediary components. An important requirement for both RPC and RMI is that the server components have a publicly available, routable IP address. RMI and RPC servers behind a Network Address Translation (NAT) do not work, since they are unable to send response to the client.

In numerous real-life scenarios mobile networks have only one gateway to the public Internet and use local IP addresses within the network. Ahoy has kept consideration with this problem and supports only IPv6 addresses, in which case all nodes have public addresses and routing problem due to NAT or other hindrances do not occur. We have followed the same design for CDA: All CDA components are required to have IPv6 addresses and IPv6 routing must be enabled in the networks where CDA is running. If we were not to do this, we would be required to implement complex NAT and routing functionality in the CDA nodes to make sure protocol messages can cross the border of MSP and Ahoy. With IPv6 addressing that is not necessary. For the Ahoy network it is necessary to have routing in place, since Ahoy queries rely on an underlying routing protocol.

This project has not gone in further detail about routing issues that can become a hindrance, since it would take away focus from the main goal of the project.

3.2.5.2 Relayed Context Distribution Mechanisms

Relayed context distribution mechanisms are an alternative to the one-to-one models we considered until now, namely RMI and RPC.

Relayed context distribution would mean that clients and services are not having a direct dialog like they have with RMI and RPC, rather, they would communicate indirectly, passing the messages through each component along the way. Looking at the MSP framework first, the messages would likely have to cross through the CDA Broker, the CDA Surrogate and the CDA Device Service. In Ahoy they would cross from node-to-node. Ahoy would require an entirely new protocol layer implemented, since in its current form it does not support any context distribution. We believe that instead of developing new protocols that are unique to the problem at hand we rather reuse existing industrial or open standards.

Another argument against relayed communication model is the problem of interpreting the returned end values. The data types returned by the context distribution protocol can be diverse, e.g. numeric, textual, streaming video or other. We would need to make sure that the various data types get interpreted and processed correctly by all components involved, most importantly by the clients. A textual template-based (e.g. XML) representation would prove flexible enough for this purpose. If, however, we are ready to choose an XML-based technology, we better choose one of the numerous established solutions instead of implementing yet another new protocol. Other, more general disadvantages of the indirect communication model are that it results in a more centralized, less flexible architecture.

Relayed context distribution mechanisms do not seem to have advantages over one-to-one solutions. We are convinced that the XML-RPC solution we chose is the best we get.

3.3 CDA Architecture: Final Conclusions

In the previous sections we covered all functional segments of the CDA protocol. In the process the CDA architecture has gained final shape. The core components of the CDA protocol are: (CDA) Broker Service, CDA Service and CDA Daemon.

Broker Service: Its main task is to handle Ahoy and MSP client queries and to maintain, i.e. update, the cached CDA and MSP service lists. It performs ABF and Bloom filter computations and classifies context sources.

CDA Service: It consists of the CDA Surrogate and CDA Device Service. Its main role is to interconnect the mobile and fixed environments through the Interconnect protocol and to interconnect MSP with Ahoy by exchanging Ahoy messages with the CDA Daemon across the local network interface of the CDA node.

CDA Daemon : It is the Ahoy daemon running on the CDA node. Since it interacts directly with the CDA Device Service, this daemon is different from regular Ahoy daemons, it has slightly enhanced functionality.

The core CDA components, their events and supported message types, are described in further detail in Section 4.

Two third-party helper components also form an essential part of the CDA protocol: Lookup Service and Surrogate Host. Other user-defined helper components are MSP clients, MSP services, Ahoy clients and Ahoy services.

4 Protocol Specification

4.1 Functional Description

In this section the CDA protocol is described in terms of the events and actions associated with its components. The three core components for which we discuss the event-response cycles are: 1. **CDA Service**, which we split into ***CDA Device Service***, ***CDA Surrogate*** and ***CDA Service Proxy***; 2. **Broker Service** and 3. **CDA Daemon**.

4.1.1 CDA Device Service

Startup

The startup of the CDA Device Service is the moment when the CDA Service becomes alive. The Device Service performs the following actions before entering idle state:

- It reads the service properties file to obtain parameters required for setting up connection with the Surrogate Host and to obtain the Ahoy parameters that are not distributed by the Ahoy protocol. These Ahoy parameters are the hash function, keep-alive timer and keep-alive jitter. The initial values of the Ahoy depth and Bloom filter parameters are also taken from the file. Parameters required for the surrogate connection are the IP address of the Surrogate Host, the URL where the JAR files of the Surrogate are published, the keep-alive parameters needed to maintain the Surrogate Host connection and some other optional ones.
- It sends an Ahoy update request on the loopback network interface, port 5000 to the CDA Daemon, expecting the daemon to respond by sending its latest announcement.
- It registers its Surrogate with the Surrogate Host. The Surrogate Host activates the Surrogate and registers it with the JINI Lookup Service.
- It sends a one-way HTTPInterconnect message to the Surrogate containing the latest Ahoy announcement and the non-distributable Ahoy parameters.
- It starts a thread to listen to the periodic Ahoy announcements coming from the CDA Daemon on the loopback interface at port 5555. It starts a second thread that listens to the incoming Ahoy client queries, this too on the loopback interface, port 5556.

- It enters idle state.

Idle State

The following events can occur to the CDA Device Service while in idle state:

- Ahoy announcement from CDA Daemon (Section 4.1.1.1)
- MSP announcement from CDA Surrogate (Section 4.1.1.2)
- Query for Ahoy context source from CDA Surrogate (Section 4.1.1.3)
- Query for MSP context source from CDA Daemon (Section 4.1.1.5)
- Response to MSP client query from CDA Daemon (Section 4.1.1.4)
- Response to Ahoy client query from CDA Surrogate (Section 4.1.1.6)
- Keep-alive timer expires (Section 4.1.1.7)

4.1.1.1 Ahoy announcement from CDA Daemon

The Ahoy announcement is received from the CDA Daemon on port 5555 on the loopback interface. Upon receiving the announcement, the CDA DS does the following: It extracts the width, depth and ABF values from the announcement and updates with those values its local variables that have the same name. It creates a one-way HTTP Interconnect Ahoy announcement message containing the received values and sends the message to the CDA Surrogate. Section 4.2.2.1 describes the message format in detail.

4.1.1.2 MSP announcement from CDA Surrogate

The MSP announcement is a one-way HTTP Interconnect message originating from the CDA Surrogate. When the CDA DS receives the MSP announcement, it takes the following actions: It extracts from the received message the ABF that contains the filters of the MSP service names, it creates a regular Ahoy announcement message in which it includes the received ABF and sends it to the loopback IPv6 interface to port 5000, where the CDA Daemon is listening.

4.1.1.3 Query for Ahoy context source from CDA Surrogate

The query for Ahoy service reaches the CDA DS as the request segment of a two-way HTTP Interconnect message originating from the CDA Surrogate. Upon receiving the query, the DS performs the following actions: It extracts the string parameter from the received message, creates a regular Ahoy query message with the extracted string as service name and sends the query message to the loopback IPv6 interface to port 5000, where the CDA Daemon is listening. The thread that sends the query message is blocked until it receives a response event or until socket timeout occurs.

