

# Adaptive middleware for context-aware applications in smart-homes

Markus C. Huebscher  
DSE Group, Department of Computing  
Imperial College London  
mch1@doc.ic.ac.uk

Julie A. McCann  
DSE Group, Department of Computing  
Imperial College London  
jamm@doc.ic.ac.uk

## ABSTRACT

We propose an adaptive middleware design for context-aware applications that abstracts the applications from the sensors that provide context. Further, we use application-specific utility functions to choose, given multiple alternatives for providing a specific context, one alternative at any time that provides the context for all applications, whilst maximising the applications' total "satisfaction" with the quality of context from the chosen provider. Our middleware also implements autonomic properties, such as self-configuration and resilience to failures, in the provision of context information to context-aware applications.

## Categories and Subject Descriptors

C.2.4 [Distributed Systems]: Distributed applications—*evaluating alternative service providers*

## General Terms

Adaptive distributed service provision, context-awareness

## 1. INTRODUCTION

Smart-home projects envision the home of the future as being filled with sensors that can determine various types of contexts of its inhabitants—such as location and activity—while applications in the home use this information to provide context-sensitive services to the inhabitants and for context-aware home automation. For example, applications of smart-home technologies and context-awareness include supporting people with health problems, such as elderly living alone at home, or general lifestyle improvement, e.g. a smart fridge that monitors its contents and warns the user of products that are expiring or finishing.

Because any specific context can often be provided by a variety of different types of sensors and used by different applications, a number of middleware projects have emerged in which the middleware layer provides context information to applications whilst abstracting from the sensors and context logic that are used to acquire and process that information, the most well-known project

being the Context Toolkit [3]. We extend the notion of middleware layer for context-aware applications, and design a middleware that exhibits autonomic properties. In particular, we consider situations where multiple sources may be available for the same type of context and introduce an adaptation engine that chooses one provider of context that is most appropriate among the currently available ones. In fact, we go one step further and assume that context acquisition is a costly operation, such that given different alternatives for acquiring one specific context, it is preferable at any time to use only one alternative for all applications interested in that context<sup>1</sup>. Our adaptive middleware uses utility functions to determine—given the Quality of Context (QoC) requirements of applications and the QoC of alternative means of context acquisition—which alternative should be used at any time. The proposed solution can also be generalised to the case where more than one context provider can be used simultaneously among the applications and we need to choose for each application the best provider of context.

One of our major goals in the design of our middleware has been to provide *adaptation with good performance*, i.e. a change in context provision is detected quickly and adaptation occurs swiftly. This is particularly important in health-related applications that monitor a person with health problems, as these usually require prompt responses to changes in the health condition of the person.

Section 2 explains the notion of context-awareness and describes common metrics to specify the quality of context. Section 3 looks at the structure of our middleware, in particular context acquisition from sensors and delivery to applications, while Section 4 describes the adaptation component of our middleware. Section 5 looks at the issue of trust between components in the middleware. We then describe related work in Section 6 and conclude with Section 7.

## 2. CONTEXT-AWARENESS

Context-awareness is the ability of an application to adapt itself to the context of its user(s). A user's context can be defined broadly as *the circumstances or situations in which a computing task takes place* [6]. One of the most common contexts is the location of the user (or of objects of interest). In smart-homes, location can be obtained using a variety of different alternative sensor types, including ultrasonic badges, RFID-tags, video cameras and even pressure sensors in the floor [5]. The quality (which is quantitatively application-specific) of the location information acquired by different sensors will however be different. For instance, ultrasonic

<sup>1</sup>Whether this is true can certainly be debated and we agree that it does not hold for all types of sensors. Wireless sensor network and sensors that are battery-powered or share a low-bandwidth communication channel, however, are examples where sensor data acquisition does come at a cost.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing  
Toronto, Canada  
Copyright 2004 ACM 1-58113-951-9 ...\$5.00.

