Adaptive Context Aware Services

XAVIER RONDÉ-OUSTAU

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2006

COS/CCS 2006-4

Adaptive Context Aware Services

Xavier Rondé-Oustau

27 June 2006

School of Information and Communication Technology (ICT)
Royal Institute of Technology (KTH)
Stockholm, Sweden
Industrial Advisor Theo Kanter, Ericsson/KTH
Examiner Gerald Q. Maguire Jr

Abstract

Context information is information that describes the user's context. The goal of the Adaptive Context Aware Services (ACAS) project is to enable applications to use context information in order to adapt their behaviour to the user and his environment without requiring the user to manually change/manage parameters. While the concept of linking context aware entities together to form a logical "context network" was introduced earlier in the project, some questions regarding context information discovery and the discovery of context aware entities were previously unanswered.

The goal of this thesis was to design and evaluate such a context network -- allowing entities to discover each other and exchange information regarding their services and context information. For this purpose, a "Context Entity Registrar" has been developed allowing entities to register, thus they can easily be found by other entities who can query this registrar.

During the design of this proposed solution, a special focus has been given to the performance of the registrar, especially how it scales when answering a large number of requests, in order to validate the design's potential as a solution to context aware entity discovery. Measurements have shown that this proposed solution scales well, making it a key element of a context network.

Discovery of other entities and of context information play an important role to determine the performances of a context aware implementation. This masters thesis addresses first the issue of the architecture of the context network and then some tests to measure the performances of the proposed solution.

Sammanfattning

Context information är information som beskriver användarens omgivning. Adaptive Context Aware Services (ACAS) projektet har som mål att möjliggöra applikationer att använda kontext information för att anpassa sitt beteende till användaren och dess miljö, utan att kräva att användaren ska sätta eller hantera alla parametrar manuellt. ACAS projektet har tidigare infört konceptet "context network" som förbinder context aware enheter. Det finns dock kvar några obesvarade frågor angående upptäckt av context information och av context aware enheter.Trots att sättet att länka ihop kontext-medvetna enheter för att forma ett logiskt 'kontext nätverk' introducerades tidigare i projektet, finns det kvar några obesvarade frågor angående upptäckt av kontext information och upptäckt av kontext-medvetna enheter.

Examensarbetets mål är att utforma och utvärdera ett sådant kontextnätverk som ger enheterna möjlighet att upptäcka varandra och utbyta information om tjänster och context information. Därför utvecklades "Context Entity Registrar" så att enheterna kan registrera sig för att kunna bli upptäckta av andra enheter som kan göra förfrågningar till detta register.

Under designen av denna föreslagna lösning har särskild fokus lagts på registrens prestanda, speciellt avseende skalbarhet med avseende på antalet förfrågningar for att validera designens potential som lösning för kontext-medveten upptäckt av enheter. Mätningar har visat att lösningen skalar bra vilket gör kontext registret till ett nyckelelement i ett kontextnätverk.

Upptäkten av andra enheter och av kontextinformation har en viktig roll i att bestämma en kontext-medveten implementations prestanda. Detta examensarbete kommer först att behandla kontext-nätverkets arkitektur och därefter några testerna för att mäta prestanda i den föreslagna lösningen.

Acknowledgements

First of all, I would like to thank Prof. Gerald Q. Maguire Jr., my examiner, for the patience and the support he provided me with during the whole length of the thesis. As well, thanks to Dr. Theo Kanter, my industrial advisor, who gave an industrial perspective to this thesis.

I would like to thank all my colleagues in the lab, that were always here to help and to discuss problems I encountered, Alessandro Sacchi, Carlos Marco Arranz, Younes Oukhay, and Alisa Devlic. I hope that I have not disturbed their work with my questions!

During this thesis, I also had the chance to be part of a wonderful team, the XVIII International “Jazzers” BEST Board with whom I have been working very closely for the development of the Board of European Students of Technology. Thanks Elina, Jiji, Michal, Nadinak, and Susan for your support and understanding when I was not fully available when working on this thesis.

Enfin, pendant la rédaction de ce mémoire de fin d'études et durant la totalité de mes études, mes parents, ma soeur et mes grands-parents m'ont toujours été d'un soutien exceptionnel de manière à me donner les moyens de parvenir à étudier et à m'investir dans BEST tout en developpant mes capacités à exercer en tant qu'ingénieur responsible dans le futur, et c'est tout naturellement que je tiens à les en remercier vivement.

Table of Contents

Index of Figures

Index of Tables

1. Introduction

The development of adaptive context aware services is highly dependant on how context information can be discovered and accessed by an application (that in most cases is not executing on the device(s) that have such context information). When the context information to be used is known a priori by the application, then it is possible to access it directly; using for example a known network address. However, a goal for context aware services is to take advantage of information that is not known a priori. Therefore the challenge is to discover such information.

The ACAS project (Adaptive Context Aware Services [1]) has developed as part of its middleware the concept of a Context Network. This network links Context Entities together, in order to ease the discovery and transmission of context information. In order to achieve such goals, Context Entities should register with Context Entity Registrars so that the discovery of such registrars leads to the discovery of other entities and of the context information that is available via such context entities. The result of adding Context Registrars is an exponential increase in context information for a given communication effort, rather than a cost which is linear in the number of potential context sources.

Beyond the discovery of information, a goal of the ACAS project platform is to support the development of context aware services. However, the focus of this thesis is facilitating the discovery of context sources via the introduction of context registrars, rather than a context aware service itself.

The first chapter presents context services, their particularities and some examples in order to better understand the kind of services that the ACAS plateform tries to support. The second chapter deals with the context information management within the ACAS architecture, how context information transits from its source to the application that needs it. The following chapters present the context network and especially the context registrar, the element that is the centre of the context network. Its performance is analysed in the fifth chapter. The last chapters broaden the scope of the thesis to examine key issues regarding context information discovery, specifically entity discovery and reasoning about context.

2. Context Aware Services

Traditional electronic services often are organised following a client-server model: the client sends requests to a server that replies with answers. The user usually enters requests manually. A service that is more evolved can substitute a machine or process for the user. In this case, a computer automatically generates a request based on some parameters, user command, etc. Consider a request for a webpage. The user can directly enter the IP address of a website and get the page from the server (providing this computer implements a webpage content delivery service, i.e., is a web server). But the user can also enter the URL. The browser automatically generates a DNS request (to a DNS server) in order to learn which IP address this URL should be directed to.

For these traditional services, requests are triggered by the user. In contrast, context aware services take as input information that is likely to change in a non-deterministic manner, at any time. For example an air conditioner adapts its behaviour based upon the temperature of the room where it is in order to maintain the target temperature. Context Services work the same way. The development of communication tools has made it both possible and affordable to have the source of context information (such as temperature, position, etc.) at a different place from where this information is used, typically where the service is provided.

The main goal of context aware services is to facilitate seamless adaptation and to enable users of mobile devices by limiting the interaction required between the user and the services and devices. In other words, the aim is to provide more intelligent services and devices, focused on users' needs, preferences, and current context. Detailed studies and examples of such services can be found in [2].

2.1 Sensors and Applications

Context Aware Services extensively use information provided by sensors. One category of sensors is physical sensors. These sensors (together with their associated device drivers) are sources of physical information such as luminosity or temperature. A well defined driver is essential in order to be able to exploit the information generated by such sensors. What does a temperature mean if not sent together with its measurement unit? Thus the source of sensor information has to both provide the sensor values and additional information necessary to utilize this value (such as units, time of measurement, validity of the measurement, precision/confidence, ...).

A sensor can also synthesize information coming from several sensors, such as combining data from two images of the same scene in order to extract depth information. This is called sensor fusion . The sensor that provides this depth information, derived from the two images is a virtual sensor as it doesn't directly sense this information. The advantage of using sensor fusion is that the aggregation of information can be computed outside of the devices that need this info and sensor fusion can produce information which no single sensor is able to directly sense. (which reduces the resources that these devices need.)

In addition to physical sensors, there exist also software sensors producing information such as CPU utilization, list of active processes, maximum bandwidth of a link and its current utilizations. A key software sensor is the service sensor as it provides information about the different services available on a device. Once a device is discovered, one can therefore find out which services can be accessed and used. This is closely related to the concepts of self-description in both programming languages and device discovery (such as Sun's JINI [3][4]).

Applications can use this sensor information in different ways. For example, an application that is supposed to display a level of grey that is proportional to the noise level would need to regularly update their knowledge of the noise level information in order to update the level of grey displayed. The updating of this sensor information can be scheduled. On the other hand, we can have applications whose behaviour is event-triggered. Consider a simple application that displays green when the noise level is less than a predefined value and red otherwise, the sensor would send to the application an update only when the noise level exceeds a predifined threshold. Efficient communication between sensors and applications is a central part of the platform being developed by the ACAS project.

2.2 Context Information

Context Aware Services are composed of applications and sensors that exchange context information. The difference between raw sensor data and context information is that context information is to be understood by external devices and their applications. The ACAS project uses an XML based description of a context element [5]. The different fields of a context element are the following:

id	A unique identifier.
value	The value of a property of some entity.
datatype	The datatype of the value, like integer, real, string or XML.
unit	The property to which the value refers.
entity-reference	The URI to the entity which the context element describes.
time	Time and date when the value was captured or composed.
source uri	The URI to the entity which captured or composed the context element.
source content	Optionally a description that explains the context element.

Here is an example of a context element:

```
<acas:contextelement id="123c">
        <acas:value datatype="integer"
                unit="temperature/kelvin">292</acas:value>
        <acas:entity-reference rel="acas:dsv.su.se/k2/r7741/t"/>
        <acas:time>Sat Apr 24 00:05:21 CEST 2004</acas:time>
        <acas:source uri="uri:acas:dsv.su.se/k2/csf/apax"/>
</acas:contextelement>
```

Of course, as for all language supposed to explain something, words should be part of a well defined ontology shared between all devices that can have access to the system. This ontology enables all parts of the context network to understand each other. (See section 7.1) for further details of this ontology.) ACAS doesn't define a single ontology, but only defines a limited number of words and concepts for testing purposes and illustration. More details about the ACAS Context Description Language can be found in [6].

2.3 Example of Context Aware Services

2.3.1 The WhichJacket Service

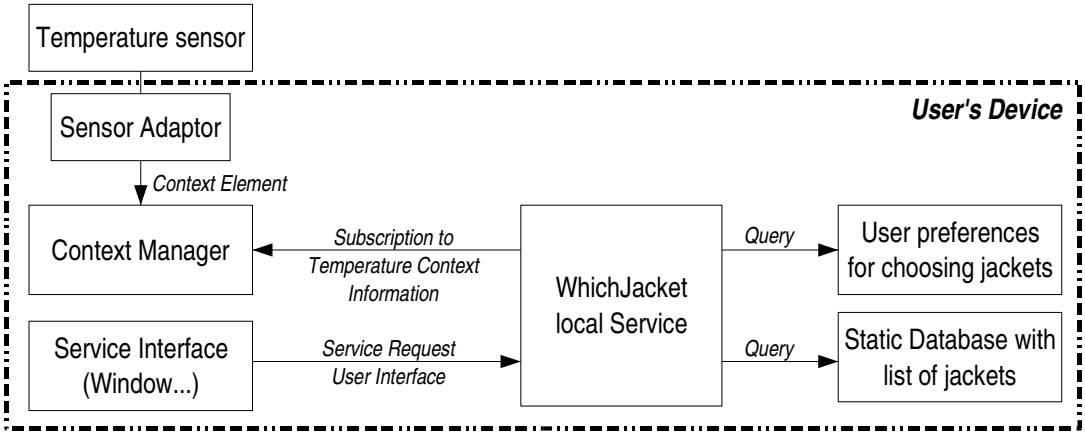


Figure 1: Schema of the "WhichJacket Service"

This is an example of a potential service that would use temperature and weather information in order to give recommendation about which jacket to take before going out. The user maps their list of jackets to different temperature intervals and different weather. When requested, the service would match a jacket with the current outside temperature and expected weather resulting in recommendations of which jacket the user should take with them. The major context information in this case is produced by a physical sensor (a temperature sensor).