4.1.1.4 Response to MSP client query from CDA Daemon

The response message from CDA Daemon is received on a random port on the loopback interface, the port being the same where initially the DS sent the query from. Upon receiving the response, the following actions happen: The CDA DS sends the entire byte array to the CDA Surrogate in the response segment of the initially received two-way HTTP Interconnect query message. The format of the HTTP Interconnect response message is described in section 4.2.2.6.

4.1.1.5 Query for MSP context source by CDA Daemon

The DS receives the query message on port 5556 on the loopback IPv6 interface. It performs the following actions: It extracts from the message the query ID and the query string (i.e. queried service name). It saves the query ID as a local variable since it will be needed for the query response. Then, the DS creates a two-way HTTP Interconnect message and sends the query string in the request segment of that message expecting a response to it. The format of the HTTP Interconnect query message is described in section 4.2.2.3.

4.1.1.6 Response to Ahoy client query from CDA Surrogate

The response message reaches the CDA DS as the response segment of a two-way HTTP Interconnect message which the DS itself initiated. Upon receiving the message, the DS reads from local variables the query ID belonging to that response and the IP address and port number of the Ahoy node that initially sent the query, the so-called destination address and destination port. The DS creates two Ahoy response messages, both of them reusing the same query ID and the service address (byte array) received in the HTTP Interconnect response message. One of the Ahoy response messages gets sent to the loopback IPv6 interface and the stored destination

port. The other Ahoy response message gets sent to the stored destination address and port. The significance of sending two response messages is that the CDA DS does not know whether the initial Ahoy query came from the CDA Daemon or another Ahoy node. If it sends the response to the stored destination address, the response gets discarded when the destination address is the public IP of the CDA Daemon. To prevent that from happening the response message is sent twice and once to the loopback interface. The CDA Daemon only receives the response once, since a second response to the same query ID is always discarded.

4.1.1.7 Keep-alive timer expires

The initial values of keep-alive timeout and jitter are obtained during startup from a configuration file. When the timer expires, the DS sends a new keep-alive message to the CDA Daemon and resets its keep-alive timer to the value: *keepalive-time +/- (keepalive-time*keepalive-jitter/100)*, conform the Ahoy specification. The format of the keep-alive message is the same as in Ahoy.

4.1.2 CDA Surrogate

The events of the CDA Surrogate are either default NMS events or they are events specific to the CDA protocol. We only consider those that are specific to the CDA protocol.

- Ahoy announcement from CDA Device Service (Section 4.1.2.1)
- Query for MSP context source from CDA Device Service (Section 4.1.2.2)

4.1.2.1 Ahoy announcement from CDA Device Service

The Ahoy announcement reaches the Surrogate as a one-way HTTP Interconnect message sent by the CDA Device Service. Upon receiving the message, the Surrogate does the following: It extracts from the message the Ahoy hash, depth, width and ABF values and calls a method on the CDA Service Proxy passing the extracted values and its own service ID as arguments. The Surrogate's service ID is required so that the Broker Service can identify uniquely the cached CDA Service proxies.

4.1.2.2 Query for MSP context source from CDA Device Service

The query message for MSP context source reaches the CDA Surrogate as the request segment of a two-way HTTP Interconnect message sent by the CDA Device Service. In response to the event the Surrogate extracts the query string from the message and calls a method on the CDA Service Proxy with the query string as argument. It expects the method to return a byte array containing the address of the queried MSP service. It reads the return value and sends the byte array to the Device Service in the response segment of the two-way HTTP Interconnect message which initiated this event.

4.1.3 CDA Service Proxy

The CDA Service Proxy is instantiated by the CDA Surrogate. It is also the Surrogate that registers the Service Proxy with the JINI Lookup Service.

- Ahoy announcement from CDA Surrogate (Section 4.1.3.1)
- MSP announcement from CDA Broker (Section 4.1.3.2)
- Query for Ahoy service from CDA Broker (Section 4.1.3.3)
- Query for MSP service from CDA Surrogate (Section 4.1.3.4)

4.1.3.1 Ahoy announcement from CDA Surrogate

By Ahoy announcement we mean the local method call on the CDA Service Proxy initiated by the Surrogate. When the method call is executed the Service Proxy checks whether the callback interface of the CDA Broker Service is not null and calls a remote method on it passing all the arguments it received from the Surrogate: The ABF array, depth, width, hash parameters and the Surrogate's service ID.

4.1.3.2 MSP announcement from CDA Broker

By msp announcement we mean a remote method call initiated by the CDA Broker. The Service Proxy does the following when the call is executed: It creates a one-way HTTP Interconnect message which takes as parameter the ABF byte array that it received as argument and sends the message to the CDA Device Service. The ABF contains the Bloom filters of the MSP service identifiers that the Broker Service knows about. Section 4.2.2.2 presents the HTTP message format.

4.1.3.3 Query for Ahoy service from CDA Broker

The query event for an Ahoy context source is a remote method execution initiated by the CDA Broker. The method has as argument a string value which encodes the name of the queried Ahoy service. The Service Proxy reacts to the event by creating a two-way HTTP Interconnect message and encoding the received query string in its request segment. It then sends the two-way message to the CDA Device Service awaiting the response segment to arrive. When the CDA Service Proxy reads the response segment, it decodes the byte array and returns it immediately to the Broker Service.

4.1.3.4 Query for MSP service from CDA Surrogate

The query event for MSP context source is initiated by a local method call that the CDA Surrogate executes. Upon call execution the CDA Service Proxy calls a remote method on the callback interface of the CDA Broker passing the received query string as argument. It expects a byte array as return value. When the Service Proxy reads the return value it immediately returns it to the CDA Surrogate.

4.1.4 Broker Service

Startup

The CDA Broker Service is modeled after a regular JINI service. It therefore consists of a server and a proxy object. We do not distinguish the events of these two components, they get discussed as being the events of the Broker Service.

Startup

The following events are performed by the Broker Service at startup:

- It initializes two hash maps to store the CDA and MSP service lists and initializes a third hash map for the JINI lookup services.
- It discovers lookup services. In function of the provided command line arguments either unicast or multicast discovery is performed. If a valid IP address is provided, the Lookup Service located at the given address is discovered. Otherwise, the Broker Service registers as discovery listener and waits for lookup services to announce themselves.
- Register as service listener to the CDA and MSP context sources.

Idle State

In idle state the following events can occur in the Broker Service:

- Listener event from Lookup Service (Section 4.1.4.1)
- Listener event from CDA or MSP service (Section 4.1.4.2)
- Query request for Ahoy context source from MSP client (Section 4.1.4.3)
- Query request for MSP context source from CDA Service Proxy (Section 4.1.4.4)
- Callback event: Ahoy service update (Section 4.1.4.5)

4.1.4.1 Lookup Service listener event

This event happens when a new JINI Lookup Service appears or an existing one gets removed. For the event to happen the CDA Broker must have initially used multicast LS discovery. When the even occurs, the Broker Service adds the found Lookup Dervice to its local cache of lookup services and registers its Service Proxy and service attributes with the newly found LS.