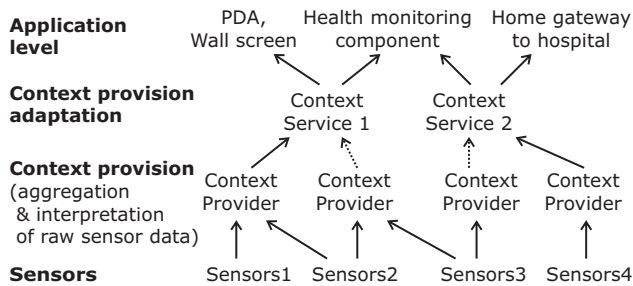


Figure 1: Layers of context provision in our middleware

badges can determine location with a precision of up to 3 cm, while RF location is limited to 1–3 m. Thus, we can define properties which we call *Quality of Context* (QoC) attributes that characterise the quality of the context data received. QoC is, as we will see in Section 4.3, essential to our middleware in choosing the best alternative among the available ones for delivering a specific type of context.

## 2.1 Quality of Context

While different types of contexts will have QoC attributes specific to them, there are certain attributes which will be common to most contexts. Based on [2], we identify the following common attributes:

**Precision** measures how accurately the context information describes reality, e.g. location precision.

**Probability of correctness** measures the probability that a piece of context information is correct. For instance, determining the current posture of a person (sitting, standing, lying on the floor in distress) using a video camera has a different probability of correctness than using pressure sensors in the furniture.

**Resolution** denotes the granularity of information. This can for example mean spacial coverage. For instance, in a kitchen there can be hot-spots with high temperature (oven, cooker, toaster) which may not be picked up by temperature sensors in the room if spacial coverage is low.

**Up-to-dateness** specifies the age of context information. Also of interest is how likely the measurement is still accurately describing the present.

**Refresh rate** is related to up-to-dateness, and describes how often it is possible or desired to receive a new measurement.

QoC differs from QoS, because context information has quality metrics even when it is not provided as a service to any clients. In our middleware, context providers need to specify QoC attributes for the context information they deliver. These attributes may vary over time and therefore must be updated regularly.

## 3. STRUCTURE OF OUR MIDDLEWARE

Figure 1 shows the layers in context provision in our middleware (the adaptation part is not shown and is introduced only in Section 4).

At the bottom are *sensors* that deliver raw sensor data. These could be wireless sensor networks, ultrasonic badges for location, RFID tags for identification, video cameras for tracking, or others. These produce raw sensor data, often preprocessed for saving communication cost as much as the (often power-constrained) sensor

devices allow. These data are passed up one level in the framework to the *context providers*.

*Context providers* (CPs) are components (software or hardware) that aggregate and interpret the sensor data to produce some higher-level context, e.g. location, identity, type of activity, health condition of a person. As illustrated in the figure, more than one CP may access the same group of sensors. Similarly, it is possible for one CP to use sensor data from a group of sensors, as data redundancy often improves the total reliability of the derived context. This is rather more general than the typical approach in the Context Toolkit, where one sensor typically maps exactly to one context widget, and separate components called aggregators and interpreters collect and differentiate the sensor data. However, as an example, we could imagine the Context Toolkit being used to implement the context provision layer of the middleware. In any case, we image CPs to be often divided into sub-components with reusable capabilities.

Moving up one level, we have *context services* (CSs). These connect below to different context providers that provide the same type of context, e.g. location, but implemented using different underlying sensors, and above connect to the applications. They allow an application to use a type of context whilst abstracting from the actual instance of a context provider. An application can poll for context information or subscribe for notification when a certain condition on the context information or QoC values is met, e.g. when the temperature exceeds 26°C or when Alice enters the kitchen.

Certain CPs and applications may use context information from a CS to provide higher-level context. For instance, a CS may provide location in the form of 2D or 3D coordinates in the home. A CP may then use this information to determine the room in the home this location maps to. Again, this higher-level context would be delivered through a context service<sup>2</sup>.

CSs execute adaptation in the system. This means switching between available CPs, e.g. when the QoC of the currently used provider deteriorates excessively or the QoC of an unused but available provider improves greatly (*self-optimising*). It also handles automatic failure recovery (*self-healing*) when the currently used provider is unable to deliver context, e.g. because of excessive sensor failure, and automatic upgrade to newly installed sensors and context providers (*self-configuring*). Thus, our framework implements adaptation through the basic use of a *proxy* abstraction layer.

In the next section, we describe how this adaptation process is decided and executed.

