

Bridging Context Management Systems in the Ad Hoc and Mobile Environments

Pravin Pawar, Hanga Boros, Fei Liu, Geert Heijenk, Bert-Jan van Beijnum

*Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente, The Netherlands.
{ p.pawar, f.liu, geert.heijenk, beijnum}@utwente.nl
hanga.boros@gmail.com*

Abstract

The pervasive computing world in which the context-aware applications are aimed at is constituted of multiple network environments, e.g. ad hoc, mobile and fixed. There exist specialized Context Management Systems (CMSs) addressing context management needs of every network environment and the existence of a single CMS catering all the network environments does not seem to be plausible. Placing CMS bridges between different types of CMS is a practical approach for exchanging context information in the multiple network environments. Herewith, we focus on a bridge named as Context Discovery Adapter (CDA) which bridges two CMSs namely Mobile Service Platform (targeted towards the mobile network environment) and Ahoy (designed for the ad-hoc network environment), with greatly different properties. CDA enables context discovery and exchange across MSP and Ahoy by means of a mobile device which is capable of participating in both of the CMSs and specialized CDA broker service. The successful working prototype of CDA offers possibility for richer pervasive context-aware applications and a platform for further research.

1. Introduction

In today's pervasive computing world, there is an increasing demand for context-aware applications that span over multiple network environments such as *ad hoc*, *mobile* and *fixed*. For example, in the *flood level monitoring* application, the water level data (context) collected from the wireless sensors covering the endangered area (ad-hoc network) is transmitted using the cellular Internet service (mobile network) to the monitoring center located far away (fixed network), which could take a decision, for instance opening the safety clasps on a dam to guide water flow. In the other scenario of *remote patient monitoring*, when a possible occurrence of seizure (context) is detected, the healthcare professional (fixed network) could instruct

using the cellular Internet service (mobile network) to the patient's mobile device to search the nearby doctors using for example Bluetooth services (ad hoc network). In the context-aware computing model, the necessary components for enabling such decision making are provided by the *Context Management System* (CMS). CMS provides the means to publish and discover context sources and acquire the context information which could be later aggregated and reasoned upon to take a certain decision [1]. In line with the examples mentioned herewith, in the *mobile network environment*, participating mobile devices have connectivity to the Internet, while in the *ad hoc network environment*, the participating mobile devices could communicate with each other, however they don't have any Internet connectivity.

In our previous work [2], we argued that the existing CMSs are very much environment-specific; e.g. targeted towards home environment [3] or large-scale mobile environment [4]. Depending on the characteristics of the target environment and specific functional requirements, the architecture, implementation choices and context formats of these CMSs could be widely different from each other; which make them hardly interoperable. As a consequence, the existence of a single CMS catering all the environments does not seem to be plausible. However, to enable scenarios similar to those outlined in the above paragraph, it is necessary that the CMSs in the different environments should interoperate specially for the seamless discovery and exchange of the context information. As identified in [2], placing bridges between different types of CMS is a practical approach to discover and exchange context information in different network environments. A bridge connecting two CMSs makes possible discovery and exchange of context information between two CMSs so that the applications native to each CMS could make use of the context information provided by other CMS enabling so called context mobility. For a

comprehensive discussion on CMS bridges, we refer to [2].

In this paper, we focus on a CMS bridge referred to as *Context Discovery Adapter* (CDA) which interlinks two fundamentally different CMSs namely *Mobile Service Platform* (MSP) [5] and Ahoy ([6], [7]), with greatly different properties. MSP is a *Service Oriented Architecture* (SOA) based CMS which enables handheld mobile devices capable of connecting to the Internet to take on a role of the context source. On the other hand, Ahoy is designed for the mobile ad-hoc networks and uses the concept of *Attenuated Bloom filters* to publish and discover context sources. Ahoy is targeted towards more resource constrained devices than by MSP. Due to the differences in the target network environments, architecture and implementation of MSP and Ahoy; bridging across them was not a simple task. Nevertheless, the end result has been a successful working prototype of CDA, which offers possibility for richer pervasive context-aware applications and a platform for further research.