4.1.4.2 CDA or MSP service listener event

This event occurs when new CDA or MSP context sources get registered with the lookup services or when existing ones get removed or modified. Upon receiving this event the CDA Broker does the following: It checks whether the received event is a CDA or MSP service event.

If the event is an MSP service event the CDA Broker performs an appropriate action with the service (e.g. adds to, removes from or updates the MSP service cache) and then it sends a new MSP announcement to the CDA Service Proxy. When no MSP services are left in the cache an empty ABF array is sent. That notifies Ahoy that all MSP services have been revoked.

If the event is a CDA service event, the Broker performs other actions. If the CDA service is a new one gets added to the service cache and the Broker Service registers itself as a callback interface with it. This is required so that the Broker Service can receive ABF updates from the CDA services and so that the CDA Service Proxy can relay to the Broker incoming Ahoy client queries.

4.1.4.3 Query request for Ahoy service by MSP client

This event is initiated by the MSP client in form of a remote method call which has the query string as argument. The Broker does the following as response to the event: It iterates through all CDA services it has stored in its local CDA service list, for each service it reads their Ahoy parameters (e.g. depth, width, hash function) and stored ABF value and using those values computes the Bloom filter of the received MSP query string. For each found CDA service it checks whether the Bloom filter of the query string matches with the stored ABF or not. If a match is found the Broker calls a remote method on the respective CDA Service Proxy expecting a byte array as return value. The byte array contains the IP address, port number, address type and address size of the queried Ahoy service conform the Ahoy specification. The CDA Broker reads the return value and passes it on to the MSP client.

4.1.4.4 Query request for MSP service by CDA Service Proxy

This event occurs when the CDA Broker receives a remote method call from the CDA Service Proxy querying for the address details of an MSP service. The method has a query string as argument. When the CDA Broker detects the event it iterates through the MSP services stored in its MSP cache and for each service it checks whether their service name equals the provided query string. If a match is found, the Broker calls a remote method on the MSP service requesting its full address. The returned byte array is passed on immediately to the CDA Service Proxy which called the remote method.

4.1.4.5 Callback event: Ahoy service update

This event occurs when the CDA Service Proxy receives an Ahoy announcement from the Device Service and calls a remote method on the Broker Service callback interface. The Broker deals with the event the following way: Based on the received service ID it searches its local CDA service cache for a match and it replaces the stored service attributes of the found key. This event was implemented as a work-around, because we did not succeed to implement the user-defined service entries which would have offered a more generic solution.

4.1.5 CDA Daemon

As stated earlier, the CDA Daemon is different from regular Ahoy daemons, since it has enhanced functionality. We do not consider the regular Ahoy

events only the ones that are CDA-specific.

The CDA-specific daemon events are the following:

- Latest announcement changed (Section 4.1.5.1)
- Query for MSP context source by Ahoy client (Section 4.1.5.2)

4.1.5.1 Latest Announcement Changed

Whenever the latest announcement in the CDA Daemon is different from the previously stored latest announcement the standard Ahoy event flow is that the daemon broadcasts the latest announcement to its neighbors. The CDA protocol requires however that the CDA Device Service also gets notified. Hence, the default Ahoy event flow got modified the following way: When the daemon broadcasts the latest announcement to its neighbors it also sends it to port 5555 on the loopback IPv6 interface. The CDA Device Service is required to listen for announcements on that port.

4.1.5.2 Ahoy Client Query

When the CDA Daemon receives a query and does not offer the service the default Ahoy event flow demands that the daemon forwards the query to the default Ahoy port (5000) of all its neighbors that know about the queried service. In case of the MSP services this neighbor is the CDA Device Service and its address is the loopback interface. According to default Ahoy behavior the daemon would have to forward the query to port 5000 on the loopback interface. There, however, the daemon itself listens and not the CDA Device Service. To solve this issue the default Ahoy event flow was modified for the CDA Daemon. The CDA Daemon checks the address of the neighbor before it forwards the query. If the address is the loopback address, it forwards the query to the reserved CDA port with number 5556.

4.2 Message Types

4.2.1 Channel 1: CDA Device Service and CDA Daemon

The messages in this group are UDP messages exchanged by the CDA Device Service and the CDA Daemon across the loopback IPv6 interface of the CDA node. The message formats are conform the Ahoy protocol specification and are described in [4].

4.2.2 Channel 2: HTTP Interconnect Protocol

This group consists of HTTP messages with a format specified by the MSP-Project's Interconnect protocol implementation[8]. The messages are identified by their type and operation code. In CDA two message types are used: One-way and request messages. One-way messages are useful when no reply is expected. Request messages are two-way, consisting of a request and response segment.

4.2.2.1 Interconnect Ahoy Announcement Message

The Interconnect Ahoy Announcement (IAA) message is a one-way HTTP Interconnect message sent by the CDA Device Service to the CDA Surrogate to announce the latest Ahoy service updates. The message contains the hash, depth, width and ABF array extracted from the Ahoy announcement that the DS received from the CDA Daemon. Table 1 shows the message format in detail.

Size (byte)	*Description*	*Java type*
1	opcode	byte
4	hash	int
2	depth	short
4	width	int
depth*width/8	ABF	byte[ ]

Table 1: *Format of Interconnect Ahoy Announcement (IAA) message.*

4.2.2.2 Interconnect MSP Announcement Message

The Interconnect MSP Announcement (IMA) message is a one-way HTTP Interconnect message sent by the CDA Service Proxy to the CDA Device Service to announce the latest MSP service updates. The message consists of the ABF calculated from the MSP service identifiers. Table 2 shows the message format in detail.

Size (byte)	*Description*	*Java type*
depth*width/8	ABF	byte[ ]

Table 2: *Format of Interconnect MSP Announcement (IMA) message.*

4.2.2.3 Interconnect MSP Service Query Message

The Interconnect MSP Service Query (IMQ) message is a two-way HTTP Interconnect message sent by the CDA Device Service to the CDA Surrogate to request the contact details of an MSP service. Table 3 shows the request message format in detail.

Size (byte)	*Description*	*Java type*
service name length	service name	String

Table 3: *Format of Interconnect MSP Service Query (IMQ) message.*

4.2.2.4 Interconnect Ahoy Service Query Message

The Interconnect Ahoy Service Query (IAQ) message is a two-way HTTP Interconnect message sent by the CDA Service Proxy to the CDA Device Service to request the contact details of an Ahoy service. The format of this message is the same as the format of the IMQ message, they even have the same opcode. Only their direction is different.

4.2.2.5 Interconnect MSP Service Response Message

The Interconnect MSP Service Response (IMR) message is the response segment of the two-way IMQ message. The IMR is sent by the CDA Surrogate to the CDA Device Service as reply to the Ahoy client query. It is not a stand-alone message. In CDA, in line with the Ahoy specification, this message consists of 20 bytes in total, comprising the address type and address size, as well as the IP address and port number of the queried service. Table 4 illustrates the format of this response message.