## 4. ADAPTATION ENGINE

We start by describing service discovery in our middleware in Section 4.1. While service discovery is necessary in any service-oriented distributed environment, we tailor the service discovery protocol (SDP) to provide efficient monitoring of CPs for the purpose of adaptation. This includes rapidly discovering failure of CPs and updating the QoC of each CP. Section 4.2 mentions how adaptation is triggered, while Sections 4.3 to 4.5 explain how the adaptation engine determines the new CP to switch to.

### 4.1 Service discovery

To allow context services to find relevant context providers, a *directory service* (DS) is used. When CPs enter the network, they advertise their presence to the DS, supplying it with descriptive attributes for the service they provide, i.e. the context type and QoC

<sup>2</sup>Also in object oriented programming it is not unusual to always abstract an implementation through an interface, even when it is unlikely that there will be more than one implementation of the interface.

attributes. Then, a CP must regularly send a *heartbeat* to the DS, a short message that implicitly tells the DS that this CP is still alive and informs it of updated values for those QoC attributes that change over time. Thus, through the heartbeat, the DS is able to monitor the QoC of CPs and quickly detect CP failure, provided that the negotiated maximum time interval between heartbeats is kept short. We believe that this scales reasonable well within a smart-home if the heartbeat is limited to a simple message that can be processed quickly<sup>3</sup>. In comparison, Jini<sup>4</sup> has a recommended minimum leasing time of 120 seconds, and UPnP<sup>5</sup> a minimum recommended advertisement time of 30 minutes. These times would be inadequate for reacting and adapting promptly to failures.

The DS also takes care of initialising the CSs. When the first CP for a particular context type advertises itself to the DS, a corresponding CS is created. Conversely, when no more CPs are listed in the DS for a particular context, the CS terminates (possibly after notifying interested applications that this context is no longer available).

## 4.2 Triggering adaptation

As we already mentioned earlier, the context services *execute* adaptation by switching between different context providers. However, they do not decide adaptation, i.e. when to switch and which CP to switch to. Instead, adaptation is decided externally by an event-triggered adaptation engine (see Figure 2). The DS monitors the status of CPs and can fire events to the adaptation engine. Events that are of interest for adaptation purposes include notification of CP failure, that a new CP has appeared, or of changes to QoC in available CPs, in particular degradation of provider in use or improvement of unused but available provider. Given the event, the adaptation engine decides whether an adaptation should occur based on the current QoC of the available alternatives and, in the affirmative case, which CP the CS should switch to. Figure 2 shows some adaptation scenarios. We envisage the DS and the adaptation engine to share a database (or tuplespace) in which QoC values are stored by the DS and accessed for adaptation by the adaptation engine. The database can also be used to create a history of QoC and delivered context information. How the adaptation engine chooses the new CP to switch to is described in the next section.

## 4.3 Choosing the best CP

When an application is interested in acquiring context information from our middleware, it looks up in the DS the location in the network of the context service (CS) for the context type of interest. Then, it contacts the CS and informs it of the QoC it wishes to receive. This information is then relayed to the adaptation engine. When it is then time for an adaptation, the adaptation engine uses the applications' wishes to determine which CP is currently the best choice, i.e. which maximises the applications' "satisfaction" with the delivered context information.

It seems most convenient and flexible to define an application's satisfaction for a particular CP as a utility function that maps the CP's QoC attributes to a value that quantifies the application's satisfaction (where values  $\geq 0$  mean the application is satisfied with this CP and values  $< 0$  mean the application is not satisfied, e.g. the service the application provides to the user suffers from the bad

<sup>3</sup>In a deployment environment larger than a home, the centralised nature of the DS can certainly become a problem. Federating the DS, similarly to Jini, could solve this problem.