2.3.2 The WhereToEat Service.

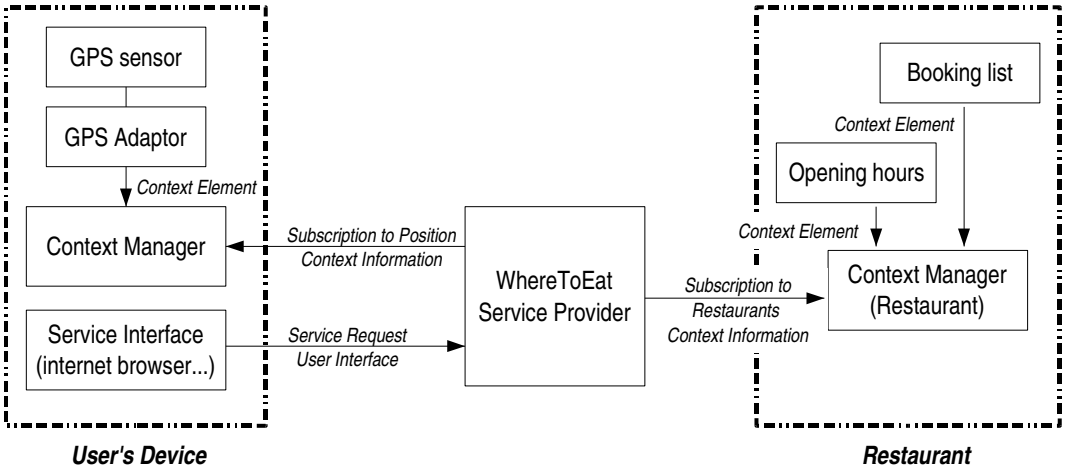


Figure 2: Schema of the "WhereToEat Service"

This service uses location information concerning the user's current location and matches this information with a database of restaurants. More advanced versions of this service could make restaurant suggestion based on user's agenda, preferences, prices, etc. The ACAS platform allows the service provider to access my location (context information) and some other software sensors such as my agenda and preferences in order to select the appropriate restaurant recommendation(s). Such a service could be further extended by providing detailed instructions about how to get to the selected restaurants, based upon my current position, the weather, errands along the way etc.

Context Aware Services can use a broad range of context information. The challenge is to make the discovery of information and services efficient as well as enable the different entitiesto both efficiently communicate and to understand each other.

3. Context Information Management

3.1 Context Manager as a link between sensors and applications

The Context Manager is an intelligent program/agent whose purpose is to provide applications with the context information that they need. It is a link between the sensors and the applications as well as with the other entities of a context network [7].

The link between a Context Manager and a sensor is defined by the sensor developer/manufacturer. Just as a sensor requires some drivers to communicate with a computer, it also needs an interface to the context manager in order to communicate within the context network.
ACAS defines the interfaces between the ACAS middleware and third-part sensors. The basic requirement is that gross information output by the sensor must be converted into an understandable context element as described in section 2.2. Other information and behaviours should be defined, such as the frequency of refresh, the possibility to get fresh info when requested, etc.

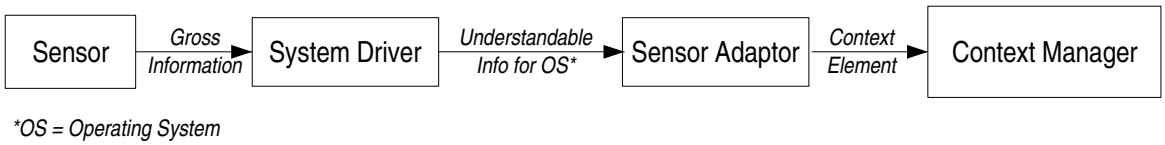


Figure 3: transformation of context information

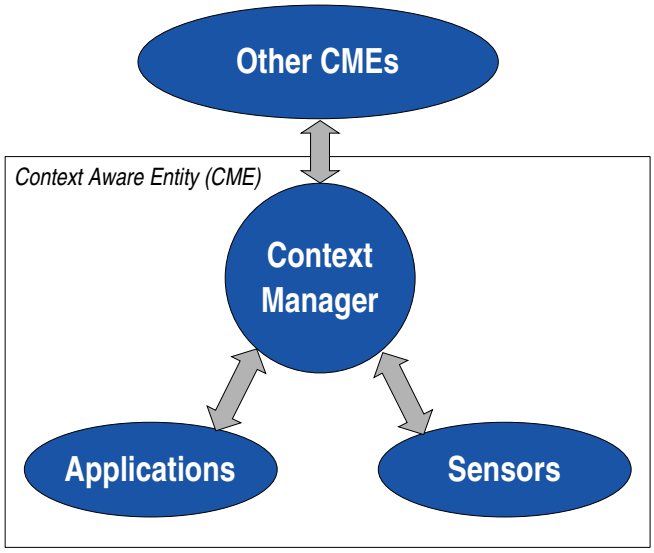
The link between applications that use context information and a context manager is in many ways similar to the link between a context manager and a sensor. An API is defined so that ACAS-compliant applications can communicate with the Context Manager using this API. Here again, the application developer must use the context manager API in order to communicate with the Context Manager. This API mainly specifies how an application can subscribe to some context information, how to forumate the requests, how to handle replies, etc.

The context manager also needs to communicate with other entities when looking for specific context information. An application that needs real time information about the temperature in Stockholm requires the context manager to find the correct information. Currently, we have some specific ways to get this information, i.e. the application connects to a predefined server that has such information. However, ACAS wishes to enable an application that doesn't know where the temperature in Stockholm can be found, to still manage to access this value and to use it. This will be developed later on in section 7.2.1; but in this section, we will examine the role that the context manager has to play in this context information discovery.

The third type of link provided by a context manager is a “forwarding link”. When a request from an application cannot be resolved locally, it is sent to other context managers that are likely to resolve it. To which context managers and how it is forwarded depends on the implementation and how the discovery process occurs, which information is available, etc. This will be briefly explained in section 7.3.

3.2 Managing Subscriptions

As a context manager is linked to both sensors and software, a subscription from applications must be handled and understood before being processed and passed on to the "management part" of the manager. The Management Center of the manager aims to optimize the processing of subscriptions and communicates directly with the sensor adapter of the context manager. We therefore have 3 distinct parties involved in managing subscriptions. Each of them is described below.



3.2.1 Application Interface

An application makes requests using the Tryton Subscription Language [8]. It indicates which information is needed and under which conditions (for example, when it changes, every 2 minutes, etc.). The challenge is to label the needed information in such a way that it can be understood by the context manager and retrieved either locally or remotely. See the paragraph regarding Ontology in section 7.1. The conditions are easier to formalise as they are mainly tests (equality, comparision) and variables such time (if a sensor is embedded in a device) or other characteristics. An example of a Tryton Subscription could be:

```
IF (ContextInfo.unit=="sine") AND (ContextInfo.value < -0.3333) THEN
  CREATE(eref=a.eref,
         unit="text/plain",
         value="below")
END
```

The IF part gives the condition and the CREATE initialises a context element. The context manager must first understand what "ContextInfo" is, look for it, and send information to the application context elements of the form defined by CREATE when the condition is true [8].

3.2.2 Sensor Adaptor

The sensor must communicate to the context manager which information it can provide and the format of the context element it can provide. The context manager must then appropriately label the sensor information to make it accessible from the outside of the device. For example, an temperature sensor would call itself “temperature” and the context manager would need to add some properties, for example, “owner = Xavier Rondé-Oustau”, “location = IBM T30 laptop number #”...

It is also important the the context manager knows the format of the data it communicates. For a temperature sensor, it needs specifically to know the unit (°C, °K, °F) so that other applications or context managers can adapt information from this sensor.

3.2.3 Context Management Entity

The “Managing Part” is a focal point that gathers and handles subscriptions coming from the local device as well as from outside. When a subscription cannot be handled locally, it is forwarded to other context managers that are likely to handle this request and if they can't, then the request is forwarded further until it either times out or reaches the maximum number of forwarding hops allowed.

Inside the Context Managemenet Entity, there are three main subentities: the Context Refiner, the Context Repository, and the Subscription Manager. A more detailed description of these components can be found in [9].

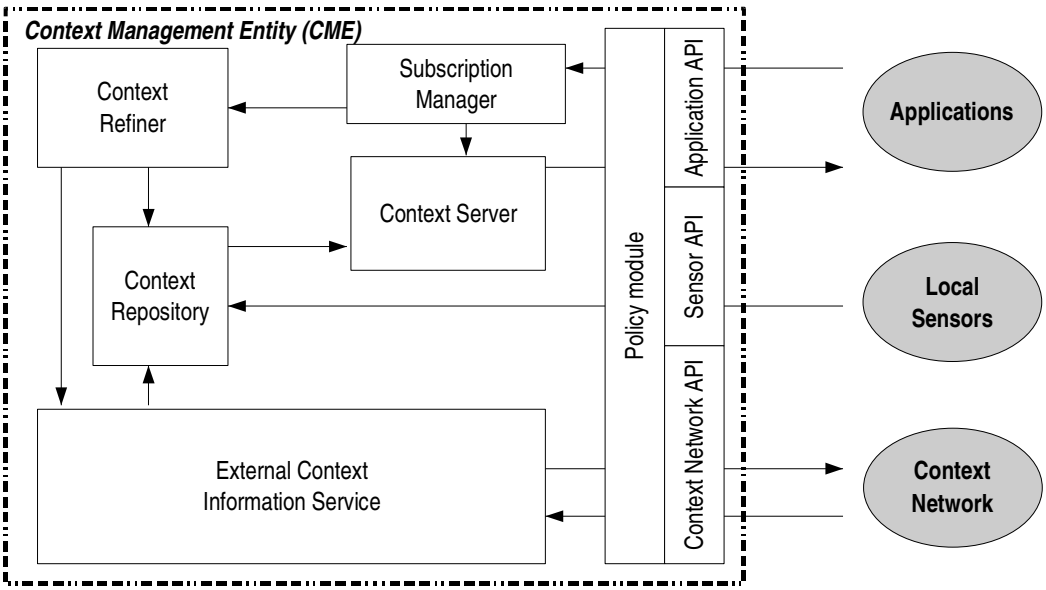


Figure 4: Structure of a Context Manager

The Context Repository stores the latest contextual information it has delivered. For example, if I have an application that receive updates every minute from sensors A and B, a copy of the value of A and B every minute will be stored in the Context Repository. When a second application subscribes to sensors B and C, a new subscription is set for the C sensor while this second application will get

information from the repository, that will be updated depending on registration from both applications. Therefore, the load on the sensor will be less as it will not need to serve each individual request from an application, but only the requests from the Context Management Entity that will act on behalf of all the applications. This Context Repository can also be useful to quickly answer requests without accessing the sensor, by acting as a cache.

The Context Refiner is used to process complex requests that cannot be handled directly by the Context Manager. It breaks complex requests into smaller requests that can be handled independantly. The resulting subscriptions are then sent to the relevant context managers or handled locally. The local Context Manager then computes the answer to the original request by aggregating context information coming from answers to these subsequent subscriptions. (see also section 7.2).

The advantage of using a Context Refiner is that you can outsource the handling of the request to another context manager that will issue the necessary subscriptions to the relevant sensors, bearing the cost of the transmissions, bearing the costs of the local processing to come up with the final result, and simply sending this result to the original requesting entity. This feature is particularly useful for context managers embedded in mobile entities (as today these entities frequently have limited computation and communication capacities).

The first role of the subscription manager is to keep track of all the requests that have been issued by local applications and to keep track of when the application must be updated. It also handles the subscriptions coming directly from the applications or the Context Refiner, if the requests need to be refined, to be forwarded to other context managers. Finally, the last important role of the subscription manager is to forward state changes of the local sensors that other context managers or applications have subsriptions with. This means that if a subscription requests “warn me when value of sensor A becomes positive”, the subscription manager of the context manager attached to A will track A for values becoming positive, using the sensor adaptor interface.

3.3 Policy management

The final role of the context manager unit is to enable policy based management, i.e., which local application can have access to which data, which local context information can be accessed by whom, who holds ownership of which data, how can owners of context information manage policies etc. That is the main purpose of this module withing the Context Manager [6].

Policy management is therefore enforced between the applications and the subscription manager or by the refiner and the subscription management in case a subscription needs to be refined. This ensures that the Subscription Manager only processes subscriptions from legitimate applications running on legitimate computers accessed by legitimate users.

However, the ACAS project has not yet developed a policy management mechanism. Any policy manager that already exists can be used for controlling access to a precisely defined resource by a well defined application. Authentication mechanisms already exist and their development is beyond the scope of this project.

4. Context Network

4.1 What is a Context Network

A context network is a logical network that supports discovery and exchange of context information between context aware modules like sensors, context information producers, service providers, applications, the nodes of a context network are therefore mainly the context aware entities themselves and the purpose of such a Context Network is to link them. More generally, a Context Network gathers entities that have discovered each other and can now interact with each other.

There are two main approaches to establish such a Context Network. First, the peer-to-peer approach. It consists of having each node playing a role of client or server. Each context entity supplies context information and/or it can requests such information from other nodes of the network. Learning about nodes and their characteristics is done without a central node; meaning that if a node doesn't respond it doesn't prevent the network from functioning with the remaining nodes. This methods avoid having “a single point of failure”. New nodes added to the network means new resources to be shared with the other entities, often adding both processing and bandwidth. The drawbacks of such an architecture are due to the lack of structured means to find routes, nodes, available information and services, even if some techniques exist to speed up the process, the lack of a central server makes it more difficult to reach performance close to that of client/server architectures for both discovery and routing.