Size (byte)	*Description*	*Java type*	*Default Value*
1	IP address size	byte	20 (IPv6)
1	IP address type	byte	1 (IPv6)
2	port number	byte[ ]	
16	IP address	byte[ ]	

Table 4: *Format of the Interconnect MSP Service Response (IMR) message.*

4.2.2.6 Interconnect Ahoy Service Response Message

The Interconnect Ahoy Service Response (IAR) message is the response segment of the two-way IAQ message. The IAR is sent by the CDA Device Service to the CDA Service Proxy as reply to the MSP client query. The format of this message is the same as the format of the IMR.

4.2.3 Channel 3: Java RMI in MSP

Java RMI is used as communication channel in MSP, hence Java RMI is to some extent also integral part of the CDA protocol. In the context of Java RMI we cannot speak of message types, since it is not a message passing technology, it uses remote method calls instead. The service interface definitions of the respective CDA components that use Java RMI technology is included in the documentation of the project.

4.2.4 Channel 4: Client-Server Communication

Initially we designed the CDA protocol to use XML-RPC for cross-framework client-server communication and context exchange. In the end the XML-RPC client and server modules have not been integrated with CDA, therefore we do not discuss them.

5 Implementation Report

5.1 Implementation Steps

This section gives a chronological account of the implementation of the CDA protocol, mentioning turning points and challenges encountered through the implementation.

The implementation followed a bottom-up approach. We created a basic JINI client with simple GUI and a skeleton CDA Service interface modeled after a regular JINI service to test some standard JINI functions with them. Such as service registration with the LS and service discovery. In this phase the MSP back-end was not being used. A Lookup Service was running on the local machine.

In a second phase we began to interface the CDA JINI service with Ahoy. Scrutiny of the Ahoy protocol revealed that the best way to interface the Ahoy daemon and the CDA Service was through the loopback network interface, using UDP sockets. First the Ahoy update and response messages were implemented, enabling the CDA Service to obtain an initial announcement from the Ahoy daemon. We found out that socket timeout was necessary for the update request message, or else the CDA would pend waiting. Later socket timeouts proved necessary in more than one place.

A second interfacing with Ahoy was to make the CDA Service listen to periodic announcements from the daemon. The daemon sends the announcements to the IPv6 multi-cast network interface, an artificial construct on which the CDA Service could not listen to messages. Neither could it listen for announcements on the default Ahoy port 5000, since that port was taken by the daemon. To solve this we made port 5555 the mandatory port for the CDA Service to listen on. At that point the Ahoy daemon required a modification: Before it broadcasts its latest announcement it forcibly sends the announcement to port 5555 on the loopback IPv6 network interface, where the CDA Service receives it.

The message passing between the daemon and the CDA Service required that we program various bit and byte operations in Java, which were placed in a utility package. At that stage we implemented and tested extensively our Java implementation for Bloom filters. The dummy JINI client was made to query the CDA Service for Ahoy services, for which the Bloom filter of the query string had to be computed. The CDA Service needed to send Ahoy query messages and read the response to it. The query-response Ahoy interaction caused no surprise.

Next we expanded the JINI components and created the Broker service,

making the JINI client query the Broker service instead of the CDA Service. The Broker was made to store the CDA Service reference in a hash map, which was to become the future local cache for CDA services. Around that time we implemented two new JINI services which were serving as MSP services to be announced to Ahoy. The computation of ABF in the Broker had to be throughly tested. The Broker started using a second hash map for the MSP services. The CDA Service was made to send announcement messages to the daemon. The end result was that the MSP services could be successfully announced to Ahoy. The daemon automatically registered the CDA Service as its neighbor, logging the loopback network interface address as the neighbor's reference.

New challenges appeared when we tried to run Ahoy queries for the announced MSP services. We found that the daemon would forward the query to itself by sending it to the loopback interface port 5000, as demanded by the Ahoy specification. To make the query reach the CDA Service, another modification of the daemon interface was necessary. It got hard coded in the daemon that whenever it would forward a query to the loopback interface, it would use port 5556 and the CDA Service would listen there expecting query messages. We faced another difficulty when the response to the Ahoy query was sent to the daemon. We realized that the CDA Service had no way to know whether the initial Ahoy query came from the local node (a local client query) or from another Ahoy node. The CDA Service was implemented such way that it returned the response message to port 5000 to the address of the query sender. When this address was the public IP of another node, the node would receive the response fine. When, however, the address was the public IP of the local daemon, the query response would be discarded, since the daemon considered the query inactive. We provided a work-around for this by requiring the CDA Service to send all response messages to both the loopback network interface and to the IP address of the sender. This way the daemon always receives the response through the loopback interface, but can read it only when itself initiated the query. In other cases the response is discarded.

Until that point we have not used the MSP backend. A new phase began when we transformed the CDA Service from a simple JINI service into an NMS-like service. The conversion was painful and took up much time, because various configuration issues (e.g. Linux hosts file entries, NMS-bound keep-alive timers), broken module dependencies, RMI security and code base issues as well as clashing module version problems prevented the CDA Device Service from registering and keeping alive its Surrogate for more than a few seconds. Once all issues got solved and the Surrogate could stay alive for stable time, thorough testing took place to make sure that all previous functionality of the JINI-like CDA Service was unaltered in the NMS-like

CDA Service.

When the CDA Service proved to function well as NMS, we pursued to add service listener properties to the Broker Service. The goal was to have the Broker Service automatically receive updates about CDA and MSP services. The implementation would not work without any indication about the reason. After experimenting with the alternative of the Broker service polling regularly the LS for service information, the root of the problem was found: It was a missing RMI code base that was supposed to point to an external JINI module that contained the Java classes required to activate the service listener functionality. That phase of development ended with success when the Broker successfully started to receive the service updates. One observed anomaly was that the Broker would never receive actual update events, only events about adding and removing CDA and MSP services. The answer for this is in the JINI specification. Another issue, however, might explain the lack of updates. That issue is what we discuss next.

The initial design of the CDA protocol assumed that the ABF array and other Ahoy parameters could be stored as service attributes of the CDA Service in the JINI Lookup Service. The reason for this was primarily that JINI offers automatic updating of these service attributes. We envisioned that every time the ABF of a CDA Service changes, the JINI would receive the automatic update and by being a service listener the Broker Service would receive the same update in almost real-time. To be able to store the ABF and other Ahoy parameters as service attributes user-defined Java *Entry* objects need to be implemented that comply with the JINI specification. This we have performed, yet the Lookup Service would ignore them. Scrutiny revealed that the Lookup Service requires a code base parameter which points to a JAR package containing the user-defined Entry classes. We attempted to modify the code base parameter of the Lookup Service running on the virtual MSP back-end, yet the service attributes were still ignored. Another attempt was to use a different Lookup Service, on the local machine. We modified its code base and it accepted the new Java Entry types, yet the solution caused problems with the registration and discovery of the CDA Service, which relied heavily on the MSP back-end. The solution adopted in the end was to not use the user-defined entries. Instead, we made the Broker Service register itself as callback interface to the CDA Service. This way the updates are sent directly to the Broker Service and the ABF is not registered in the LS. Solving this issue is a challenge for future work.