<sup>4</sup>Sun Microsystems' service discovery protocol built on Java. URL: <http://www.sun.com/software/jini/>

<sup>5</sup>Universal Plug and Play, a service discovery protocol based on web services. URL: <http://www.upnp.org>

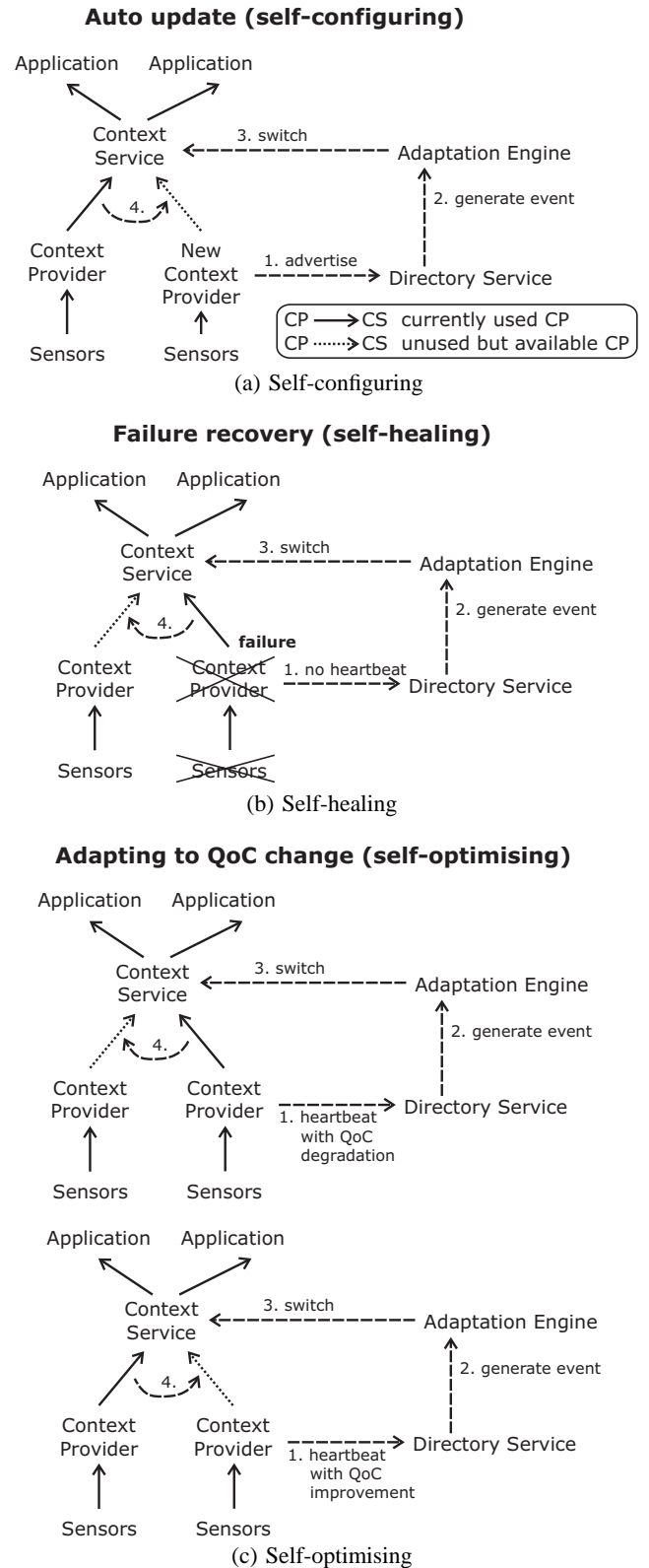


Figure 2: Adaptation scenarios.

QoC of this CP). An application can deliver its utility to the CS in the form of a function written, for instance, in a functional programming language such as Haskell, ML or Lisp.

Given each application's idea of CP utility, the adaptation engine can then apply each application's utility function to each CP and for every CP sum the applications' satisfaction value, thus obtaining for each CP the total perceived satisfaction.

The utility function could, for example, use linear distance (1-norm), Euclidean distance (2-norm), or return the maximum distance (max-norm) between a CP's provided QoC and an application's QoC expectation. However, as we are combining numerical distance values of different properties, e.g. location precision in metres with refresh rate in Hz, we need to define a standard scale for all dimensions. The simplest way to do this is to introduce a scaling factor for each QoC attribute. That is, if a particular context type has  $n$  QoC attributes and let the application's wishes for this context be  $\vec{a} = (a_1, \dots, a_n)$  (where we use  $a_i = \odot$  to indicate an application's indifference to the  $i$ -th QoC attribute), a CP's QoC  $\vec{c} = (c_1, \dots, c_n)$  (where we use  $c_i = \ominus$  to indicate a CP's inability to provide a quantitative value for the  $i$ -th QoC attribute) and the scaling factors for the QoC attributes  $\vec{s} = (s_1, \dots, s_n)$ , then we have:

Using  $\vec{d} = (\vec{c} - \vec{a}) \cdot \vec{s}$

$$|\vec{d}|_1 = |d_1| + |d_2| + \dots + |d_n| \quad (1\text{-norm})$$

$$\|\vec{d}\|_2 = \sqrt{|d_1|^2 + |d_2|^2 + \dots + |d_n|^2} \quad (2\text{-norm})$$

$$\|\vec{d}\|_\infty = \max\{|d_1|, |d_2|, \dots, |d_n|\} \quad (\text{max-norm})$$

where  $c_i - a_i = 0$  for  $a_i = \odot$  and  $c_i - a_i = o(a_i)$  for  $c_i = \ominus$ .

$o(\cdot)$  determines an application's satisfaction (or dissatisfaction) when the CP is unable to provide an estimate of a QoC attribute, given the value wished for by the application.

Actually, the various norms won't usually be applied directly as described above, but one also considers the sign (satisfaction (+) vs. dissatisfaction (-)) of each dimension and also of the final distance. For instance, it may only makes sense to add positive terms in the distance if there are no negative ones, i.e. extra quality in one dimension does not counteract the lack of quality in another dimension. In general, this definition implies that an application prefers it when it gets better QoC than it asked for, i.e.  $c_i - a_i > 0$  potentially contributes to an application's satisfaction. For instance, an application that finds objects for a forgetful user will work better as precision of location information improves. However, there are cases where an application does not need and cannot use better QoC than it asked for, e.g. location precision in the case of a light control system that turns on and off lights when an individuals enter and leave rooms. In that case,  $d_i$  should be redefined as

$$d_i = \begin{cases} (c_i - a_i) \cdot s_i & \text{if } c_i - a_i < 0 \\ 0 & \text{otherwise} \end{cases} \quad \forall i = 1, \dots, n$$

In other words, an application is satisfied (i.e.  $d_i = 0$ ) when the  $i$ -th QoC attribute is at least as good as it wishes it to be and the distance of this attribute can only affect the total distance when there is dissatisfaction (i.e.  $d_i < 0$ ).

Furthermore, up to now we have assumed that each QoC attribute is equally important. Since this is not generally the case, e.g. precision may be more important than refresh rate, applications may set weights  $\vec{w} = (w_1, \dots, w_n)$  to the  $n$  QoC attributes. Thus, we perform  $\vec{d} \leftarrow \vec{d} \cdot \vec{w}$  before computing the final  $p$ -norm distance (or any form of distance).

The utility functions mentioned above are already predefined in the middleware, so that an application on a resource-constrained device (e.g. a PDA or a mobile phone) need only select a predefined function and pass the appropriate input parameters instead of sending an entire utility function to define its QoC wishes.

One issue remaining is how to choose the scaling factor. In principle, we could determine the scaling factor by choosing a sensible scaling factor for each QoC attribute, based on an interval of likely values. However, this is tedious and subjective. We believe that a better way is to use a special case of the *Mahalanobis distance*<sup>6</sup>, where we measure for every QoC attribute the standard deviation over all available values from CPs, and then express application distances, i.e.  $\vec{c} - \vec{a}$ , as multiples of the standard deviation for that attribute. In other words, given the standard deviation vector  $\vec{\sigma}$ , where  $\sigma_i$  is the standard deviation of the  $i$ -th QoC attribute over all CPs, the scaling factor becomes  $\vec{s} = 1/\vec{\sigma}$ . This distance has the advantage of being independent of the scale used in the dimensions, i.e. using centimetres instead of metres to measure precision will not affect the numerical value of the distance. However, the range of values of the available CPs will affect the scaling factor of each QoC attribute.

Simple utility functions could set independent thresholds on a certain number of QoC attributes, e.g. a certain amount of precision with a certain refresh rate is desired. However, more complex utility functions could use dependence between QoC attributes. For instance, for location, an application may be interested in a certain amount of precision. However, if refresh rate is good, then lower precision is also acceptable. This can be modelled as a decision tree, which will be useful in the next section<sup>7</sup>. Also, applications can send updated utility functions at run-time, e.g. in response to context change or user input.