The remaining of this paper is organized as follows: Section 2 describes the related work. Section 3 briefly describes MSP and Ahoy and outlines requirements for CDA. Section 4 is on the CDA architecture. Section 5 is on the implementation of CDA. Section 6 describes the test scenarios and CDA performance measurement results. Section 7 concludes the paper and illustrates the future work.

2. Related Work

There exist a number of approaches towards realizing so called *ubiquitous context awareness* [8], which basically aim at providing anytime access to the context information across the heterogeneous and distributed environments irrespective of the origin of the context information. For example, it is argued in [9] that an integration layer which hides the differences between heterogeneous context-aware systems is required to achieve global scalability. However, most of the previously suggested approaches propose an entirely new framework (e.g. [8]) for realizing ubiquitous context-awareness or remain limited to the architectural level (e.g. [9]) without any realistic implementation.

Regarding implementation of context-awareness in mobile networks and handheld devices for context-awareness, we find a few solutions. For example, *EgoSpaces* [10] and *Mobi-Blog* [11]. Though both of them give little attention to the interlinking and concentrate more on the context management aspects. *EgoSpaces* [10] project has developed a middleware framework that delivers context information in an abstract form to the applications running in context-

aware mobile networks. The *EgoSpaces* project also refers to other context-aware middleware solutions for the mobile networks, e.g. LIME, TuCSon and GAIA. *Mobi-Blog* [11] 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.

Comparing with the related work, we distinguish CDA in the following aspects: 1) CDA is a pioneering light-weight, bridge-like framework which succeeded to interlink context-aware ad-hoc and mobile networks at the application level. 2) CDA interlinks Ahoy and MSP in such manner that it preserves the characteristics of the underlying networks and application environments i.e. it keep the lightweight character of Ahoy as much as possible unaltered, yet also interface it with the SOA based MSP. 3) CDA rises from the architectural level abstraction to the working prototype implementation and shows that such bridging is indeed feasible.

3. Brief Introduction to Ahoy and MSP

Throughout this paper, we split CMS operations into three distinct functional parts. The first part is *context announcement* which is responsible for publishing the context sources so that they could be discovered. The second part is *context discovery* which enables the interested clients to query for a particular context source (based on the provided context information) and receive the specifications about contacting a particular context source. The third and final part is *context acquisition* i.e. obtaining the context information from the selected context source.

3.1. Ahoy

Ahoy [7] is a lightweight CMS targeted towards the resource-poor, wireless multi-hop ad-hoc networks. Ahoy is developed and tested on Unix-like (GNU/Linux) platforms and its stand-alone modules are written in *Ruby* programming language. Each Ahoy node (either context source or client or both) must have Ahoy daemon running on it. The Ahoy daemon communicates with the other Ahoy nodes through UDP sockets and with the local context source or client running on the same node through Unix-specific AF LOCAL sockets. Figure 1 shows an Ahoy node with its two main components and their functionality.

Attenuated Bloom Filter (ABF) is a unique adaptation by Ahoy to represent the availability of context sources. ABF is an array of Bloom filters [12] which consist of layers of bit vectors to indicate the existence of context information a certain number of hops away in ad-hoc networks. Each Ahoy node stores

an ABF from each direct neighbor. By consulting ABF, nodes only send queries to destinations that are likely have the requested context information with a small chance of false positive.

Context Announcement: In Ahoy, a *daemon* on every node keeps track of the local context sources and the sources from the neighbors within a particular hop distance (so called *depth parameter*) using ABFs. Any change in this situation results in the re-calculation of a new ABF and this change is sent to the neighbors one hop away, which subsequently re-calculate their own ABFs and send to all the nodes those are one hop away from them, so on and so forth. Thus any context announcement change is propagated *incrementally* in the Ahoy network.