Thus the second approach is a client/server architecture, as presented briefly above. A new entity joining a network would first register with a server and all communications, such as service discovery or context information discovery, would go through some central server(s). Clients are typically active, sending requests while the servers are passive, simply listening for and answering these requests. The contrast with a peer-to-peer solution is obvious. The failure of a main server is likely to interrupt servce for many clients. Moreover, each additional client will consume some extra resources from the server with regard to bandwidth, power, processing time, memory, ... On the other hand, servers are useful in order to be able to list registered clients registered, to find available information and services, etc.

ACAS decided to use an hybrid peer-to-peer solution (see Figure 5) in order to benefit from the strength of both solutions. The idea is that there is a central server that performs some minimal operations in order to speed up context information discovery as well as discovery of context aware entities present in the same network. This central server, called “a Context Registrar” is the first thing that a context aware entity will look for when trying to join a network – so that it can register its presence. The Context Registrar then maintains a list of registered entities and can disclose all or part of this list upon request; the response will depend on the entity requesting such a list. Thus Context Aware Entities can easily learn who is on the network and then perform service and information discovery systematically or in some other way.

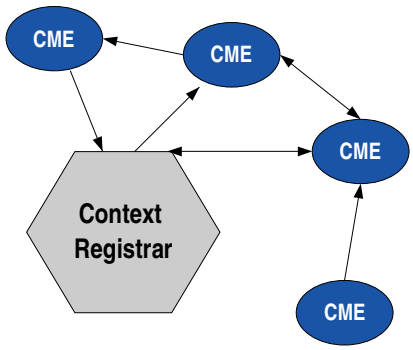


Figure 5: Context Network and Registrar

The Context Network can easily be compared to a “Context Aware” guest checking into a hotel using the hotel's registrar. The hotel's registrar keeps track of all the guests and their address. If another guest gave prior authorisation to the hotel, the concierge can disclose his address to the “Context Aware” guest which can allow them to exchange information or propose services to each other.

Even though the Context Registrar performs only basic registration, it is important to know how it behaves when the number of registrations increases, in order to identify the limitations of this approach, just as in any other client/server system. This thesis proposes a design for such a context registrar and evaluates if this solution is viable or if we should find another solution to perform context aware entity and/or context information discovery.

Figure 6 details a proposed registrar client inside the CME that would handle access to the Context Network. The External Context Information Service Manager contains modules to register to different Context Registrars, to find out registrars' addresses, to manage the different registrations and reregistrations; it contains therefore the “registrar client” that has been developed in this test implementation. This entity therefore maintains links with other CMEs either directly or through registrars.

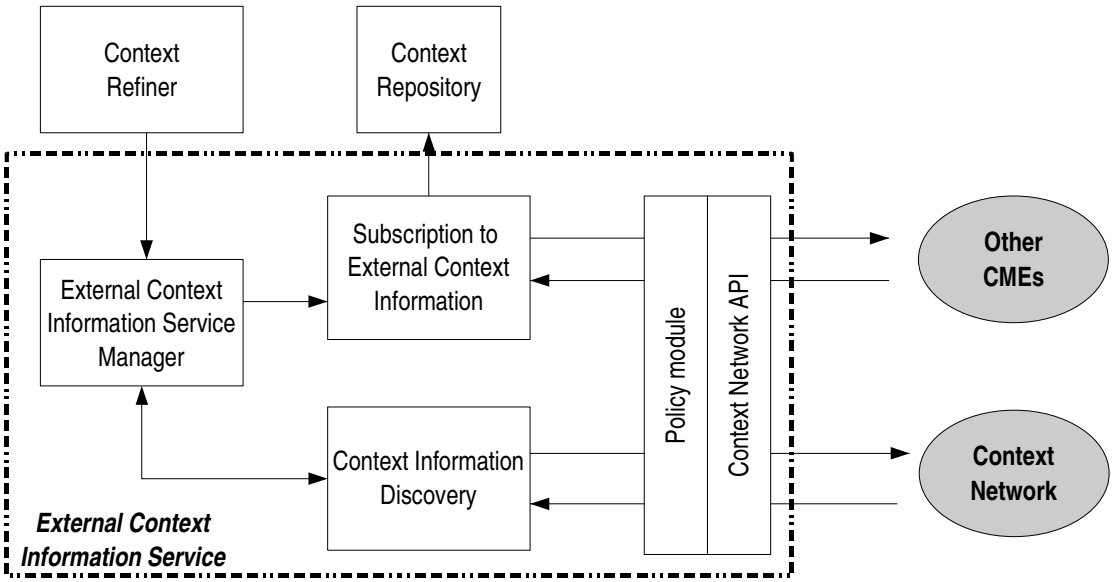


Figure 6: Context Network Interface in a Context Management Entity

The Context Information Discovery module maintains a list of available context information and context services available in the context networks the device is part of. It can also point to other entities maintaining such lists and performing the actual search and indexing of available services and information. This could be the case when the handheld device has limited computation/communication/storage capabilities: the "working module" could be placed in the infrastructure, such as in a personal server, the handheld device only communicates to the personal server addresses of CMEs discovered in the context network. The realization can also be mixed, for example external CMEs not reachable from the personal server are taken care of locally while other CMEs are taken care of by the personal server.

The second role of the External Context Information Service Manager is to match needs from local applications, coming from the refiner, with available external context information. Once the relevant information has been found, it passes on to the third module - the "Subcription to External Context Information" Module - the address of the CME holding the information and the description of the information needed by the refiner. This module takes care of subscribing to context information and outputs them in the local Context Repository. For local applications, it is therefore transparent whether context information is found locally or not. More information about Context Information Discovery can be found in section 7.

4.2 Discovery of Context Entities

Context Networks are hybrid networks using a central Context Registrar to keep track of entities on the network which are registered to the Context Registrar. This is the easiest and simpliest way to discover entities in such a network.

However, a Context Network can also be seen as a entity-centric network. In fact, an entity doesn't necessarly need to go through a registrar to discover other entities to exchange context information or services with. Additionally an entity can be a node in several different Context Networks that together form the entity's Context Network.

4.2.1 Discovery of Context Entities using a Context Registrar

A Context Registrar contains registrations from different Context Entities. However due to some privacy issues, registrars can play several different roles regarding whom they accept registrations from and whom they send replies to - each of these rolesthat will be analysed below. One can also refer to [10] for examples of scenarios for the use of adaptive context aware applications. The following uses of registrars can easily be linked to the case studies of [10].

4.2.1.1 Open Public Registrar

In this case, a registrar openly shares all information about all entities registered with it. It is not necessary that an entity is registered in order to access the list of registered entities. The registrar provides a public service that gives addresses of context managers. Additionally, any entity can also register with this registrar.

Examples from the current Internet of such behaviour include public community lists. I can see who is on the list, I can contact them, I don't need to register, and I can register if I want to be contacted later on. For example, a community of volunteer translators, that anyone can register to join, and anyone can contact them for free translations.

Such a registrar could be used to create a "volunteer community" of entities which interact with each other, but especially those that wish to interact, with external entities.

The main advantage of such a registrar is that it is easy to administer and to maintain. There is no special operation to perform on the context entities; neither on the registrar itself (to allow new entities to register) or to the clients who wish to register. By default, all clients will gain access to the registrar, then easily discover each other.

The main issues regarding such a registrar is that it is very sensitive to denial of service attacks due to a lack of control of the registering entities. An attacker could easily generate many dummy registrations to keep the registrar busy, and prevent anyone from successfully using this registrar. There is no way to know who is behind the address of a registered entity, nor is there control of the quality or truth of information provided. Therefore, such a registrar might not be suitable for use on a public network.

However, this kind of registrar would be particularly useful for devices belonging to a closed network controlled by a limited set of trusted users. For example, for a set of devices using only short range connectivity interfaces, the risk for interference from a hacker is very minimal or perhaps even non-existent. Such a registrar would allow easy connectivity of devices belonging to these users – for example, to form a personal area context network. The registrar would indeed be publicly accessible by any entity on the network. If it is hard or impossible for an unauthorised entity to access this physical network, privacy and trust are ensured through these mechanisms and the legitimate users can benefit from the openness of such registrar. When a new context aware device is bought by the user, there is no need to add settings within the context manager of this device or inside the registrar to make it connect to this Context Network using this open public registrar.

4.2.1.2 Restricted Public Registrar

By a Public Registrar, I mean a Registrar that can be accessed by any entity, to get information about registered entities. Restricted indicates that registration is restricted to some entities only.

A good exemple can be the yellow pages. Everybody, even if not registered, can get access to the addresses contained in this phone book in order to contact them. However, in order to register (to create an entry in this phone book), one has to pay a subscription fee and sign a contract with a phone operator and hence have the phone operator's authorization to be listed in these pages, also when such phone books are edited by independent directory publishers. Thus there is a process that restricts registration to this phone book even though it is for public use.

Such a Restricted Public Registrar can be used as directory for some services. There could be a registrar that only shops from a shopping centre can register to, allowing users' applications to monitor all kinds of special offers on a selected among these shops and service providers: i.e., those located within a defined shopping area.

The main advantage of this Restricted Public Registrar is to have quality control process to ensure that the quality of the list of addresses and enforce their common characteristics. This way, a context manager can learn something about these registered entities simply by learning from one entity, the registrar itself.

However, this control process means that some operations need to be carried out on the registrar and/or another entity in order to add a new entity to its database of registered entities. This can be costy and difficult to manage.

This kind of registrar could be advantageous in closed subnetworks: only entities from the subnetwork can register to the registrar while all entities from the main network would be able to access information from this subnetwork. The owner of the subnetwork or its administrator would guarantee the relevance of registered entites to the registrar and a new entity would be seamlessly registered as soon as it connects to this subnetwork. This reduces maintenance costs of the context network by matching it to a physical network.

4.2.1.3 Open Private Registrar

This Registrar requires prior registration before being able to access address of other registered entities. However, registration is open and any entity can join, as long as it agrees to contribute to the Context Network by sharing the address of its context manager. Privacy is still controlled within the registrating context manager itself and there is no prior requirement for sharing context information with others, just to give others the opportunity to contact this context manager in exchange for receiving the benefit to contact other entities.

A good example could be a community sharing (legally) files over the internet. In order to get access to the address of the other members of the community, one must first register and allow others to contact you. The files that a member wants to share depend on the user. Some communities could put requirements such as sharing at least 1 GB of files, but this doesn't mean the access is not open. By open, I mean that access is anonymous and open to any entity that meets the requirements, without necessiting authorisation.

The main advantage of this approach is that entities contribute at least a minimum to the context network by accepting to be queried via the registrar. However, as there is no control over the entities registering, problems due to denial of service can be very serious, just as in the case of Open Public Registrars.

However, applications for communities can be very interesting. Many scenarii regarding the use of mobile devices in the future emphasize the community aspect, the ability to momentansously join a community, for example, in an airport while waiting for a plane, in order to meet people or find something interesting to do based on context and available services/possibilities existing at this moment.

4.2.1.4 Restricted Private Registrar

This registrar is of course a combination of the cases (1) when an entity needs to be registered to access addresses of registered entities, and (2) the registration is restricted to some entities. Here again, we are still operating at without privacy: all registered addresses are available to all entities that registered.

Currently, this system exists for group services such as those proposed by Yahoo!. Once I am a legitimate member of the group, I can learn who is connected as well as receive additional services through the another member of the group.

Using a group registrar makes it easy for groups to secure their context network by controlling the way the registrar can be accessed. This context network gives members of the group the possibility to freely exchange context information as by definition, only entities registered to the group registrar belong to the group.

Services to groups can be provided by entities that are not group members (member = an individual) but could be an "associated group member", an entity allowed to register to the "group registrar" and share services with them. From an ACAS point of view, it doesn't make any difference if a context aware service is provided by a group member or by a thirdy part. Context information that such a "group service entity" could provide is the number of buddies online and available. I can subscribe to this information to learn how big my group currently is.

Of course, as this information is derived from context information coming from the group members, one can simply subscribe to availability (sensors) of each member and locally compute the number of available entities. However, instead of doing this refinement in each entity, it could be done once by the "group service entity" that would be continuously connected to the group's registrar.

Security is also ensured as in this case if the group service entity doesn't respond anymore, entities can still come up with this refined solution themselves while waiting for the group service entity to function again. As soon as it is functional, they can outsource context information refinement and subscription to this group entity, leaving more local resources available for other purposes.

4.2.2 Discovery of Context Entities without using a Context Registrar

An entity can use other discovery methods to discover other entities to exchange context information with. Even if discovery through acontext registrar should be widespread within the ACAS infrastructure, other discovery processes could be used by context aware entity. For example, C. Ayrault has described how Bluetooth discovery processes could be well adapted to local context aware entities [11]. Diego Delgado presents in [12] an evaluation of a Service Peer Discovery Protocol that could be used in order to detect context aware entities without using registrars.