A few more challenges were addressed toward the end of the implementation: The CDA Service was made to send keep-alive messages to the daemon to prevent the MSP services from being unregistered prematurely. Various hard coded parameters (IP addresses, port numbers, Ahoy parameters, etc.) were made configurable to facilitate the final performance testing of CDA.

Performance testing was first performed in the implementation environment, which implied a rather artificial setup: CDA Device Service, Broker Service, MSP and Ahoy client, as well as MSP services were all running on the same host. Only the CDA Surrogate was running on the virtual host. We have modified this by creating a model of the MSP framework on the virtual host and a model of the CDA node on the local machine. In the new setup the CDA Surrogate, Broker Service, MSP client and MSP services run on the virtual host, while the CDA Device Service, CDA Daemon and Ahoy clients run on the local host. The latter is a more realistic setup, which is why we only report those test measurements.

5.2 Implementation Tools

As seen in the previous section, the implementation environment consists first of one machine, known as the local host or host platform. Initially, a Lookup Service was running on the local machine, but in later phases the Lookup Service only runs on the MSP back-end. The MSP back-end is referred to as the virtual host. The full-blown implementation environment consists of the following:

- Linux (Ubuntu 2.6.9.12) host platform.
- Eclipse[26] SDK version 3.1 for Linux on the host platform. Eclipse is an advanced development kit with powerful Java/Ant plug-ins. Ant was used as Java building tool instead of the default Eclipse Java Builder.
- VMWARE Player[25] hardware simulation engine running on the host machine.
- MSP back-end[8] virtual application running with the help of the VMWARE engine. The two of them together are referred to as the virtual machine or virtual host. The MSP back-end is an Ubuntu-based virtual image with JINI 1.2 environment installed and with the following MSP components running: JINI Lookup Service, Surrogate Host, web server and RMI daemon.
- Various MSP, Ahoy, JINI and XML-RPC libraries. The packages are described in detail in the *Installation instructions* provided with the documentation of the project. Environment settings, configuration parameters and package dependencies are also documented.
- Apache2 and JINI default web servers and an RMI activation daemon.

5.3 Test Report

5.3.1 Test Scenarios

Tests have been carried out throughout the CDA protocol implementation. With the help of tests we were able to catch design, programming and configuration errors. Most tests have been run in the implementation environment, which initially consisted of a host machine only and later got extended with a virtual machine.

The final performance tests have been performed in the following test setup:

MSP back-end: Broker Service, Lookup Service, MSP client, MSP services, CDA Surrogate and Surrogate Host run on the virtual host, also known as the MSP back-end. These components form a symbolic MSP framework.

Host machine: CDA Device Service, CDA Daemon and Ahoy clients run on the host machine forming a symbolic CDA node.

The advantage of the chosen test scenario is that it creates a virtual separation of the MSP components from the components meant to run on the CDA mobile node. The web server that hosts the CDA Surrogate and other essential JAR files is running on the host machine, though it could just as well be running on a remote machine.

Each test run consists of the following steps: 1) Announcing 10 different Ahoy services; 2) Announcing 10 different MSP services; 3) Querying 10 times for 10 different Ahoy services; 4) Querying 10 times for 10 different MSP services.

The order of the four steps was varied arbitrarily, with step 3 always following step 1 and step 4 always following step 2. Otherwise, the queries were not successful. In total ten such test runs were carried out during the interval of one day. After every test run the results were collected from different log files and merged into a data spreadsheet where further processing was performed with them.

5.3.2 Test Measurements

The following values have been measured:

- The time it takes to register Ahoy and MSP context sources with the MSP and Ahoy frameworks, respectively.
- The time it takes an Ahoy and MSP client to query for MSP and Ahoy context sources, respectively.
- The size of extra modules required on a CDA node compared to the size of modules required by a regular Ahoy node. This value is not time-dependent, it has been calculated independently.

To be able to collect the test data a few modifications were required in the CDA Daemon, the Broker Service and in the generic MSP service.

To measure the announcement time of MSP services the MSP service logs the system time stamp when it receives a service ID from the JINI LS and the CDA Daemon logs the time stamp and sender's ID when it receives a network announcement. The announcement time of MSP services is the time elapsed between the two.

To measure the announcement time of Ahoy services the CDA Daemon logs the system time stamp upon receiving a local client announcement and the Broker Service logs the time stamp when it receives a service update from the CDA Surrogate. The time elapsed between the two gives the value we are interested in.

To measure the duration of queries for MSP services the CDA Daemon logs the time stamp upon receiving a local client query and it logs the time stamp when it receives a query response from the network. The elapsed time between the two time stamps gives the query time. We only measure the time of queries initiated by local clients on the CDA node, since non-local Ahoy queries follow the same path within the CDA Daemon. If their queries take longer it is due to Ahoy.

To measure the duration of query for Ahoy services the Broker Service logs the time stamp when it sends the query string to the selected CDA Service Proxy and it logs the timestamp when it receives the response to that same query. The elapsed time between the two time stamps gives the value we are looking for.

To get an estimate of the extra load on the CDA node compared to regular Ahoy nodes we added up the sizes of modules required for the CDA Service to run. These modules include the JAR files required for the NMS functionality, as well as the libraries needed for the XML-RPC client and server.

5.3.3 Test Results

Announcement time of Ahoy and MSP context sources in CDA

Figure 17 shows the mean announcement times of Ahoy and MSP context sources obtained for ten discrete test runs. The relative position of the graphs shows that the announcement of MSP context sources takes on average longer than the announcement of Ahoy context sources.

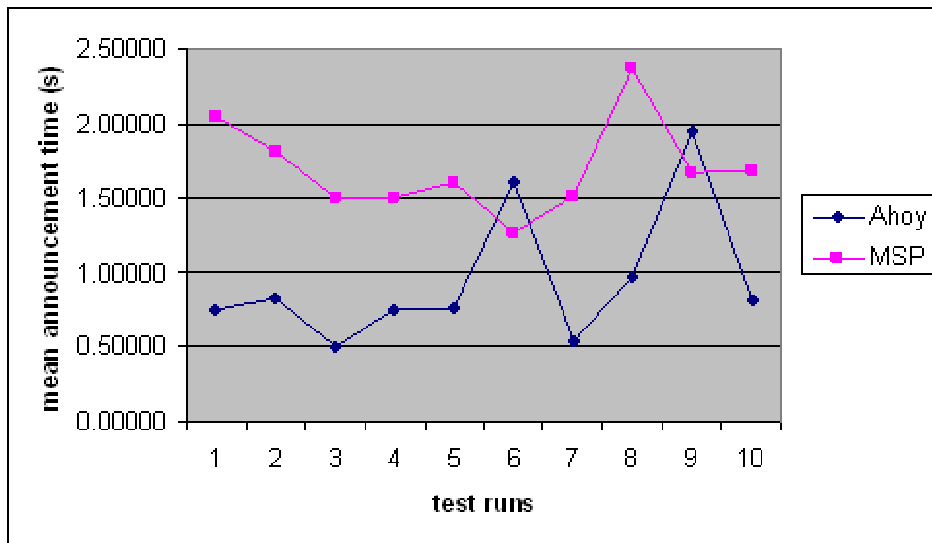

Figure 17: *Measured mean announcement times of Ahoy and MSP context sources obtained from ten discrete test runs. The X axis displays the number of test runs and the Y axis displays the measured mean values (seconds) per test run and type of context source.*