Context Discovery: When an Ahoy client needs to discover a context source, it sends a query to the local Ahoy daemon, which initially checks first its local context sources. In case the context source is not offered locally, the daemon checks the ABFs it received from its neighbors. If any match is found, the query is sent to the neighbors whose ABF showed a match. The neighbors may either have the context source in which case it sends a query reply otherwise the query is propagated to their neighbors, the ABF of which revealed a match. The node that publishes the context source replies to the querying node by sending a query response (*IP address of the node and port number*) through direct unicast.

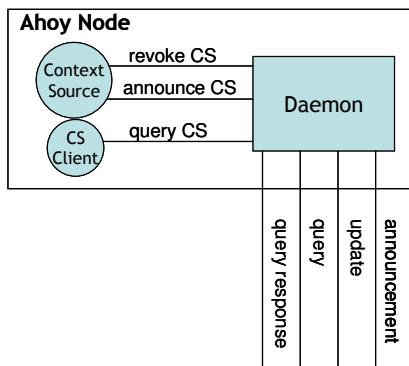


Figure 1: Components of the Ahoy node

3.2. Mobile Service Platform

Mobile Service Platform [5] is a SOA based CMS for the mobile networks. An MSP context source consists of two parts: *device context source*, which resides on the mobile device and its corresponding *surrogate* which is located somewhere in the Internet. These two communicate with each other using the *HTTP Interconnect protocol* which is based on the *Jini Surrogate Architecture Specification*. The other components of MSP include a *Surrogate Host*, *JINI lookup service* and client. The device context source

must first register its surrogate with the surrogate host, upon which the surrogate host registers MSP context source as a Jini service in the Jini lookup service. The MSP client must follow the principles of Jini service discovery and invocation to find the appropriate context source and obtain context information.

Context Announcement: The device context source contacts the surrogate host identified by a predefined URI (Universal Resource Identifier) and registers its surrogate with it. Once the surrogate is registered, the surrogate host discovers the Jini lookup service and according to the Jini specification, registers the context source proxy and attributes with the Jini lookup service.

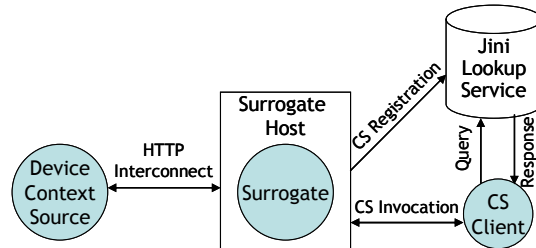


Figure 2: Components of MSP

Context Discovery: In MSP, the context source clients contact the Jini lookup service and send a number of query parameters (e.g. context source's Java interface type or a service attribute or a combination of both) which are mapped by the Jini lookup service to those of the registered context sources and if any suitable context source is found, its proxy is sent to the client as a query response.

Context Acquisition: The client instantiates the context source proxy received by the Jini lookup service and invokes the methods in the proxy, which in turn communicates with the context source surrogate using *Java RMI* mechanism. For certain methods that the client calls on the context source proxy, the surrogate contact the device context source, both of which exchange user-defined HTTP messages as defined in the HTTP Interconnect protocol.

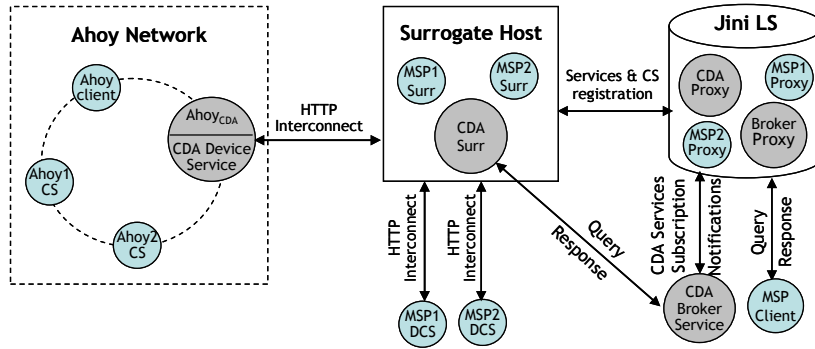
4. Context Discovery Adapter Requirements Elicitation and Architecture

We considered a number of alternatives to the bridge-like architecture for interconnecting MSP and Ahoy. However, they were deemed inappropriate. For example, a *CORBA-like federation system* would be too much overhead for lightweight Ahoy. Keeping in mind the requirement that the original CMSs should be left as much as possible unaltered, we are left with one choice: a hybrid, bridge-like architecture design.