One can also cache addresses of discovered entities it discovers for later use, while not being guaranteed to be able to connect later, for example when caching addresses of local entities and changing locations, these entities might not be reachable.

4.3 Joining a Context Network through a registrar

Once a Context Aware entity has discovered a Context Registrar (see section 6), it uses a first set of rules to find out what to do with this registrar. There is a configuration file, using a XML [13] structure and defining some attributes for registars as well as a default handling of registrars. In our implementation (see related source code), it's the file RegistClientConf.xml. The attributes of each registrar, including the default_configuration, are the expiry time of a registration and the time after which the client performs an automatic reconnection (an update of the existing registration). The value 0 means that there is no automatic reconnection, i.e., once the registration has expired, then a new one needs to be requested "manually".

In our implementation, we use a predefined list of addresses of potential registrars as the discovery of a registrar has not been tested and evaluated since it is very network dependant. The client therefore parses the list of registrars and matches it with the configuration parameters as described above to build a list of new registration requests.

The client goes through this list and sends each registration request to the corresponding registrar. The structure of a registration request is based on SIP registration requests [14], adapted to the ACAS infrastructure and looks like this:

- Registration Type

* REGISTER (for new registration),
* UPDATE (to update an already-existing registration), or
* REMOVE (to remove an already-existing registration).

- Address of the Context Manager to Register

This is mainly for the case of a context manager who does not register directly but through a proxy or an entity that handles registrations on behalf of context managers. However, in our implementation, we have not considered this case, thus and this address is de facto equal to the address of the context manager registrating with the Context Registrar.

- Registration ID

This is an identification for the registration. It is randomly choosen upon the first registration and with any update uses this Registration ID to refer to an existing registration. It is used both by the client and Context Registrar in order to log changes and update registration when needed, as well as for cancelling registration.

- Sequence ID

This is a number that is incremented each time there is an update or a cancellation of a registration (e.g. for a same Registration ID). This is done in order to order the registration requests and responses chronologically.

- Expiry Time

This is the time a registration will be valid, from the moment it is accepted, in seconds.

The registrar analyses the request and either sends an acknowledgement message to the client or sends an alternative registration proposition if the request was not valid. Both client and registrar use rules and a request sent or accepted by the client must follow these rules, and the same on the registrar side. In our implementation, the rules are defined in the classes RegisterMessage whose method isValid() returns true if the RegisterMessage satisfies the conditions defined in the method and false otherwise. In the experiments, unless otherwise specified, conditions on expiry times are set so that no first request satisfies the Registrar's rules, thus each counterproposition sent by the Registrar satisfies the rules of the clients so that we simulate a 4-way handshake transaction between the Client and the Registrar.

Of course, if a counterproposition from the registrar doesn't satisfy rules dicted by the client, then the client sends another proposition and so on until the maximum number of iterations is exceeded. However, in a realistic case, the client will first ask for the rules of the registrar (if those are public), and then compute a request satisfying both the client's and registrar's rules or abort, resulting in a maximum 4-way handshakes transaction, as tested in the following experiments. A client can also store parameters of registrars it gets to know so that, when registrating to a same registrar, it can directly compute a request that is likely to be accepted as it would follow the rules previously stored along other parametres if the registrar.

4.4 Discovery of a Context Registrar

As it has been presented above, the use of registrars is an efficient way to join a context network and discover new entities while minimizing the use of resources (such as time, bandwidth and processing): as one can learn a lot from a single query to the registrar. However, this also means that the registrar must be discovered. Below are several examples of how a registrar can be discovered depending on situations (as illustrated by scenarios).

4.4.1 When joining a subnetwork or a LAN

This case typical occurs when a user accesses a network he has never visited before. It could be when visiting a friend, going to a shop, in a conference center, etc. The first thing such a user does, generally in an automated manner, it's to send a DHCP request in order to configure its connection parameters. Such DHCP request returns an IP address, a network mask, DNS server addresses, etc. It is reasonable to have the DHCP server also provide addresses of local registrars allowing the user to query these registrars and to register.

DNS allows registering services attached to a domain. A reverse DNS query for the local subnetwork lists services available on the subdomain given that they are known to the local DNS server and that they depend on it to resolve the request [15]. However, in order to be useful, the client needs to understand that this “name” is that of a context registrar and not another service, similarly to other services that use standardised names and ports. The server itself doesn't need to have a standardised domain name in order to be resolved, only that its description of the DNS server needs to be standardised. We can compare this to resolving the local SMTP server of a subdomain: it can have the name “mail.mysubnetwork.net” or “smtp.mysubnetwork.net” but the record of this SMTP server must be standardized.

4.4.2 When joining a proprietary network

Some networks can be protected in such a way that the administrator sets up its own protocol, specific to the network, to give access to the network to authorised users. This is the case when using a GSM network for example, the operator can include in the preliminary handshake information about an available registrar (for example for a specific cell). The user is not proactively looking for this registrar but receives the information directly upon joining a network using some private means as opposed to open DHCP requests.

4.4.3 Using an external directory

The registrar can also be considered a service that can be advertised and searched for on an internet, for example by querying to a directory to return a list of registrars that meet some criteria. The user can then register with or query these registrars.

4.4.4 Manually entering a registrar addresses

One can also manually enter a registrar addresses that have been discovered through other ways. This is the option we choosed in these experiments. The registrar addresses are stored in an XML file that is used by the client to connect to these registrars.

In this ACAS prototype, users can define preferences regarding registrars, which ones to connect to, when, in which circumpstances, etc. The “registrating engine” inside the context management entity matches the list of registrars the user can connect to, from the discovery process, and the preferences in order to register when specified by the user.

5. Testing the Context Registrar

The Context Registrar is the central element of the context network that aims at speeding context information discovery as well as service discovery. The tasks of this registrar are to handle registration requests from users, store information, and answer requests regarding registered entities. This section is dedicated to tests of the registrar for handling registration requests in order to find what limits might pertain to an implementation of this registrar is running on typical computers. The three areas of focus are: average response to an isolated request, response to different bursts of requests, and behaviour under a high load of requests.

5.1 Timeline for a registration

Before testing the registrar in different situations, it is important to examine how it reacts to a single registration request in order to etablish a base timeline and to see where most of the time is spent, how long the request takes, etc. This experiment was carried out under two different conditions: first on a local area network (LAN) and then remotely via the internet, with the client and the servers being a few kilometers distant from each other, on separate networks linked by Internet. The registrar I have implemented uses TCP [17] in order to ensure reliability at the cost of the TCP overhead compared to UDP.

5.1.1 Description of the experiment

The client used for this test is a bit different from the normal client. It has been modified to send a new registration request every 2 seconds (since a simple preliminary test showed, it takes a lot less than one second to complete a registration). Therefore by sending requests at such an interval, we are confident that they won't overlap each other when no network failure (retransmission, packets getting lost,...) happens.

A first test was performed and all requests were answered without overlapping with each other, under similar conditions so that we can average the results. Another test was made with 200 requests and some problems started to occur such as retransmissions due to network failures resulting in shuffling of the requests, which we wanted to avoid.

Since it requires manual operation to “clean” these data and since we want to have an average normal timeline (disregarding some exceptional network conditions), a value of 100 sequences was a good trade off between “clean data” and the statistical relevance (the standard deviation of the processing time was around 3 ms out of an average of 15 ms response time). Of these 100, only 2 requests had an unusual behaviour, the first request (that initialise the client and the registrar) and another which had one frame more than the others, due to a late ACK sent by the registrar.

By averaging the times of the request/response for these 100 requests (reseting the time to zero for each initial frame, i.e. the TCP SYN frame), we have a good estimate of both processing and transmission times.

The registrar was running on a laptop, an IBM T30 1800 MHz, running Debian Sarge with Kernel 2.6.12-1-686 using a java 1.4 environment, located in the north of Stockholm on Tele2's broadband access network. The modified client was running on a server in the lab at the Wireless@KTH building. The client was running on a Dell server with an Intel XEON 2800 MHz processor and running Suse 9.3 with Kernel 2.6.11-4-21.9.

On both client and registrar, a network analyser, ethereal [16], was running in order to capture the packets exchanged by both entities. This way we can measure how much time is required at each side in order to process an incoming message and to send a relevant response. The delay D caused by the network can also be approximated by 1.421 ms as will be explained in the next paragraph. In fact, ethereal on the client sets time = 0 when the first socket is open whereas on the registrar, ethereal sets time = 0 when it actually gets the first frame from the client. In order to align the timeline of events on both registrar and client, it is necessary to figure out the offset between both ethereal captures, e.g. to find what time it is on the client when the time at the registrar is set to zero, this is the transmission time for a packet.

An approximation of such a delay D has been made by averaging over the whole sequence the durations of all possible round trips (Client/Registrar) that have been calculated using the following formula:

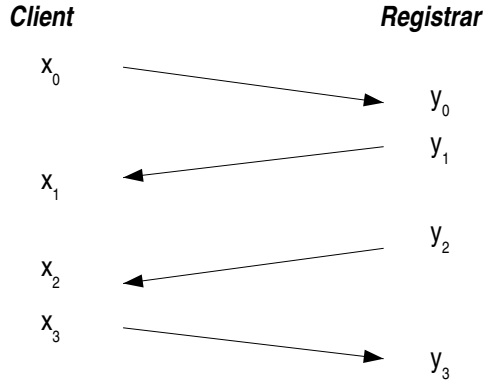


Figure 7: Schema of a 2-way handshake transmission

The round trip delay T can be approximated by the following:

$T_1 = (x_1\text{-}x_0)\text{-}(y_1\text{-}y_0)$
$T_2 = (x_2\text{-}x_0)\text{-}(y_2\text{-}y_0)$
$T_3 = (y_3\text{-}y_1)\text{-}(x_3\text{-}x_1)$
$T_4 = (y_3\text{-}y_2)\text{-}(x_3\text{-}y_2)$

To find an approximate value of T, we simply average the different T_i. We can generalize it (it can be easily demonstrated using recursion) that for A (set of frames sent by the client to the registrar) and B (set of frames sent by the registrar to the client), $T = \frac{\sum_{i \in A} y_i - x_i}{N_A} + \frac{\sum_{i \in B} x_i - y_i}{N_B}$ where N_A and N_B are the number of frames within A and B.
In order to find D, we assume that the channel is symetric so that $T=2D$.

5.1.2 Presentation of the timeline

The results we get from ethereal at each network interface is a list of relative times when a network activity has been detected at the port of the registrar, TCP port 54321. Besides these timestamps, ethereal also captures other information such as IP addresses, protocols, packet length, and information about the packet. It takes as its time origin, the time that the first network activity on port 54321 was detected. Of course, as we capture traffic at the registrar interface or at the client interface using two separate computers, timing relative to activity on port 54321 does not have the same meaning. At the registrar interface, the network card is actually listening to the client on port 54321 while at the client interface, the network card sends packets to the registrar on the port 54321 from a local port.

First of all, we can notice that the client sends the first packet from port P and the second from port P+1, the third from port P+2 etc. Thus, we can pair requests with responses simply by looking at the client port that was used. That is because the TCP standard that doesn't allow a port to be reused before a certain time after the socket has been closed in order to avoid interference with delayed retransmissions with a new connection.

By looking at the results, it appears that it sometimes requires 13 frames and sometimes 12 in order to have the full registration completed. A short analysis of the packets transmitted and captured by ethereal leads to the two following registration models:

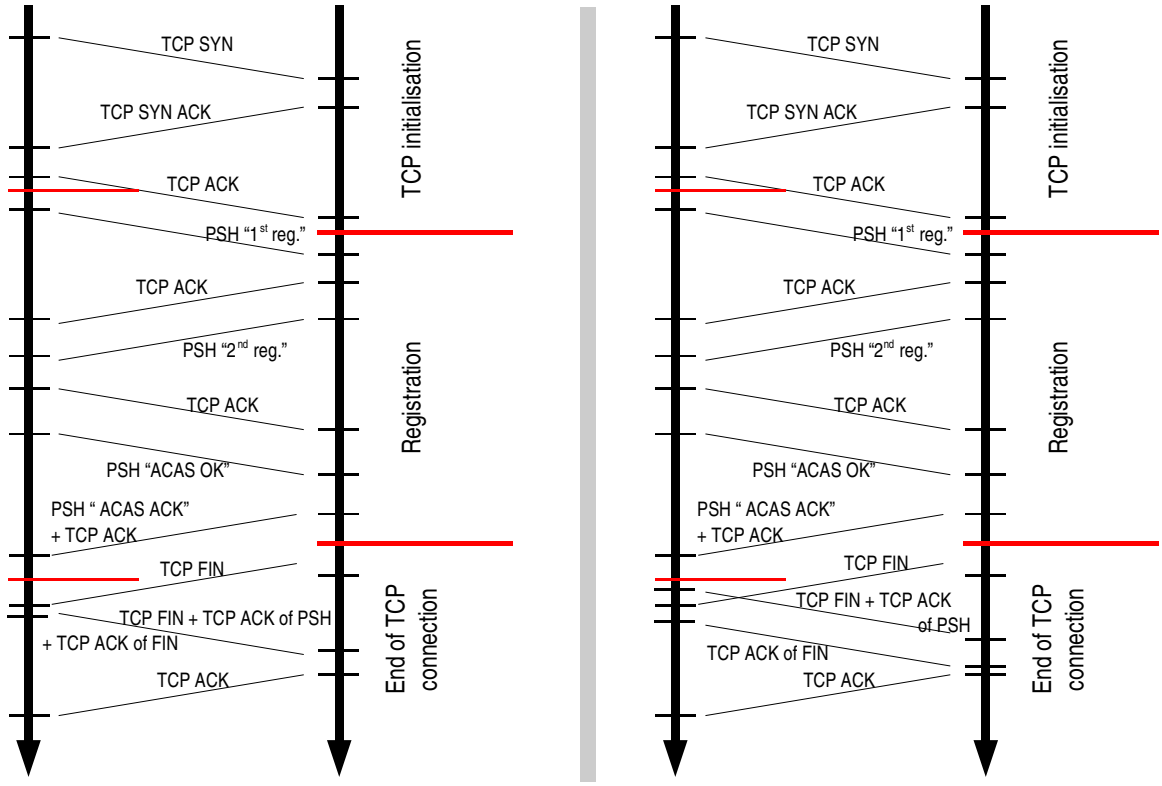


Figure 8: Model of Registration Timeline. Left: 12 frames case; Right: 13 frames case.