#### 4.4 Application weights and the role of authentication

Given each application's "perception" of the quality of the available CPs, it makes sense to introduce application priorities, such that an application's satisfaction will be weighed more heavily than others. For instance, when delivering location information in the smart home of an individual suffering from a heart condition, a health-monitoring application's satisfaction should weigh more than that of a light control system. To prevent all applications from selecting a high priority, the priority levels should be assigned by the middleware. One way to do this is for applications to authenticate themselves to the middleware to prove that they belong to a particular priority class (with predefined weight). This approach seems reasonable, since we want to anyway protect the user's context information, which can represent very sensitive data about a person's private life, from unrestricted access by malicious applications, and therefore application authentication is a necessary aspect of the middleware for access control to context information<sup>8</sup>.

<sup>6</sup>The standard Mahalanobis distance (also known as statistical distance) between two vectors  $\vec{x}$  and  $\vec{y}$  is  $\vec{d}_M(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^t S^{-1} (\vec{x} - \vec{y})}$ , where  $S^{-1}$  is the inverse of the covariance matrix (the variance, i.e. the squared standard deviation, of the variables is on the diagonal of the matrix). Compared to our special case, the use of the covariance matrix means that it also takes into account the correlation between variables.

<sup>7</sup>In a functional programming language, it can be implemented using boolean guards.

<sup>8</sup>Actually, protecting sensitive data also requires encryption in addition to authentication. The same infrastructure, e.g. PKI, used for authentication could also be used to establish symmetric keys for encryption, if necessary.

In fact, policies could determine the type of information that applications with different access rights have access to. Further, QoC attributes could be used to control the quality of data received, e.g. applications with lower access rights might only receive location information every hour.

#### 4.5 Learning an application's satisfaction

When deciding the best CP based on applications' QoC wishes and satisfaction of the available CPs, we would also like to consider the case where the application is unable or unwilling to provide a utility function to the middleware describing its notion of CP satisfaction. In this case, the middleware learns to predict whether an application will be satisfied with a particular CP.

Learning usually involves some form of feedback from the entity whose behaviour we wish to understand and predict. In our case, an application that does not provide its utility must however be able to tell the middleware whether a CP satisfies its "wishes"<sup>9</sup>. With this feedback, the middleware can build a decision tree that classifies CPs into satisfying and unsatisfying CPs. We believe that decision trees are sufficient to model effectively applications' QoC wishes and have the advantage of being simple to create, i.e. they can be created without much overhead. For instance, applications that set independent thresholds on the QoC attributes can be easily modelled (with a tree that has one node at every depth), but also applications that set conditional thresholds between attributes. Because the QoC attributes are not discrete, but continuous, the decision-tree learning algorithm must find split points, i.e. thresholds, for branching the tree on numerical attributes. Various approaches have been suggested to accurately and efficiently solve this problem [4].

Thus, in the event of an adaptation, the middleware uses decision trees to predict acceptance of the various CPs for applications that did not provide utility function and thus to choose the new CP to switch to. After the adaptation, the middleware can ask the application whether its prediction was correct. The reason for not asking the application in the first place instead of relying on error-prone prediction is a matter of performance: we want adaptation to occur rapidly, and therefore delays caused by communication with remote entities on the network become unacceptable, delays that can become very high for portable devices on low-bandwidth wireless links, e.g. Bluetooth. Also, if we assume that the applications use a utility model similar to a decision tree, the middleware can learn to predict accurately an application's acceptance of CPs fairly quickly. In particular, if we can assume that of all the available QoC attributes describing the context information, the application is likely to only use a subset of them in its decision, then using a relevance-based decision-tree learning (RBDTL) algorithm enables faster accurate predictions with fewer examples<sup>10</sup> [8]: it first identifies a minimal set of relevant attributes and then passes this set to the decision tree algorithm for learning.