The following requirements are elicited for CDA:

- Context discovery and exchange between Ahoy and MSP should be mutual i.e. the clients in both

the networks should be able to discover and obtain



Acronyms: CS: Context Source DCS: Device Context Source Surr: Surrogate LS: Lookup Service
Ahoy CS registration: Ahoy CS → Ahoy_{CDA} → CDA Device Service → CDA Surrogate → Jini LS → CDA Broker Service
MSP CS registration: MSP CS → Surrogate Host → Jini LS → CDA Surrogate → CDA Device Service
Ahoy CS Discovery: MSP Client → Jini LS → CDA Surrogate → CDA Broker Service
MSP CS Discovery: Ahoy Client → AhoyCDA → CDA Device Service → CDA Surrogate

Figure 3: CDA Components and Interactions

context information from the any of the context sources.

- The communication between the two CMSs should be seamless i.e. the differences between these two should stay hidden from the clients.
- The CDA should impose minimal changes and overhead onto the original design of Ahoy and MSP (so called *minimal impact requirement*).
- CDA should have a feasible implementation prototype. The implementation should show realistic performance, robustness and resource utilization features.

4.1. Architectural Components of CDA

In line with the requirements outlined above, we figured out that the CDA connecting Ahoy and MSP must have a gateway device called as *CDA node* which is capable of participating in both, the Ahoy and MSP network. Thus the CDA node must have Ahoy daemon (*CDA daemon*) for participating in the Ahoy network and *CDA device service* and corresponding *CDA surrogate* to participate in the MSP network. For the Ahoy network, CDA node could advertise MSP context sources as Ahoy context sources and vice-versa. However, this needs that the MSP context sources need to announce themselves using ABF based representation in the Ahoy network, while Ahoy context sources should appear as Jini services in the MSP network. We designed a special Jini service called as *CDA broker service* to aid with the ABF computations required for these two functions. There could exist multiple CDA device services and corresponding surrogates to cater with the multiple Ahoy networks. These components of CDA and interactions within are shown in the Figure 3.

4.1.1 Announcing Ahoy Context Sources in the MSP Network

The CDA node exposes itself as a nomadic mobile service in the MSP network by implementing CDA device service on it. The CDA device service and the local Ahoy daemon communicate with each other through UDP sockets across the loopback network interface. For registering Ahoy context sources in the MSP network, the CDA node first converts the ABF array representing the Ahoy context sources into a number of service attributes of the CDA device service.

These attributes are later sent to the CDA surrogate; which in turn registers CDA service with the Jini lookup service along with these attributes. Please, note that it is not possible to convert ABF based context source representation to the string format (as by the MSP context sources), because computation of ABF involves one-way hashing and is irreversible. Every time there is change in the Ahoy context sources, the CDA daemon is notified of it, which sends updated service attributes to the CDA device service. These are reflected in the Jini lookup service via CDA surrogate.

4.1.2 Announcing MSP Context Sources in the Ahoy Network

We considered a couple of alternatives to announce MSP context sources in the Ahoy network. For example, 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. Hence, the chosen solution is that the CDA device service pretends to be a neighbor of the local Ahoy daemon and announces the MSP context sources in ABF format through the local network interface. To avoid the resource consumption at the CDA node, the additional CDA Broker Service has the responsibility of converting the string identifier of MSP context sources into the ABF format. The ABF gets received by the CDA Surrogate, which at its turn forwards it to the CDA Device Service, from where the Ahoy daemon gets it. The CDA Broker Service registers itself as service listener to the Jini lookup service to receive events every time a MSP context source is added, removed or modified

and uses a callback interface to pass on the changed ABF to the CDA Surrogate services.