The analysis of the results has been performed using a spreadsheet application (OpenOffice.org 2.0) to import the output of ethereal, using both client and registrar data. The offset of the times at the registrar has been corrected using the delay we found above. In order to average the timestamps, we separate the 12-frame-registrations from the 13 frame-registrations. We have then 60 (resp. 38) measurements of the times when each frame is sent and received by client and registrar, in each model. We can compute an average for each specific frame of the protocol. This results can be found in Tables 1 and 2.

frame N°	*# frames*	*avg. (client) µs*	*avg. (registrar) µs*
1	38	0	1329.55
2	38	2670.05	1384.34
3	38	2696.47	4066.34
4	38	3128.47	5044
5	38	6641.53	5154.82
6	38	10178.26	8204.55
7	38	10194.26	11525.84
8	38	11087.76	12362.11
9	38	13883.37	12812.61
10	38	14269.08	12925.37
11	38	13981.03	15325.42
12	38	16624.97	15350.76
13	38	14285.45	15742.47

Table 1: Statistical results for the timeline of a registration (when 13 frames are sent)

frame N°	# frames	avg. (client) µs	avg. (registrar) µs
1	60	0	1493.87
2	60	2715.92	1548.4
3	60	2742.02	4138.05
4	60	3155.48	5161.05
5	60	6509.48	5278.73
6	60	9182.85	7488.23
7	60	9198.52	10646.95
8	60	10059.17	11530.73
9	60	13198.98	11931.67
10	60	13218.07	12043.7
11	60	13373.18	14857.95
12	60	16146.95	14883.22

Table 2: Statistical results for the timeline of a registration (when 12 frames are sent)

Explaination of the tables:

- The first column identifies the type of the frame within the registration process, as shown in Figure 9.
- The second column gives the number of frame in the sample (after having "cleaned" the data)
- The third column gives the average timestamps for each frame, measured on the client
- The forth column gives the average timestamps for each frame, measured on the registrar

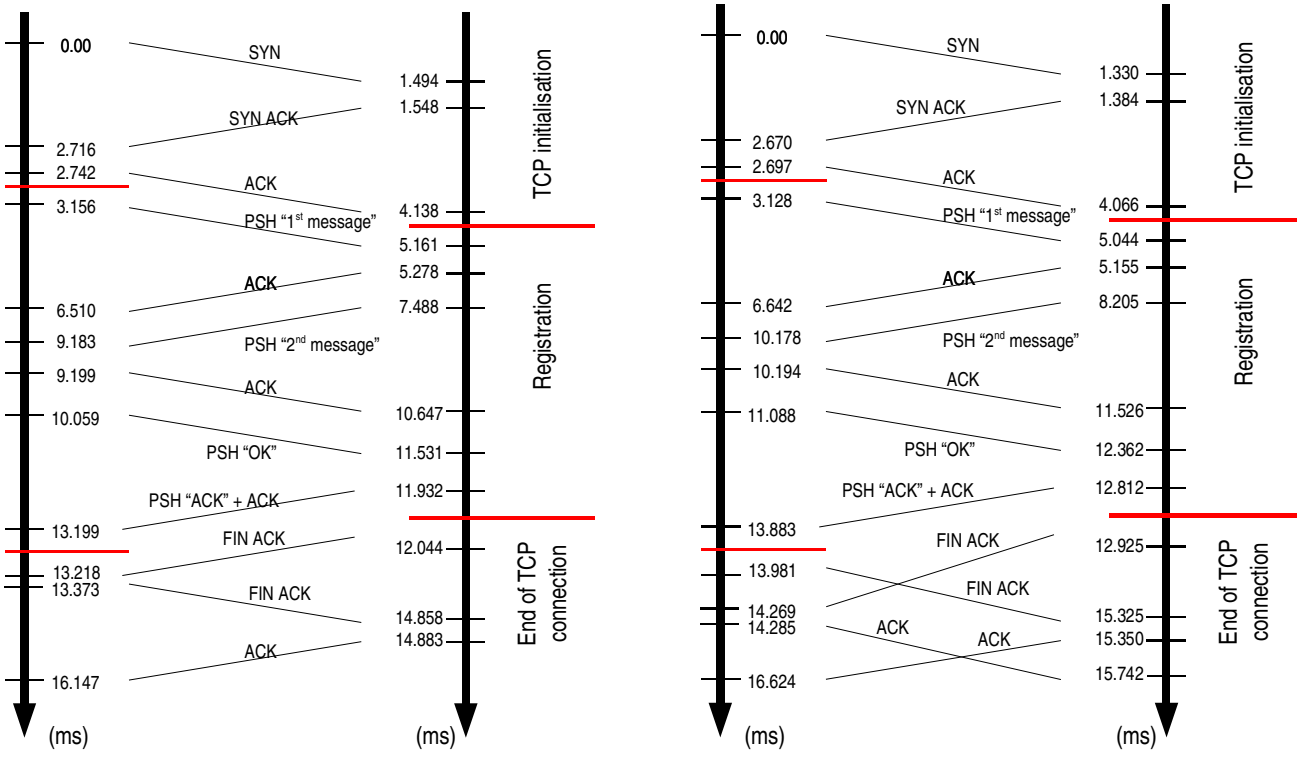


Figure 9: Registration timeline using experimental timestamps (left: 12 frames ; right: 13 frames)

TCP initialisation block

The first packet that is sent is the first step in establishing a TCP connection between the client on a port that changes from request to request. Note that the registrar always listens on port 54321 to incoming TCP connection requests. The registrar answers with an acknowledgement that the intial packet has been received by sending a packet with both the ACK and a SYN message to establish a connection from its side. The client replies with an ACK thus completing the initialization of the TCP connection. That is the normal TCP 3-way handshake for connection initialisation.

The processing times are very small in comparison to the network delay since it takes only 65µs for the registrar and 26µs for the client to send back these ACK packets.

Registration

This part describes the actual registration of the client to the registrar. The client sends first a registration message described in section 4.3, on port 54321 on which a registrar is listening. The registration is explicitly made so that it is **not** accepted by the registrar – who in turn proposes an alternative, in order to siumulate a 4-way handshake as explained in section 4.3. The client accepts the proposal by sending an "OK" message to the registrar confirming the registration in its internal database by sending an "ACAS ACK" message. Note that this ACK is different from the TCP ACK that acknowledges the reception of a TCP packet whereas this "ACAS ACK" is a TCP message (of type "PSH").

End of TCP Connection

The connection is closed first at server side. Just after the ACAS ACK message is sent, the registrar sends a FIN message to the client to inform it that the connection can be closed. The registrar enters the "FIN WAIT-1" state, waiting for a similar FIN message from the client in order to synchronise the closure of the connection.

The client, upon reception of the ACAS ACK message closes the connection and sends a FIN message together with the acknowledgement of the ACAS ACK. It enters "LAST-ACK" state, waiting for the Registrar to acknowledge the reception of this FIN message. The client also sends an acknowledgement of the Registrar's FIN as soon as it gets it. Since the Registrar sends consecutively this ACAS ACK and the FIN message, the client can acknowledge both messages at the same time together with its own FIN message. This explains why the closure of the TCP connection can be done sometimes with 3 and sometimes with 4 messages, depending upon if the first FIN message received by the client is acknowledged or not together with the second FIN sent by the client. In fact, if the FIN message from the registrar arrives before the ACAS ACK or around that time, the client sends three messages in one: its own FIN, the ACK to the ACAS ACK, and the ACK to the registrar's FIN.

Once the two FIN and two ACK messages have been exchanged, the client and the registrar need to wait for twice the MSL (Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system. This is defined to be 2 minutes by default in the IETF RFC 793).

More information regarding TCP protocol can be found in[17] and in the RFC 793 [18].

5.1.3 Analysis of the results

The main conclusions we can draw from this experiment is that the actual processing time of a request at the registrar is very low compared to delays in a public network and the potential retransmissions that might be needed. The exact processing time of a request by the Registrar cannot be calculated using this timeline since it does not take into account other processes that are running on this machine. However both the processing time upper bound and lower bound can be measured - we can assume that the minimum is the actual processing time it takes for the Registrar to answer a TCP message from the client; while the maximum represents the processing time plus the time the process had to wait to get the processor due to the demands of other processes.

When a client is situated far from the registrar, i.e.,not on the same local network, the maximum total processing time is about 4 ms. In this case a Registrar could deal with a continuous high load of registrations at approximately 250 requests per second (based upon the upper bound of the processing time). This will be however tested further in the next section dealing with the response of the Registrar to a burst of registration requests.

When a client is situated on the same LAN as the registrar, the performance is somehow different since the transmission delays are minimal (See Table 3). However the processing times are similar, especially regarding frame 6, sent by the registrar 3.3 ms after frame 5 which is of the same order of time as when client and registrar communicate through Internet (2.2 / 3.1 ms).

frame N°	# frames	avg. (client) µs	avg. (registrar) µs
1	13	0	111.54
2	13	190.46	154.38
3	13	215.46	325.31
4	13	604.85	750.85
5	13	813	777.62
6	13	4137.23	4082.31
7	13	4153.77	4271.92
8	13	5126.69	5238.46
9	13	5414.38	5347.23
10	13	5431.85	5421.54
11	13	5551.08	5659.15
12	13	5675.31	5678

Table 3: Statistical results for the timeline of a registration on a LAN (when 12 frames are sent)

5.2 Reponse to a burst of requests

In the previous section, we have seen how the registrar behaves for isolated registration requests in two different situations: first when the client and the registrar are on the same LAN, and

when they are attached to different networks that are linked by Internet. In this section, the focus is to identify how the registrar handles different sizes of bursts, how much time it takes to answer all these requests, how many errors occur, etc. The goal is to understand how this registrar scales, if the response time grows linearly when the load on the server increases or if it grows exponentially. For this purpose, we measure the average time for a request to be answered for different kinds of bursts. In the last section we saw that we can expect the registrar to handle more than 333 requests per second in case of a continuous uniform flow of requests. This section will determine first if it is true when the requests occur in bursts.

5.2.1 Test environment

To carry out these measurements, we use a similar test configuration as previously: the registrar still runs on the IBM T30 1800 MHz and the requests are generated by the Dell station located in KTH. The laptop will be located at first time in the same KTH Lab, i.e., on the same small local network area, and in the second test it will be located on Tele2's broadband access network, a few kilometers away from Kista, linked to KTH's network by internet. This way we simulate the situation when a context aware device joins a local network, as well as the situation when a context aware device connects to a registrar through internet.

The client is set to very quickly start new threads that initialize, in each of them, a TCP socket to send registration messages and handle response from the registrar.

5.2.2 Understanding which information is processed and how: with ethereal

Two kinds of information can be extacted from each test. The first one requires ethereal to capture each TCP packet that is going through the client's network interface. Most of the header of each frame is then imported into a spreadsheet application, specifically OpenOffice.org Calc 2.0. The most important information are the port numbers, the time the capture occured and the packet lengthes.