Figure 18 illustrates the mean announcement time computed from the same data set only this time averaged across all of the then test runs. We see from this second figure that the obtained mean value for Ahoy announcements is 0.94612 seconds (946.12 milliseconds), while for MSP announcements the mean value is 1.69767 seconds (1697.67 milliseconds). The data confirms what we already saw in the first figure, namely that announcing MSP context sources in CDA takes on average longer than announcing Ahoy context sources. The standard deviation of the Ahoy announcement values is very large, with a value of 0.93633 seconds, which amounts to the same value as the computed mean announcement time. This suggests that we cannot rely much on the results obtained for Ahoy service announcement. The standard deviation of the MSP service announcement values is within normal range, with a value of 0.85498 seconds compared to the mean value of 1.69767

seconds.

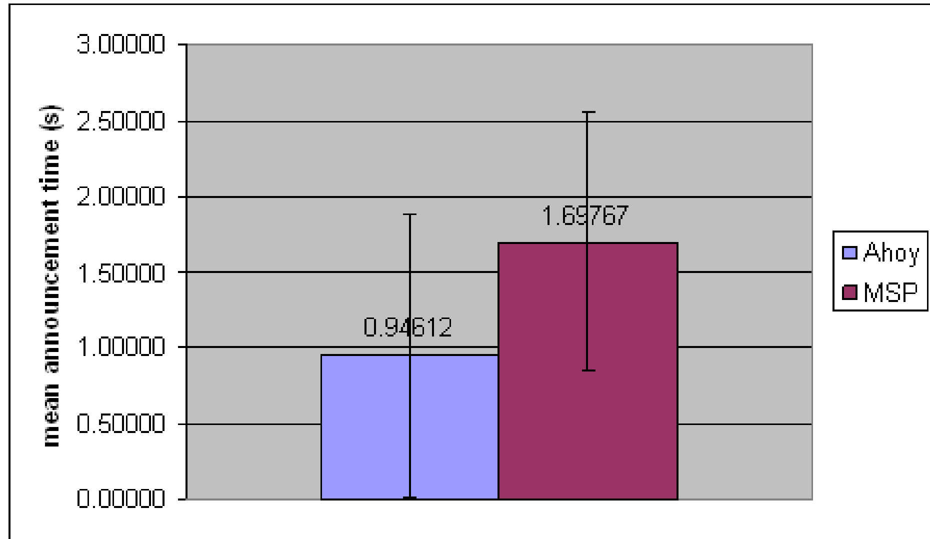

Figure 18: *Measured mean announcement time of Ahoy and MSP context sources obtained from accummulated data of ten test runs. The X axis illustrates the two types of context sources and the Y axis gives the mean announcement time (seconds) per type of context source.*

Among the announcement times of Ahoy context sources we repeatedly found negative values. The likely explanation is that the announcement took such a short time (say ten or hundred milliseconds) that it was shorter than the clock difference between the local and virtual machines on which the tests have been run. The two clock were however synchronized by the same network time server, which is why the outcome is surprising. Another explanation might be that logging performed by the CDA Daemon introduced some time anomaly. If that is the case, the time anomaly affected all test results, so it must be the very short announcement time of Ahoy services which made the negative values appear in this particular data set.

In spite of the measurement errors the results clearly suggest that the announcement of Ahoy context sources takes less than the announcement of MSP context sources. We believe the main reason for the difference is that announcing MSP services requires that the Broker Service calculates a new ABF every time. While, when announcing Ahoy services the received ABF is transferred unaltered from the CDA Daemon all the way to the Broker Service. The results suggest that Java-based ABF computation introduces a delay factor. To obtain conclusive results, more tests should be performed.

Query time of Ahoy and MSP services in CDA

Figure 19 shows the mean discovery times of Ahoy and MSP context sources obtained from ten discrete test runs. The relative position of the graphs suggests that discovering MSP services takes less time than discovering Ahoy services.

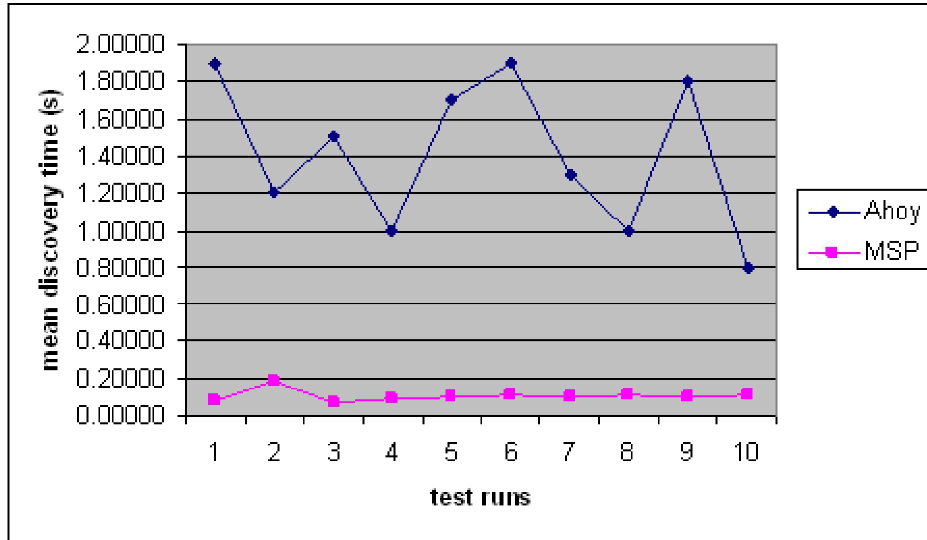

Figure 19: *Measured mean discovery times of Ahoy and MSP context sources obtained from ten discrete test runs. The X axis shows the number of test runs and the Y axis gives the mean discovery times (seconds) per test run and type of context source.*

Figure 20 shows the mean discovery time obtained from the same data set only this time computed from the accummulated data across ten test runs.

The standard deviation for both Ahoy and MSP discovery falls within normal range. The results suggest that the discovery of MSP context sources takes measurably less than the discovery of Ahoy context sources. It is difficult to explain these findings. The only obvious difference between the two discovery paths is that for MSP service query no Bloom filter computations are required contrary to query for Ahoy services. During MSP service discovery the Broker Service matches the received query string against the cached MSP service names, while during Ahoy service discovery the Broker computes the Bloom filter of the received query string and matches it against the ABF array of available CDA services. The results suggest that the Java-based bit array computations involved in the latter process introduce delay. To be conclusive about these findings more accurate tests should be performed.