4.1.3 Discovery of Ahoy Context Sources by the MSP Client

For querying Ahoy context sources by the MSP client, it is required to convert the query name of the desired context sources into ABF format. However, MSP clients and CDA device service are inappropriate places to perform ABF computations, because in the former case this solution lacks modularity and scalability and in the later case exhausts resources at the CDA device service. Our chosen solution in this case is to use the *CDA broker service*, which is a special Jini service designed to factor out common functionality within the CDA protocol. It converts the string representation of the query name into ABF based representation. The CDA broker service registers with the Jini lookup service to receive events about the CDA services and it keeps a *local updated cache* of CDA services so that whenever it receives a query from the MSP client, it searches through the service attributes of the CDA services from the cached list and matches the query string's Bloom filter with the stored ABFs in the service attributes.

4.1.4 Discovery of MSP Context Sources by the Ahoy Client

The Ahoy client query originates in the Ahoy network and eventually reaches the Ahoy daemon on the CDA node. Since the CDA device service pretends to be the neighbor of the CDA daemon, the daemon believes that the MSP context sources are offered by a direct neighbor (which is the CDA device service). The CDA device service forwards the query to the CDA surrogate and the surrogate invokes a callback method on the available CDA broker service proxies. The contacted CDA broker service has a list of cached MSP context sources along with their description in the ABF format. The CDA broker service matches the received query string against the names of the MSP context sources and if a match is found, the broker service calls a remote method on the respective MSP context source to obtain its contact details which are later sent to the Ahoy client via CDA surrogate and device service.

4.1.5 Acquisition of Context Information in the Ahoy Network by the MSP Client

At the start of the CDA development, Ahoy did not have any support for context acquisition, so we developed one for the purpose. The context acquisition mechanisms such as Java RMI used by the MSP

seemed pretty heavyweight for resource constrained devices to which Ahoy is aimed at. Moreover, the Ahoy protocol needed to be modified to transfer serialized objects or the reference to a public URL where the RMI service proxy can be dynamically downloaded from; which is against the minimal impact requirement introduced in the Section 4.1. Our chosen solution for this problem is that of *Remote Procedure Call* (RPC), which is a lightweight precursor of RMI. Of the numerous implementations of RPC, we selected Apache XML-RPC. 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. To further take into account the minimal impact requirement, the Apache XML-RPC server runs on the CDA node, which takes care of contacting appropriate Ahoy context source based on the IP address and port parameters and returning the context information to the MSP client.

4.1.6 Acquisition of Context Information in the MSP Network by the Ahoy Client

As illustrated in the Section 3.2, MSP context sources support Java RMI-based distribution mechanism. For RMI to work the Ahoy client needs to get a copy of the MSP service's proxy. At the time of CDA development Ahoy did not support any message format that would allow the transport of the MSP proxy object. Therefore we decided to use the same XML-RPC acquisition protocol that we use for MSP clients to communicate with Ahoy context sources. An Ahoy client receives an IP address where the MSP service is running and a port number as a return value to the context source discovery, which suffices for the client to set up a dialog with the remote context source.

4.1.7 A Note on the Routing Challenges

The chosen context acquisition solution, i.e. XML-RPC, ensures that the client and server communicate one-to-one, without intermediary components. Our choice demands 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 and CDA 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. This means that all CDA components are required to have IPv6

addresses and IPv6 routing must be enabled in the networks where CDA is running. Otherwise, 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.