No	Time	Address Source	Port Src	Address Destination	Port Dst	Prot.	Length	Info
3279	0.713432	130.237.15.247	2381	213.100.40.6	54321	TCP	66	2381 > 54321 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=59337378 TSER=118481297
3280	0.713535	213.100.40.6	54321	130.237.15.247	2382	TCP	74	54321 > 2382 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=118481297 TSER=
3281	0.713550	130.237.15.247	2382	213.100.40.6	54321	TCP	66	2382 > 54321 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=59337378 TSER=118481297
3282	0.714211	130.237.15.247	2349	213.100.40.6	54321	TCP	428	2349 > 54321 [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=362 TSV=59337379 TSER=11848129
3283	0.714487	130.237.15.247	2350	213.100.40.6	54321	TCP	428	2350 > 54321 [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=362 TSV=59337379 TSER=11848129
3284	0.714655	130.237.15.247	2351	213.100.40.6	54321	TCP	428	2351 > 54321 [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=362 TSV=59337379 TSER=11848129
3285	0.714667	213.100.40.6	54321	130.237.15.247	2383	TCP	74	54321 > 2383 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=118481297 TSER=
3286	0.714691	130.237.15.247	2383	213.100.40.6	54321	TCP	66	2383 > 54321 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=59337379 TSER=118481297
3287	0.714696	213.100.40.6	54321	130.237.15.247	2311	TCP	72	54321 > 2311 [PSH, ACK] Seq=364 Ack=369 Win=6864 Len=6 TSV=118481299 TSER=59337
3288	0.714712	213.100.40.6	54321	130.237.15.247	2311	TCP	66	54321 > 2311 [FIN, ACK] Seq=370 Ack=369 Win=6864 Len=0 TSV=118481299 TSER=593373

Table 4: Example of data imported from Ethereal

The registration messages and the ACAS ACK messages are identified and sorted by using the particular length of their packets. Here, the registration messages contain more than 400 bytes and the ACAS ACK messages contain exactly 72 bytes and no other message sent using this registration protocol also contains exactly 72 bytes. We also track the FIN messages by analysing the information

field of each frame that contains “FIN” when this message is sent on the network. When such a message is sent, it means that the registration was successful.

It is important to identity the ACAS ACK and FIN messages as these are sent if and only if the registration was successful. Since ethereal drops frames especially as the load increases, it is important to know if a missing “ACAS ACK” for example means that the registration didn't work or if the packet has been received but not captured by the analyser. For a 5% drops ratio, we have only a 0.013% probability (i.e., 0.05*0.05*0.05) that all three “closing packets” are dropped by ethereal which is small enough to be neglected, especially since we have drops ratios below 5% most of the time (Ethereal displays the number of frames that went through a network interface and that could not be captured).

Registrations are identified by the port number used by the client. A port number is reserved until a certain delay (Maximum Segement Length) has elapsed (see section 5.1.2). Therefore, when a TCP connection has been initiated on port P by the client, the next connection in the burst happens too soon to reuse this port number, hence it uses port P+1.The available ports for the Dell computer range from 1024 until 29999.

The data imported into the spreadsheet are processed by a macro that indexes in a separate sheet all frames from a given TCP connection, i.e., a registration request, using the port number of the client as the differenciating criteria. In one row, we have the port (request identifier) followed by some timestamps and the number of frames exchanged on this socket.

The timestamps extracted by this macro for each TCP connection are the following:

- the time when the first packet has been sent (a TCP SYN)
- the time when the first registration message has been sent
- the time when an ACAS ACK has been received
- the FIN message sent by the registrar
- the FIN message sent by the client
- the time of the last packet sent on the socket (usually an ACK to a FIN)

We can compute the time when the socket was open by subtracting the first timestamp from the last one. We can also compute the actual registration time by subtracting the second timestamp from the third one.

The FIN timstamps are extracted in order to figure out whether or not a registation was successful, as some packets are dropped by the analyser. As explained abover, when a FIN message is sent, it means that the registration has been accepted as it occurs after the registrar sent the ACAS ACK. If on the sheet output by the macro one notices a line with blank timestamps for ACAS ACK, FIN1, and FIN2, it means certainly that the registration was not successful.

The numerical results I have calculated are the average of the registration times (including TCP overhead and excluding it), the average proportion of TCP overhead time in the total registration time, and for each average the standard deviation to describe how spread the registrations times are.

5.2.3 Which information are processed and how: with command line information

As explained above, the client consists of a main thread whose function is to start new threads that handle the registration requests. Just before the loop, the client writes to the standard output (the command line window from where it is run) a timestamp. At each ACAS ACK is received, the client increments a counter of successfully handled requests (it does not take into account TCP closing overhead). The main thread also displays a timestamp as soon as all the burst requests have been sent, then exits the loop. It also displays the number of ACAS ACK received so far. It does not wait for the last request to be answered, it just gives an indication about how many requests were answered while the client was sending the burst. Timestamps at both the beginning and the end of the burst allow us to find out an approximate rate at which the registrar was answering registration requests.

This information gives a good lower bound of the performance of the registrar. The difference between these two timestamps is actually bigger than the effective time used to handle a number of requests that is higher than the one displayed since they are displayed strictly before the first request is sent and strictly after the number of answered requests is displayed. However, due to the location of these instructions inside the main thread, it immediately gives a good estimate, of the performance of the registrar, without having to use ethereal and other tools.

The limitation of these results is that it is not possible to sort requests that take an extraordinary long time to be completed and the last requests that have been sent by the client. This is where the ethereal data above have an advantage, identifying worst cases and individual average time, i.e. how long it takes for one request to be handled in such a burst situation whereas the timestamps and accounting for ACK requests gives an indication regarding the maximum number of requests handled by time unit at full load.

5.2.4 Errors and problems during the tests

The main problem during these tests using ethereal is that for such a high traffic load, ethereal cannot capture all the frames and some frames are droped by this analysis tool. Even if it displays how many frames are droped, it is difficult to find out which frames are missing. However, it is not really critical in these experiments. The source of information used here always had less than 0.2% dropped frames when the final analysis is carried out, e.g. by importing data into a spreadsheet and running macros on it. For an average of 14 frames per registration, losses could affect at most 2.8% of the registrations. Moreover if a dropped frame is not the first SYN message from the client nor the last ACK from any entity, it doesn't have any effect on the time measurements, at most it can affect the “effective registration time” but this is not the main measurement result from these tests. As explained above, tracking the FIN messages gives a good accuracy even in the rare cases when the last ACK (for total duration) or the ACAS ACK (for actual registration duration) are dropped.

The impact of such dropped frames can therefore roughly be estimated to be much less than one percent which is very acceptable in comparison to the variance of the transportaion delays or other delays due to thread scheduling.

5.2.5 Test: 500 requests burst routed via internet

The first test used a burst of 500 registration requests, sent as fast as possible by the client. The configuration “via internet” means that the client was running in Kista while the registrar was running in the north of Stockholm, both hosts being interconnected through internet.

The first result we get is the number of requests handled between the beginning of the client's loop and its end. It took to the client 869 ms to send the 500 requests and meanwhile it got 240 ACAS ACK back from the registrar. This means that the registrar, can handle at least 276 requests per second under such a load. This confirms the results found regarding the timeline (see Figure 9) since the registrar takes around 4 ms to process the request (in addition to the transmissions client/registrar) in similar conditions (network, computers, operating systems).

Using the data from ethereal, we can also calculate the number of threads that are simultaneously handled by the registar. We use the data extracted by the macro performing the first analysis and listing various timestamps (see Table 4). We extract the timestamps corresponding to the first and to the last packets transmitted for each registration. The first timestamps are chronologically ordered in the first column, the last timestamps are chronologically ordered in the second column, independently. We assume that a socket is open (and therefore a thread is open on both client and registrar) at time *t* when *t* belongs to the interval [first timestamp; last timestamp]. A simple macro merges the two columns of timestamps, incrementing the number of threads index when a first packet is transmitted, decrementing this index when a last “FIN” packet is transmitted. The distribution can be found in Figure 10 .

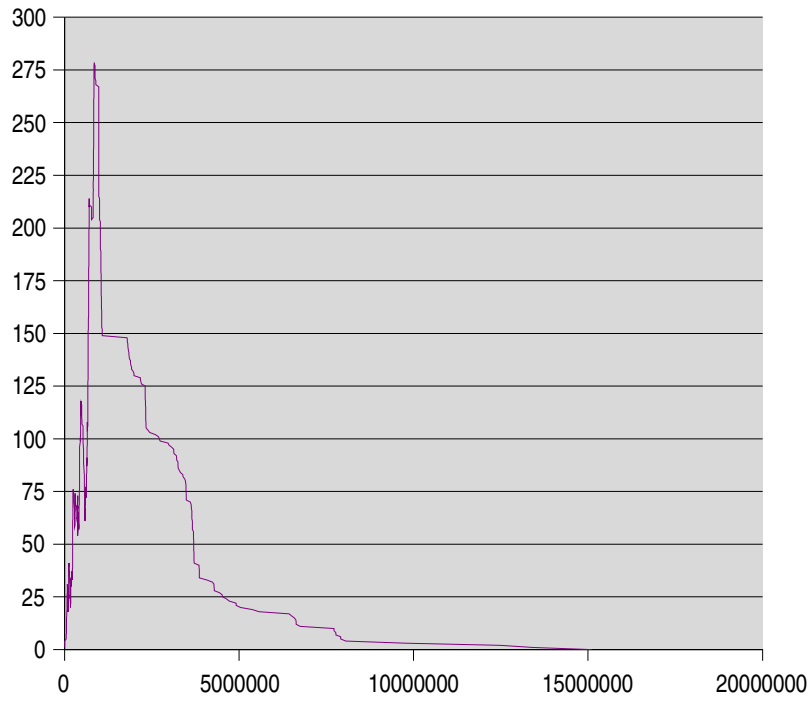


Figure 10: The number of threads running simulaneously
(X: time in µs - Y: number of threads running on the registrar)

The maximum, 276, is reached exactly when the client has just sent its 500th request, after 869 ms. Most of the requests are quickly processed after this time. The parts of the graph that are slightly horizontal for some time mean that the registrar is waiting for retransmission for several requests. The TCP protocol doubles the interval between two retransmission requests explaining these large delays for the very few requests left at the end. This is not due to the registrar, but to the network used, as in all public network applications.

Since the number of threads increases linearly until there are no new connections, this means that the registrar was not saturated and could certainly handle more connections at the same time.

	First Packet	*Registration Packet*	*ACAS ACK Packet*	*FIN Reg-Client*	*FIN Client-Reg*	*Last Packet*
Average Timestamp (µs)	496465.49	1181245.77	1440597.32	1451416.14	1492504.71	1568146.93
Standard Deviation (µs)	246471.55	1620731.17	1774233.36	1789860.76	1828259.84	1868766.67

Table 5: Average Timestamps for Key Frames

The results from the detailed analysis can be found in Tables 5 and 6. It takes an average of roughly one second for a request to be processed when sent together with 499 others within the same second. However, the dispersion is quite high meaning that if many requests are processed within a very short time, some others take much longer to be processed.

	actual registration	*total request time*	*actual/total time*
average time (µs)	258300.91	1071681.44	0.24
standard deviation (µs)	818974.18	1834389.76	

Table 6: Actual Registration and Total Intervals

These results show that it is possible to handle with good user performances 500 requests sent within the same second. There are not many cases when such a burst of requests could actually happen. It would mean that all passengers of a Boeing 747 would try to connect to a registrar within the same second upon arrival. Such a synchronisation of the registrations is not likely to happen as it takes at least a few seconds for everybody to get their mobile phones and start them.

5.3 Response to a high load

5.3.1 On a LAN

On the LAN within the lab, in the second test the client sent 50000 requests to the registrar. The purpose of such a test is to study the behaviour of the registrar under extreme loads, if it can answer requests in a reasonable time, if errors occur, if some requests are not handled, etc. The purpose here is not to make a detailed analysis of the data collected since it would be impossible due to the large number of frames expected, between 600000 and 700000 depending on the kind of network on which we run this experiment.

In this situation, it is particularly appropriate to use approximate total handling time with timestamps located in the client before and after the loop that initiates sessions. It took for example 73 seconds to send the 50000 requests and within these 73 seconds, 49822 requests have been processed by the registrar, without error or failure. This gives an average of 683 successtul registrations per second under such a continous high load, which confirms our prediction in the beginning, that the registrar should be able to handle more than 250 requests per second knowing only the response to individual isolated requests.

The situations in which there could be burst of requests are typically when a large group of users tries to connect to the registrar at the same time, following a particular event. For example, after a conference, when a plane has landed, etc. For a group of 2000 users that would all want to connect exactly in the same second (quite unlikely as discovery of the registrar would spread the registration requests over a larger period), it would take only 3 seconds on a fast and efficient local area network, using even rather an old machine to run the server. We don't need to investigate further this LAN case of LAN, the requirements in the most exigeant realistic situations are significantly lower than the system's capacity.

5.3.2 Over Internet

- ***5000 requests burst from Dell server to IBM laptop***

Under the same conditions as in section 5.2.5, the client (running on the Dell server) generates a burst of requests sent to the registrar (running on the old IBM laptop). The size of the burst in this experiment is 5000. In 13.235 seconds, the client manages to send 2243 requests, but errors occur for 2757 requests that cannot be sent "java.net.SocketException: Too many open files". This error occurs on the client side, when trying to open a new socket.