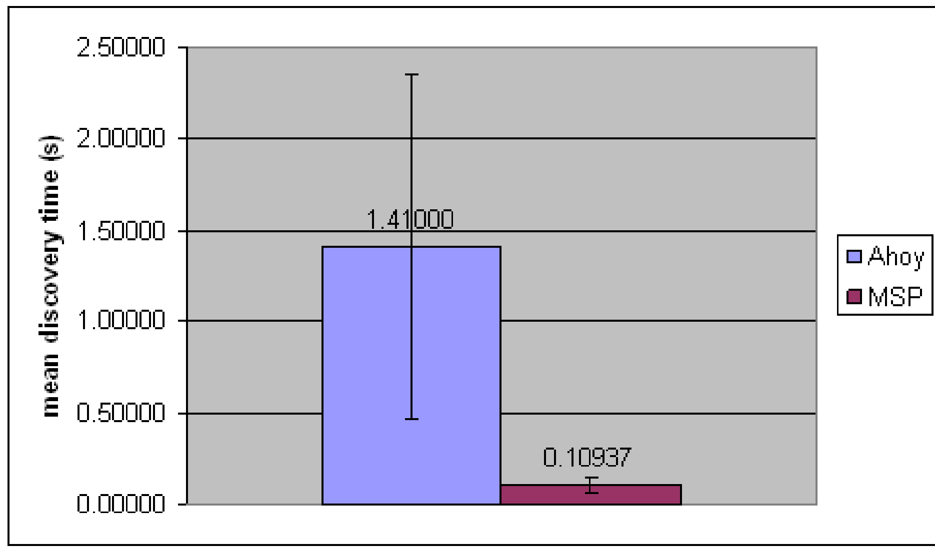

Figure 20: *Measured mean discovery time of Ahoy and MSP services obtained from accummulated data of ten test runs. The X axis shows the context source types and the Y axis gives the mean discovery time (seconds) per context type.*

CDA-specific load

The various JINI and MSP modules required by the current implementation of CDA Service to be able to run amount to a total of 3398.5 KB volume. The volume of the CDA Service Java byte code is about 70 KB. In total, the NMS-specific modules and the CDA Service itself require about 3500 KB, about 3.4 - 3.5 MB.

If we wanted the CDA node to be capable of offering services or to act as client in CDA context exchange, the XML-RPC client and server modules are necessary. These add another 800 KB to the total volume. A fully fledged CDA node would require about 4-5 MB hard disk space.

With new generation PDAs on the market that have a few GB hard disk size[27], the 5 MB demand of the CDA modules is definitely feasible. Besides the crude volume estimate it would be also interesting to know the RAM requirements of the CDA Service. For those measurements real-life tests should be carried out with the CDA Service installed on mobile device. Such advanced test setups go beyond the scope of the current project.

6 Conclusions

The successful implementation of CDA shows that interlinking between Ahoy and MSP is, indeed, feasible. With only few changes applied to Ahoy and MSP the two frameworks are now capable to share context information with each other. The protocols that announce and discover cross-framework context sources have been implemented and tested. The test results have shown that context sources get registered and discovered within realistic time limits across CDA, regardless whether they originate from Ahoy or MSP. The mechanism how CDA context sources can distribute context information to clients has been designed and implemented as a stand-alone module, yet due to time constraints it did not get integrated with CDA. The choice of design for the context exchange protocol has changed over the time and that has played a role in the delay. Our first choice was a Java RMI-based solution which, given the JINI-based architecture of MSP, would have integrated transparently with MSP. Standard Java RMI implementations, however, are not suitable for applications running in mobile networks, like the Ahoy component of CDA. The search for lightweight Java RMI implementations has revealed two promising projects, yet their implementations were unavailable. That has led us to consider the XML-RPC based solution. There we encountered difficulties of a different kind. The lightweight web server that comes with the XML-RPC modules displayed anomalies during tests performed with IPv6 addresses and remote CDA clients, while it would run correctly when a local client was used. We suspected Java security issues to cause the problem, yet policy changes would not help. With more suitable XML-RPC documentation we might have overcome the hindrance, but in lack of that we had no other choice than to leave out the XML-RPC integration.

We have met numerous other challenges during the design and implementation process. Most of them we have documented in other sections of this report. We now propose to view the CDA protocol from a broader perspective and consider to what extent its properties fulfill the research requirements we formulated at the start of this project.

An important requirement was the *minimal impact principle*, namely to preserve the characteristics of Ahoy and MSP as much as possible. We believe that the CDA has been designed accordingly. Every decision we took had to comply with this requirement. The difficulty lay in deciding which properties and components were suitable to change and which not, especially, when not modifying a property had negative consequences for CDA. An illustration for that is the JINI Lookup Service. It being a core component of JINI and MSP it was undesirable to modify it. This has resulted in a non-transparent discovery path for MSP clients. To discover

Ahoy context sources the MSP client cannot query the Lookup Service in one step as the JINI protocol demands it, rather it has to query in an indirect way through the Broker Service. It would be a more transparent design if the Lookup Service was able to make an extra call while answering an MSP client query and be capable to perform ABF computations.

The above example is also relevant for the transparency requirement. Accordingly, the CDA design is expected to support transparent (or seamless) communication. By transparency we mean that CDA components hosted either in Ahoy or MSP need not distinguish between entities of their *own* framework versus entities of the *other* framework while they participate in CDA context announcements and discoveries. Most CDA mechanisms successfully comply with this requirement, exceptions from it are MSP client queries and the designed XML-RPC solution for context exchange.

Further requirements demand that the CDA protocol architecture should be robust, modular and extensible. With respect to robustness the CDA protocol grades well. The SOA-like design gives room for redundancy, allowing multiple instances of the same service to be active, providing some degree of fault tolerance. Multi-threaded processing and race conditions could, however, reduce robustness. We tried to minimize the usage of such constructs by introducing asynchronous communication wherever possible.

With regard to modularity and reconfigurability we would not grade the CDA high. We remain on the whole skeptical about the reconfigurability and scalability of the CDA protocol. The numerous configuration parameters used by Ahoy and MSP, the various Java *classpath* and *codebase* dependencies add up to a long list that requires thorough maintenance and continuous human attendance. These issues are an impediment to reconfigurability and scalability.

Since the CDA protocol was intended as a proof of concept in general not too much attention has been given to testing. It is our belief that to obtain reliable test results the CDA protocol should be tested in real-life environment with a multi-hop Ahoy network. Extensive testing has never been the goal of this project.

Another bottleneck that can hinder the practical usability of CDA are its platform requirements. The CDA nodes must run on Unix-like operating systems since the current Ahoy implementation supports Unix platform only. We found that for the marketed hand-held devices very limited Unix support is available. Even if we were to find a Unix-based mobile device, configuring it with the required MSP and Ahoy modules is a considerable effort. Until these limitations exist, the CDA protocol remains peripheral to the consumer market and benefits only the research community.

6.1 Contribution

This project brings in significant contribution to research on context-aware computing. It is a pioneering solution which succeeded to interlink context-sensitive wired and wireless networks at the application level. CDA offers a light-weight bridge-like framework capable to interlink SOA-like wired frameworks like JINI and MSP with Ahoy-based mobile ad-hoc networks.