5. CDA Implementation Steps and Testing

The implementation of CDA followed a bottom-up approach. Initially, we created a basic JINI client with simple GUI and a simple CDA Service to test some standard JINI functions. Later, we started to interface the CDA JINI service with Ahoy. Scrutiny of the Ahoy daemon revealed that the best way to interface the Ahoy daemon and the CDA Service was through the loopback network interface, using UDP sockets. A second interfacing with Ahoy was needed to make sure that the CDA service listen to the announcements from the daemon. The message passing between the CDA 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. Next we expanded the JINI components and created the CDA Broker service, making the JINI client query the CDA Broker service instead of the CDA Service. The Broker was made to store local caches of the CDA and MSP Service references in the form of hashmaps. The CDA Service was made to send announcement messages to the daemon. The end result was that both Ahoy and MSP services could be successfully announced.

In the next phase, we transformed the CDA Service from a simple JINI service into an MSP service, i.e. two separate components: CDA device service and its surrogate. The conversion encountered a number of difficulties (e.g. broken MSP module dependencies, RMI security and code base issues) which are explained in details in [13]. A few more challenges were addressed toward the end of the implementation. For example, various hard coded parameters (e.g. IP addresses, Ahoy parameters) were made configurable. CDA performance tests are performed on the following test setup:

MSP machine: CDA broker Service, Jini lookup service, MSP client, MSP services, CDA surrogate and surrogate host. These components form a symbolic MSP network.

Ahoy machine: CDA device service, CDA daemon and Ahoy clients running on an Ahoy machine form a symbolic Ahoy network.

Each of the 10 test runs consists of the following four 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

these steps was varied arbitrarily, however, step 3 always following step 1 and step 4 always following step 2.

The following parameters are measured: 1) The time taken to register Ahoy and MSP context sources with the MSP and Ahoy networks, respectively; and 2) The time taken by the Ahoy and MSP client to query for MSP and Ahoy context sources, respectively.

Figure 4 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 an average longer than the announcement of Ahoy context sources. We believe the main reason for the difference is that announcing MSP context sources requires that the CDA broker service calculates a new ABF every time. While, when announcing Ahoy context sources 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.

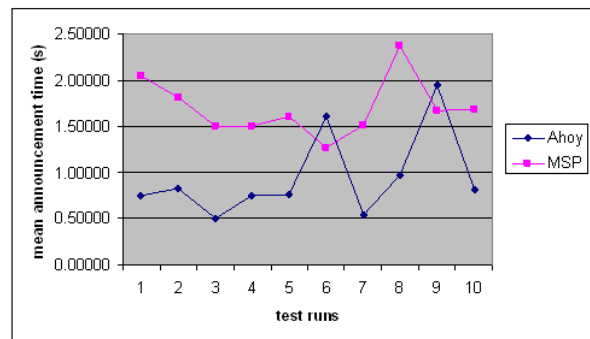


Figure 4: Measured mean announcement times of Ahoy and MSP context sources

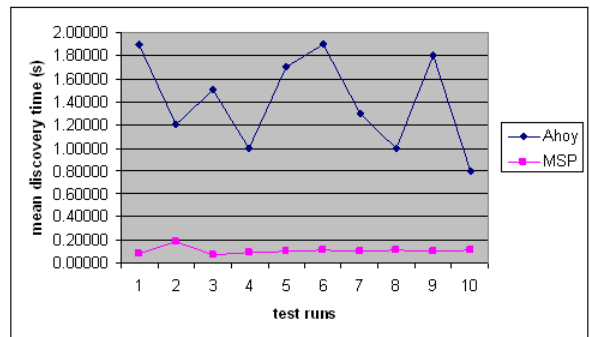


Figure 5: Measured mean discovery times of Ahoy and MSP context sources

Figure 5 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 context sources takes less time than discovering Ahoy context sources. During MSP service discovery the Broker Service matches the received query string against the cached MSP context

source 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 context sources. The results suggest that the Java-based bit array computations involved in the latter process introduce delay.

The reason that we do not report performance tests results for the context distribution is that the mechanism on distributing context information to clients are designed and implemented as a stand-alone module, however, the integration could not be realized in time due to time constraints. The various JINI, MSP and XML-RPC modules required by the current implementation of CDA node require in total about 4-5 MB hard disk space, which could be definitely supported by today's handheld mobile devices.