Among the requests that went through, the client received 2067 ACAS ACK in 13.235 seconds, which gives an average of 156 requests handled per second. Even if it is less than in the 500 requests burst case, it is still a good performance for realistic situations. The problem is that half of the requests did not get through. But as we have seen, it seems to come from limitations of the computer simulating clients, that cannot handle so many threads.

The calculation of the number of threads handled by the client, using a similar method as in section 5.2.5, shows that there is a flat range when the client has continously 1000-1061 threads opened. (see Figure 11) and cannot exceed this limit.

This is due to the per-process file handle limit that was set to 1024 in the linux kernel and that prevents the client opening open more than 1024 files simultaneously. However, this is also due to the fact that the client cannot close thread fast enough in order to open new ones. The reason for this could be that the registrar cannot handle requests fast enough, hence it is keeping sockets open for too long. Since each thread has to open a socket, the limit on the number of simultaneous files open also limits the number of threads that can be created.

We will test this hypothesis by inverting the roles of these computers. If the performance of the IBM laptop running the client are much worse for a similar burst, it means that we certainly reached limitations on client side since the registrar should run better on a faster computer.

We will test the second hypothesis also by inverting the computers (SuSE Linux for the Dell, Debian Linux for the IBM). We will then try to find out the maximum number of sockets the IBM, as client, can open.

If the performances are better by running the client on a slower computer (the IBM) and the registrar on a faster one (the Dell), it means that the performances of the registrar and of the computer that suports it are dominant.

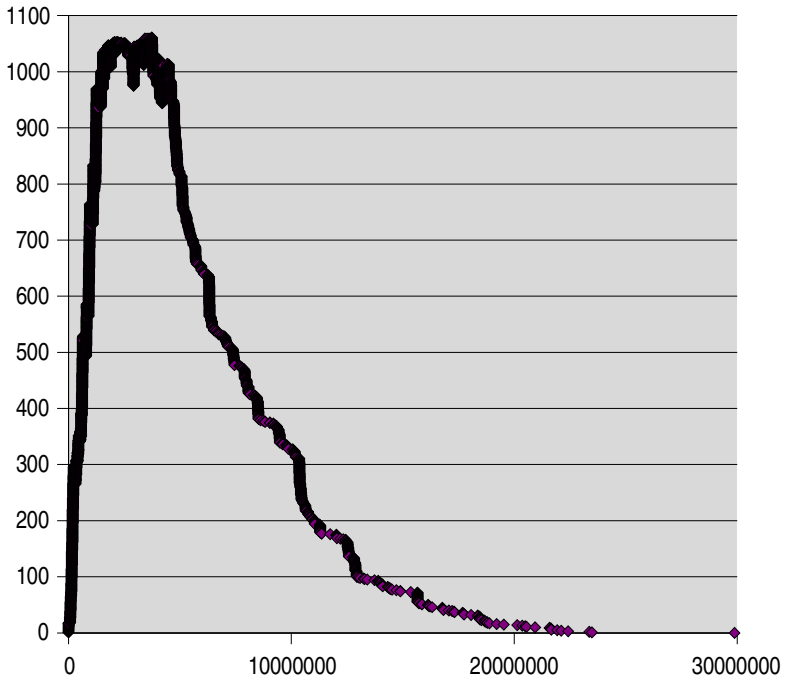


Figure 11: The number of thread running simultanously (X: time in μs - Y: number of threads running on the registrar)

- ***5000 requests burst from IBM laptop to Dell server***

The second phase of the experiment is to run the same code one knt

mt he onh other computers, to determine where the preformance limitations limitation comes from, thus the test occurs under the same network conditions. In order not to overload the client running on the relatively slow IBM laptop, ethereal is only run on the Dell server that also hosts the registrar in this experiment.

In 13.231 seconds, the client sends 5000 requests and gets 4153 ACAS ACK messages back. No error have been observed. The performance are still very high since the registrar handles 313 requests per second.

The maximum number of threads for this configuration is shown on the Figure 12 and is 701. As there were no errors, it means that there was no saturation of the server and that it handled requests fast enough to cope up with such a burst. The fact that the number of threads mainly increases throughout this experiment proves that the client still manages to send requests faster than what the registrar can handle them despite the fact that the computer hosting the registrar (the Dell server) has a much faster processor and a lot more memory available compared to the previous scenario.

The performance is much better in this configuration which means that the limitations are not bound due to the performance of the client running on the fast machine but they were due to hardware limitations of the registrar, when it was running on the old laptop.

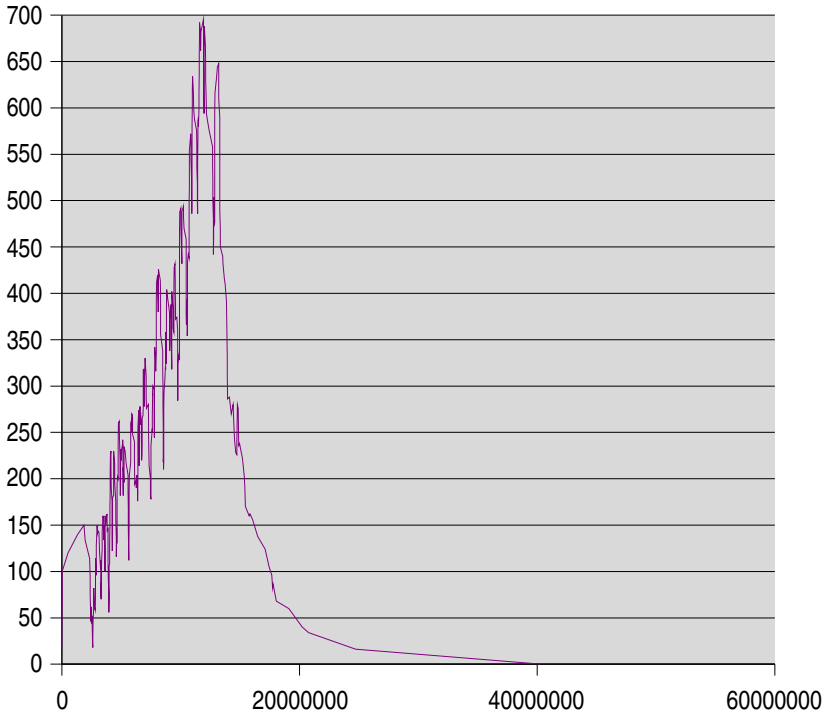


Figure 12: The number of threads running simultaneously as a function of time. (X: time in µs - Y: number of threads running on the registrar)

- ***50 000 requests burst from IBM laptop to Dell Server***

Since there were no errors in the previous experiment, we increased the load by a factor of ten times. In 124 seconds, the client sent 50000 requests, received 48158 ACAS ACK, there were 887 "too many open files" errors, and 17 time outs (set at 30 seconds). The performance is excellent as we reached almost an optimal use of the registrar, with few errors and a very high continous load. It handled an average of 388 requests per second.

In this experiment, Ethereal captured over 600 000 frames which makes an analysis using a spreadsheet impossible, even if we use filter to keep only interesting frames to calculate the number of threads: it would require keeping at least 100000 frames.

However, it is possible to estimate of the maximum number of threads that the client can handle. As long as the client keeps on sending requests on average faster than the registrar can

process, the number of threads mainly increases. Some requests are also finishing all the time which means that even if the server could accept a new registration as soon as one request finishes and keep a constant number of threads running, at a maximum, due to internal scheduling of tasks, many requests can be cleared before it processes new incoming threads, which explains why we don't have a strictly constant number of threads at saturation, but instead, having a range of number of threads

Once the client sents all the registration requests, already 48158 were finished (as it received the ACAS ACK), a few of these wee still active in the “TCP FIN” state. While 904 requests failed (we get this number through a counter when errors are handled and reported on the standard output). There are at least 50000-(48158+904)=938 requests still pending and as many threads running. Taking into account the fact that not all ACAS ACK have been already properly closed with the TCP FIN/FIN/ACK handshake, there should be roughly 1000 threads running on the client and as many on the server (since one thread on client = one socket = one thread on server). This confirms the limitation inside the kernel as the per-process filehandler was also set to 1024.

5.3.3 Some conclusions

The performance of the registrar is surely linked to the performance of the computer it runs on. The limitation is the per-process file handle that can easily be changed in the configuration file /etc/security/limits.conf. The Registrar is also limited by this parameter. Having a high limit means that the Registrar will be able to accept many requests at the same time, but it will take more time for each request to be answered which could result in some timing out, which results in more retransmissions, etc.

This experiment was also a good illustration of how much performances depends on hardware and software configurations. For the same test of 5000 requests, a poor configuration will not manage to support the load while a more recent computer will manage without any problem. Thus dimensioning of the server will be important if the load is expected to be high, but what a high load is changes continually with increasing server performance.

This experiment also showed that a good server running the registrar can almost handle 50000 requests in 2 minutes. This is more than enough for normal application, if we consider clients registering regularly every 10 minutes, it means the this registrar would be able to handle 250 000 clients, when running on a standard server, without any specific optimisation. This means that the registrar scales well enabling it to be adopted as a solution for context entity discovery.

6. Context Entity Discovery

In order to discover context information, an entity must first discover other entities that hold context information. A first way to find such entities is to query the registrar that will send back a list of addresses as described in section 4.2.1. There are then several solutions to get to learn which context information an entity B shares with another entity A that has just joined the network. The information this entity A would need are the headers of the context elements (more or less everything except the changing value, from entity B. However, if an entity holds a lot of different context information, it would mean lots of traffic to other entities that doesn't bring anything to B and only costs power and resources. Moreover, at each update of the list of context information available on device B, it would generate a high load on the device B that needs to update all the other devices that subscribe to B in order to be automatically updated.

It is possible for a context aware device to generates a list of locally available context information and context services. This list can also be seen as context information since it can be updated by the device at any time. An application could therefore subscribe to be updated when this list changes.

A way to overcome this high communication load can be to use a context information broker, that would gather information from the entities of the network and answer query from these entities. However, this client/server architecture suffers from its centralisation (if it fails, context information cannot be discovered anymore) and requires significat resources such as bandwidth, memory, and processing.

The solution seems to be once again hybrid between a peer-to-peer solution and a client/server solution as well as it should be open in order to allow different discovery processes to be used, such as the Bluetooth discovery process mentioned earlier [11]. However, there are also ways to improve this solution by relying more on the context registrar, so that an entity can more easily access the entities that are the most likely to hold the required information.

6.1 Context Entity Description

A user could set up a public profile for his entity so that other users could learn from this description and determine if the entity might hold the context information they are looking for. This information could include location, owner, type of device, brand and model, accessories, operating system, ...

This profile could be uploaded to the registrar together with the registration information and be retrieved by other entities querying the registrar. This would give the following advantages:

- queries to the registrar could be targetted and only selected contacts would be returned
- users would not query entities randomly, but rather based on their description

Not all the descriptions should be detailed in the public profile, for example, the references to a computer would give enough information to another entity to query a database containing detailed information regarding harware specifications, sensors, capabilities (screen resolution...) or

performances. Such standard information would be centralised on dedicated servers and databases such as those of the manufacturer.

Such profiles would take advantage of using a semantic language like OWL. This would allow entities to use these OWL relationships in order to precisely identify the devices that would be the target of their context information requests. This would limit the traffic generated by optimizing the information management of context entities, instead of browsing through lots of different heterogenous information.

7. Context Information Discovery

ACAS aims at implementing middleware that manages context information on behalf of the applications. This includes for example looking for sources of context information, analysing and understanding available information, as well as understanding requests from applications and serving these requests.

In the ACAS infrastructure, context aware application programmers would have to ensure compatibility and communication with the ACAS middleware by using an ACAS API (Application Programming Interface). The same goes for context information sources developers/manufacturers. The challenge for this middleware is therefore to understand the needs of the applications and the description of context information. It becomes clear that there should be a standardized vocabulary used by these three layers.

The second requirement on this middleware is to be able to perform some reasoning regarding requests and informations, to establish logical connections between words such as inclusion, disjunction, etc.

7.1 OWL as a solution for Context Information Management

For these reasons mentioned above, the OWL ontology language seems to be well suited to be used within ACAS infrastructure. OWL is a recommendation from the W3C (World Wide Web Consortium) to become a standard for the sementic web [19]. It relies on the XML structure, used for information description, and adds on the top of that a sementic layer to describe meaning of words and of word sequence to a machine.

OWL not only defines a group of words but builds relationships using special logical connectors such as “intersection”, “union”, “negation”, “inclusion” and “implication”. It is based on strict deduction from rules and hypothesis. As long as these rules and hypothesis are true, then the results of the OWL processing can also be trusted.

For example: an airbus is an airplane & A380 is an airbus => A380 is an airplane.