We believe that CDA offers a promising paradigm for research on pervasive context-aware applications and it also offers a platform for further research.

6.2 Future Work

It would be a major benefit for CDA if it gets extended with a context exchange protocol. The current design supports only context announcement and discovery, which limits the full potential of CDA. The XML-RPC modules developed in this project are one alternative that can be made operational. We believe, however, that an even better solution is lightweight Java RMI. Lightweight Java RMI would integrate transparently with the current Java RMI-based context discovery mechanism of MSP and would provide a flexible solution. For RMI support the Ahoy protocol requires significant extensions, since it would be required to transfer either the full URL of the RMI proxy where it can be downloaded from or, it would have to transfer the serialized proxy objects.

The current CDA implementation had to use a work-around for CDA and MSP service updates, because the integration of user-defined service attributes with the Lookup Service was not possible. The current implementation of CDA has a few remote calls which would not be required if user-defined entries are used. This is something worthwhile investigating since the resulting CDA architecture would be simpler than the one we have now.

Furthermore, it would be a welcome improvement to make the CDA architecture more reconfigurable and scalable. In order to do so, the Ahoy and MSP frameworks would also require changes, since it is necessary to solve the configuration issues of the underlying frameworks first. An example for such a change in MSP is to remove from the Surrogate interface the need for a hard coded IP of the web server where the Surrogate JAR file can be downloaded from. Or, consensus protocols in Ahoy could resolve the current need for initial *width*, *depth* and other configuration parameters.

Another aspect of CDA that can be improved is the classification of MSP services that get announced to Ahoy. As discussed in the report, large numbers of MSP services contribute to higher occurrence of false positives. To

prevent that problem, a classification and filtering mechanism could be designed. The current implementation of CDA provides a starting point for that: It maps MSP context sources to different layers of the ABF announcement in function of their Quality of Context attributes.

The current project has not concentrated much on tests and performance evaluations. The test results presented in this report have been measured in a relatively artificial test setup and hence they are not reliable. An indication of this is the large standard deviation value we obtained for the announcement time of Ahoy services. Large number of diverse (including real-life) tests should be carried out with CDA if we wanted representative performance evaluation. The large number of tests should help reduce measurement errors. Tests should be also performed with a more complex CDA framework, one that has more components than the CDA prototype we used in the current project. The tests would give an insight into the scalability and robustness of CDA.

Implementing support for IPv4 addresses in CDA would be also quite beneficial. The usage of either address should be configurable. Currently not all routers and networks are IPv6 capable. In any such environment the current CDA implementation cannot function, neither can Ahoy for the same matter. Together with IPv4 support solutions should be studied for potential routing risks so that CDA services behind NAT and firewall remain functional.

In CDA no attention has been given to security aspects with regard to context access. In a pervasive environment of context-aware applications consideration should be given to user privacy. In the future CDA could be extended with context security management.

A final, rather challenging future adaptation of CDA would be to integrate with it other frameworks besides the already integrated Ahoy and MSP. This might lead to a radically different CDA design.

7 Glossary

ABF = Attenuated Bloom Filter
CDA = Context Discovery Adapter
DS = Device Service
GUI = Graphical User Interface
JSAS = JINI Surrogate Architecture Specification
LS = Lookup Service
MSP = Mobile Service Platform
PDA = Personal Digital Assistant
SOA = Service Oriented Architecture

References

[1] "ZipCar Project". [Online]. Available at: http://www.zipcar.com/about/

[2] "Freeband AWARENESS". [Online]. Available at: http://awareness.freeband.nl

[3] *Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning*. Aart van Halteren, Pravin Pawar. Department of Computer Science, University of Twente

[4] "Ahoy". [Online]. Available at: http://inglorion.net/software/ahoy/

[5] *Service Discovery Using Bloom Filters*. Patrick Goering, Geert Heijenk. Department of Computer Science, University of Twente

[6] *Context Discovery Using Attenuated Bloom Filters in Ad-hoc Networks*. Fei Liu, Geert Heijenk. Department of Computer Science, University of Twente

[7] "Ruby Programming Language"'. [Online]. Available at: http://www.ruby-lang.org/en/about/

[8] "MSP Tutorial Site". [Online]. Available at: http://janus.cs.utwente.nl:8000/twiki/bin/view/MSP

[9] "JINI". [Online]. Available at: http://java.sun.com/developer/products/jini/index.jsp

[10] "JSAS". [Online]. Available at: http://surrogate.dev.java.net/doc/api/overview-summary.html

[11] "JINI Surrogate Specification". [Online]. Available at: http://www.sun.com/smi/Press/sunflash/2001-05/sunflash.20010530.5.xml

[12] *Enabling Context-Aware Computing for the Nomadic Mobile User: A Service Oriented and Quality Driven Approach*, Pravin Pawar, Aart van Halteren, Kamran Sheikh, Department of Computer Science, University of Twente

[13] "Web Services". [Online]. Available at: http://www.w3.org/2002/ws/

[14] "Web Services IBM Specification". [Online]. Available at: http://www-128.ibm.com/developerworks/webservices/standards/

[15] "CORBA". [Online]. Available at: http://www.omg.org/technology/documents/corba_spec_catalog.htm

[16] "JXTA". [Online]. Available at: https://jxta.dev.java.net/

[17] "JXTA Specification". [Online]. Available at: https://jxta-spec.dev.java.net/

[18] "Egospaces Project". [Online]. Available at: http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/trans/ts/&toc=comp/trans/ts/2006/05/e5toc.xml&DOI=10.1109/TSE.2006.47

[19] "MobiBlog Project". [Online]. Available at: http://www.ee.duke.edu/r̃omit/pubs/micro-blog.pdf

[20] "JINI Service Entry Tutorial". [Online]. Available at: http://jan.newmarch.name/java/jini/tutorial/Entry.xml

[21] *Efficient Support of Java RMI over Heterogeneous Wireless Networks.* Cheng-Wei Chen, Chung-Kai Chen, Jyh-Cheng Chen, Chien-Tan Ko, Jenq-Kuen Lee, Hong-Wei Lin, Wang-Jer Wu. Department of Computer Science, National Tsing Hua University

[22] *Performance Enhancing Proxies for Java 2 RMI over Slow Wireless Links.* Stefano Campadello, Heikki Helin, Oskari Koskimies, Kimmo Raatikainen. Department of Computer Science, University of Helsinki

[23] "XML-RPC Introduction". [Online]. Available at: http://www.ibm.com/developerworks/xml/library/j-xmlrpc.html

[24] "Apache Project XML-RPC". [Online]. Available at: http://ws.apache.org/xmlrpc/

[25] "VMPlayer". [Online]. Available at: http://www.vmware.com/products/player/

[26] "Eclipse". [Online]. Available at: http://www.eclipse.org/

[27] "LifeDrive Palm PDA". [Online]. Available at: http://news.softpedia.com/news/LifeDrive-is-the-first-PDA-with-hard-disk-2078.shtml