6. Conclusion and Future Work

The *Context Discovery Adapter* (CDA) is a *Context Management System* (CMS) bridge which interlinks two CMSs namely *Mobile Service Platform* (MSP) and *Ahoy*, with greatly different properties. MSP is a JINI-based CMS which enables handheld mobile devices capable of connecting to the Internet to take on a role of context source. Ahoy on the other hand is designed for mobile ad-hoc networks and built with the help of Bloom filters. CDA brings in significant contribution to research on context-aware computing by offering a pioneering solution to interlink context-aware ad-hoc and mobile network environments at the application level. The successful implementation of CDA interlinks Ahoy and MSP with minimal impact on both of the CMSs. 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, and due to time constraints it did not get integrated with CDA.

Considering the future work, one of the aspects of CDA that implementing support for IPv4 addresses in CDA would be also quite beneficial because currently not all routers and networks are IPv6 capable. The current performance evaluation results are reported for a smaller setup. Large number of diverse (including real-life) tests should be carried out to get an insight into the scalability and robustness of CDA. In CDA no attention has been given to security aspects with regard to context access. 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 CMSs besides the already integrated Ahoy and MSP. This might lead to a radically different CDA design.

7. Acknowledgements

This work is supported by Dutch Freeband Awareness project (under grant BSIK5902390) <http://awareness.freeband.nl>.

10. References

- [1] H. van Kranenburg, M. S. Bargh, S. Iacob, A. Peddemors, "A Context Management Framework for Supporting Context-Aware Distributed Applications", *IEEE Communications Magazine*, August 2006.
- [2] C. Hesselman, H. Benz, P. Pawar, F. Liu et. al, "Bridging Context Management Systems for Different Types of Pervasive Computing Environments", *First International Conference on MOBILE Wireless MiddleWARE, Operating Systems, and Applications (MOBILWARE 2008)*, Austria, February 2008.
- [3] O. Lehmann, M. Bauer, C. Becker, and D. Nicklas, "From home to world - supporting context-aware applications through world models", *2nd IEEE Annual Conference on Pervasive Computing and Communications (PERCOM'04)*, Orlando, USA, 2004.
- [4] I. Roussaki, M. Strimpakou, C. Pils, N. Kalatzis, et. al., "Privacy-Aware Modelling and Distribution of Context Information in Pervasive Service Provision", *IEEE International Conference on Pervasive Services (ICPS 2006)*, Lyon, France 2006.
- [5] P. Pawar, A. T. van Halteren et. al., "Enabling Context-Aware Computing for the Nomadic Mobile User: A Service Oriented and Quality Driven Approach", *IEEE Wireless Communications & Networking Conference (WCNC 2007)*, Hong Kong, March 2007.
- [6] F. Liu, G.J. Heijnen, "Context discovery using attenuated Bloom filters in ad-hoc networks". *Journal of Internet Engineering*, 1 (1). pp. 49-58.
- [7] Haarman, R. "Ahoy: A Proximity-Based Discovery Protocol". Master's thesis, Computer Science, University of Twente, January 2007.
- [8] R. C. A. da Rocha, M. Endler, T. S. de Siqueira, "Middleware for Ubiquitous Context-Awareness", *6th international workshop on Middleware for pervasive and ad-hoc computing (Middleware 2008)*, Leuven, Belgium, December 2008.
- [9] T. Buchholz and C. Linnhoff-Popien. Towards realizing global scalability in context-aware systems. *LNCS: Location- and Context-Awareness*, 3479:26–39, 2005.
- [10] "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>
- [11] "MobiBlog Project". [Online]. Available at: <http://www.ee.duke.edu/~romit/pubs/micro-blog.pdf>
- [12] P. Goering, G. Heijnen, "Service discovery using Bloom filters", in *Proc. 12th Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 2006.
- [13] H. Boros, "Context Discovery Adapter (CDA) Protocol", master thesis, Faculty of electrical engineering, mathematics and computer science, University of Twente, The Netherlands, August 2008.