One of OWL's main strength is the possibility to import ontologies much like a java programme can import other Java classes or libraries that are understood by the the main programme. OWL works in a similar way, defining methods to access objects as well as operators and static variables. Entities can therefore query for understanding or perform a translation in case there is an OWL document that cannot be interpreted locally ensuring greater interoperability. The fact that an OWL ontology is based on deduction, makes translation automatic as long as the translator can understand both ontologies. In this case, this “translator” is built on a superontology that includes and links both ontologies.

Ideally, in ACAS, the context manager would have to deal only with OWL objects that would describe context information from sensors, subscription requests from applications etc. This is already compatible with the choosen “context element” structure (see 2.2) also based on XML. OWL would

be used in order to give meaning to the tags and their arguments that are the actual description of the context element whereas the values enclosed between some XML tags describes a state of the sensor that is not necessarly meaningfull in an OWL ontology.

OWL would therefore be involved at the interface between applications and context manager and between sensors and context manager.

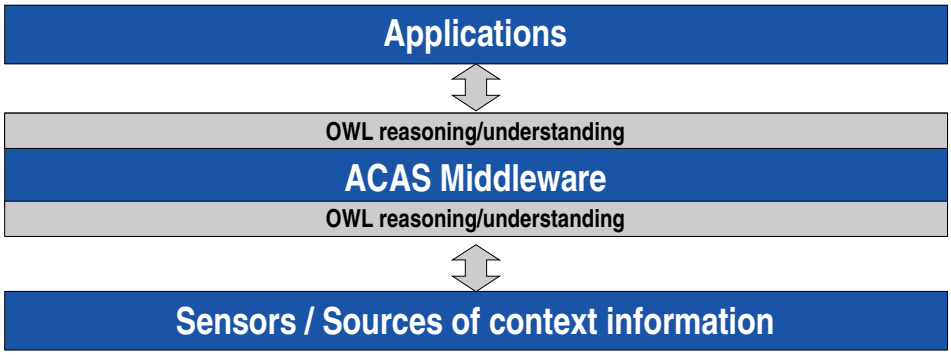


Figure 13: OWL interfaces in ACAS middleware

7.2 Reasoning about context

The first example of reasoning we will consider is OWL reasoning, applied to context information management. As explained above, it deals with the tags of the context element, or the context information description sent in subscriptions. By looking for subclasses and superclasses, of this elements, the context manager can find out if there are information that could answer the requests of the application.

7.2.1 Example of a subscription:

An application needs information about “temperature in Stockholm”. It sends a subscription describing this context information, it could have this form, using XML. The syntax is not strict and having a good comprehensive information structure would require important researches, therefore these example should be considered as illustrative and not as proposition of data structure.

```
<subscription>
        <context_information>
                <label = local_temperature>
                        <value>Temperature</value>
                        <where>Stockholm</where>
                        <unit>Celcius</unit>
                </label>
                <label = local_time>
                        <value>Time</value>
                        <unit>yyyy:mm:dd:hh:mm:ss</unit>
                </label>
```

</context_information>
<subscription_rules>

```
        IF (context_information.local_temperature.value < 0) AND
(context_information.local_time.value < 2006:03:31:08:00:00) AND
(context_information.local_time.value.mm = 30) THEN
        CREATE(eref=a.eref,
      unit="text/plain",
      value="FREEZING ALERT")
        END
```

</subscription_rules>
</subscription>

7.2.2 Explaination

The first part aims as declaring the different context information involved in the request. For the temperature, the value that is interesting is the temperature. The rest describe some properties of the temperature, like the unit and the location. For the time, we are specifying the unit (which is more a format in that case). This is important to notice that the "value" is not automatic and needs to be specified.

The second part consists of the tryton rules to handle the subscription, using attributes of the context information described above. It tells the context manager to update the application when the temperature is lower than 0 until 31 March 2006, with updates every full hour (mm=0).

Example of a sensor that produces such kind of context elements:

```
<contextelement id="123c">
    <value datatype="integer"
        unit="kelvin"
        type="temperatur">272</value>
    <entity-reference rel="acas:dsv.su.se/k2/r7741/t"/>
    <time>Sat Apr 24 01:00:21 CEST 2004</time>
    <source uri="uri:acas:dsv.su.se/k2/csf/apax"/>
    <location datatype = "string">Södermalm</location>
<contextelement>
```

7.2.3 Reasoning by the context manager

The OWL reasoning would look for some subclasses of "Stockholm" such as "Norrmalm", "Södermalm", etc. The logic of OWL would lead to conclude that "information in Södermalm are also relevant information about Stockholm". This context element can therefore be analysed further. Still using OWL reasoning, the appelation "temperature" is equivalent to "temperatur" (correspondance English/Swedish). This context element can be further investigated. Last OWL reasoning element, the unit of the temperature: Kelvin belongs to the upperclass "temperature/unit" as well as Celcius. An OWL relationship can establish that values in Kelvin are 273.15 higher than values in Celcius.

The second part is reasoning about context which means to apply rules based on context elements and their states. This context element indicates a temperature below 0°C (after conversion by the context manager), in Stockholm (since Södermalm is in Stockholm), before the date written in the rules of the request, at a time that qualifies to update the application. The context manager will therefore process the code that is in the "IF THEN ENDIF" section and update the application. This part is called "context reasoning".

7.3 Context Discovery "in chain"

This is a concept that already exists in peer-to-peer networks, but it is a key feature of Context Networks regarding context information discovery. The principle is that a request coming from outside can be rerouted by an entity. This feature, combined to the previous one, can enhance the chances for an entity to find the needed context information, each time profiting from OWL analysis of the profile. In order to avoid indefinite forwarding, some time-out can be implemented, making a request expire after some time, as well as a maximum number of hops between context entities.

It is important to make a distinction between the use of OWL for reasoning about context information, and the use of OWL for using information about devices.

- The first thing for a Context Manager to do is to analyse the subscription request (this has been presented in the last section). What does the application need, what could be satisfying, etc.

- The purpose of reasoning about device information is to efficiently route requests by knowing in advance how probable it is that a remote device can process the request (for example, the device belongs to a group to which the user has access), that the device has the relevant source of context information (this phone is an Ericsson AB1234, and by looking up in an Ericsson handset characteristics public database, that this device has a temperature sensor, ...).

- Remote devices accepting the request will then analyse it, using OWL reasoning, and refine it in order to find out how to solve it, if part of it should be forwarded, etc.

- Remote devices then apply OWL reasoning to the information about other devices in order to efficiently route the new requests, that are derivated from the original requests after having been through the refiner.

- This loop can continue until some mechanism stops it.

8. Conclusions and future work

8.1 Conclusions

The first purpose of this masters thesis was to evaluate what was lacking in order to enable context information discovery through the discovery of context aware entities. We proposed to use a registrar that would keep track of the different users registrating on the context network, giving them the possibility to find out who else is registered to this context network.

This registrar can have several uses depending on the context network it serves. Depending on its use it can therefore also be a central piece of the context network, profiding other services such as authentication, access control, profiles & presence information, etc.

The second step was to evaluate the scalability of such a registrar, i.e., whether it can support high loads, how high, and based on which parameters. The results here were excellent since even a basic and non-optimised JAVA implementation of the registrar running on a normal server can handle around 50000 registrations in 2 minutes. Other scenarios (for example when running the registrar on a more recent computer) have confirmed the dependence on the hardware and software configurations that were used, but showed that even with old hardware (such as my 4 years old IBM laptop), the performance was satisfactory enough for most possible applications (with one registrar and 50000 persons registering within 2 minutes , for example, in an airport, in a congress centre, etc.).

Finally, the last part of the thesis was to document different possibilities for context information discovery, how it can benefit from the registrar, what are the basic principles underlying reasoning, etc. This part aims to become a starting point for further research towards a viable platform for large-scale context information networks, such as pursued in FP6 Ambient Networks [20].

8.1 Future Work

Although the registrar scales well when it has to handle registration requests, it has not been tested regarding answers to requests regarding registered entities, their information, etc. We have seen that a registrar can be used at different levels, the basic level being to keep track of registered entities, to update the list, and to publish it so that any entity can learn the address of any other entities registered. However, we might assign extra tasks to the registrar in order for it to play a central role in the discovery of context information and services. It could gather information/characteristics about registered devices, it could find out which are the most relevant devices to route subscriptions to, it could route requests as well instead of sending back a list of possible entities, ... For all these applications, it is not sure yet whether it scales or not, nor what could be a realistic load. It depends very much on what context aware application and services will be created.

There could be different architectures in order to have a better performing registrar, while handling both registrations and complex requests. There could be different machines that are synchronized (which rate, how... these are good questions to answers), we could have also databases

different from the machines handling registrations and anwsering requests etc. The main problem a single registrar could have is to be overloaded with too many databases accesses and too much processing. How to solve this would be a challenge in order to have registrars working on large scales, like companies, network operators etc.

An other area to explore would be reasoning about context information and context entities. Even though there are some standards, they could certainly be adapted in order to answer problems specific to context information. The main problem is to characterise an information one is looking for, which properties it should have, how to formulate such characteristics etc. The next problem is to establish a match between requested information and available ones and especially to measure the quality of such match. It is likely that not all the characteristics of the required information can be found and thus approximative solutions would be given. Finding a way to evaluate the relevance of the results returned to the applications is important in order to give an indication to the user about the accuray of the application/results.

Finally, all the policy and privacy issues need to be carefully analysed, how to set up trust mechanisms in order to check the information that is not controlled by the owner of the application using the information. This can have a large impact on the quality of service as well as on the reliability of the service if information cannot be checked or authenticated.

References

[1] ACAS, homepage of the project: http://psi.verkstad.net/acas/, accessed on June 17th 2006.

[2] J. Kolari et al. "Context Aware Services for Mobile Users. Technology and User Experiences", Espoo 2004. VTT Publications 539. ISBN 951-38-6396-4. Available at http://virtual.vtt.fi/inf/pdf/publications/2004/P539.pdf on June 17th 2006.

[3] S. Oaks and H. Wong, Jini in a Nutshell, Oreilly, March 2000.

[4] Jini online ressources, http://www.sun.com/jini, accessed on June 17th 2006.

[5] A. Wennlund, "Distributed Context-Aware Support", Department of Microelectronics and Information Technology (IMIT), Royal Institute of Technology (KTH), September 2004.

[6] C-G Jansson et al. "Context Data Distribution. Concepts and Approaches in the ACAS Project", Royal Institute of Technology, May 2004.

[7] F. Kilander, "Distributed Context Data Management", Department of Computer and Systems Sciences, Royal Institute of Technology (KTH) and Stockholm University, January 2005. It can be found at http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-165/ , MCMP '05 First International Workshop on Managing Context Information in Mobile and Pervasive Environments Proceedings of the First International Workshop on Managing Context Information in Mobile and Pervasive Environments May 9, 2005, Ayia Napa, Cyprus

[8] C-G Jansson et al. "Context Middleware for Adaptive Services in Heterogeneous Wireless Networks", Vehicular Technology Conference, 2005. VTC 2005-Spring. 2005 IEEE 61st Volume 5, Issue , 30 May-1 June 2005 Page(s): 2954 -2958 Vol. 5.

[9] C-G Jansson et al., "Mobile Middleware for Adaptive Personal Communication", chapter VII.f in P. Corradi and A. Bellavista, "The handbook of mobile middleware", CRC Press, to be published in September 2006.

[10] Mobilife: "Initial Scenarios, Requirements and Guidelines: User-Centred Approach for the Design of Future Mobile Services and Applications", IST-2004-511607 MobiLife, D06b/D1.1b, February 2006.

[11] C. Ayrault "Service Discovery for Personal Area Networks" Master's Thesis, Royal Institute of Technology (KTH), Deparment for Microelectronics and Information Technology, July 2004.

[12] D. Delgado "Implementation and Evaluation of the Service Peer Discovery Protocol" Master's Thesis, Royal Institute of Technology (KTH), Deparment for Microelectronics and Information Technology, May 2004.

[13] XML specifications, http://www.w3.org/XML, accessed on June 17th 2006.

[14] J. Rosenberg et al. "SIP: Session Initiation Protocol", RFC 3261, IETF, June 2002.

[15] A. Gulbrandsen, et al. "A DNS RR for specifying the location of services (DNS SRV)". RFC 2782, IETF, February 2000.

[16] Ethereal, Network Protocol Analyser, http://www.ethereal.com, accessed on June 17th 2006.

[17] D. Corner "Internetworking with TCP/IP", Prentice Hall, June 2005.

[18] "Transmission Control Protocol", RFC 793, September 1981

[19] "Web Ontology Language" http://www.w3.org/2004/OWL/, accessed on June 17th 2006.

[20] N. Niebert “The Ambiant Network Architecture”, MOCCA WWI Symposium, Yokouka, Japan, March 2006. Can be found under “presentations” on http://www.wireless-world-initiative.org/Mocca_WWI_Symposium_Japan_2006/

COS/CCS 2006-4

www.kth.se