It would seem that an extension to this could be to allow the middleware to learn the the application's utility function altogether, i.e. the numerical value of satisfaction for a particular CP. However, this typically requires more complex feedback, because now the application must return for a given CP its numerical satisfaction, in order to train the middleware's learning model. If the application can do this, however, it seems likely that it will have some utility

<sup>9</sup>We prefer the word "wish" to requirement because we believe that the application must be able (or at least try) to cope with context information that has lower quality than its specified requirements.

<sup>10</sup>RBDTL biases towards a minimal number of attributes to correctly predict the training data. When all available attributes are relevant to the decision, RBDTL shows no advantage.

function to compute the satisfaction value and could therefore just as well pass its utility function to the middleware in the first place, without requiring the middleware to learn it.

## 5. THE PROBLEM OF TRUST

An issue we wish to address next is that of trust. Up to now, we have assumed that all components external to the middleware "play along". In particular, the context providers provide accurate and reliable QoC values for the context information they deliver and expeditiously send heartbeats with updates when these values are outdated. Moreover, correct and fast adaptation relies on this assumption. However, we believe we must also investigate the case where this assumption is violated. For example, wireless sensor nodes of a particular manufacturer may advertise lower QoC than they're actually capable of, so that the middleware will not switch to them often as a source of raw sensor data for context information, thereby prolonging the battery lifetime of these sensor nodes, which may give them an unfair advantage compared to a competitor's sensor nodes (perhaps installed in the same middleware). Moreover, a non-malicious video camera's trustworthiness may depend on the amount of ambient light. That is, the CP may not be advertising wrong QoC attributes on purpose, it may simply be unable to estimate accurate QoC attributes because of current environmental conditions. This makes the problem all the more important.

To counter this problem, we introduce a special QoC attribute: trustworthiness (already introduced by Buchholz et al. in [2], on which we based our QoC attributes in Section 2.1). It determines how likely it is that the information provided is correct. This is similar to probability of correctness, however while probability of correctness is provided by the CP, trustworthiness is provided by an entity external to the CP. By introducing this property, we can select a CP based also on its trustworthiness. Notice that it is up to the applications to choose how much trustworthiness affects their utility function, i.e. how much risk they are willing to take in the hopes of receiving good QoC.

The problem that now arises is how we determine the value of trustworthiness. When we allow more than one CP to be in use at any time, the CS can analyse the context information from multiple sources to determine the most likely correct value and reduce the trustworthiness of CPs that deviates too much from the QoC they advertise (as far as this is possible). But, in the special case where only one CP is used at any time, this is not possible. In this case, it is up to the applications to affect a CP's trustworthiness. This can be done through *complaints* and *praises*. If an application receiving information from a CP can determine that the information is incorrect and deviates from the advertised utility, it sends a complaint to the CS indicating the received and expected values. The CS can then forward the complaint to the adaptation engine to adjust this CP's trustworthiness. Of course, the application needs to know the current CP's utility to be able to tell whether a complaint is necessary, or whether the current CP has bad QoC and advertises this correctly. This utility can be piggybacked by the CS on context delivery when necessary.

The application could use explicit user feedback to determine the correct context, or implicit user reaction to context-sensitive application adaptation. For example, if a PDA adapts its display to show a remote control for the TV because it received information that the user is located on the sofa in front of the TV, but the refrigerator senses that it is being opened by the user and notifies interested parties (including the PDA), and furthermore the user manually switches the PDA to display an inventory of the contents of the refrigerator, we can safely say that the location information

was inaccurate and that because of that, the PDA made an incorrect context-based adaptation.

The example shows that, while trustworthiness can be inferred to some degree externally to the CP, it is by no means easy to implement and can require the cooperation of multiple applications (when the refrigerator senses that it is being opened, this information contributes to validate a location CP's delivered information, even if the refrigerator itself is not interested in the user's location), and if possible also other CPs of the same context type for cross-validation. This however introduces another trust problem: can we trust the applications' and the other CPs' feedback?

We are currently trying to solve this problem using Bayesian parameter learning, where we feed application complaints/praises into a beta distribution to evaluate probabilistically a CP's trustworthiness. Details must be omitted for lack of space.

## 6. RELATED WORK

Our work was inspired by the Context Toolkit [3]. The Context Toolkit is a framework aimed at facilitating the development and deployment of context-aware applications. It abstracts context services, e.g. a location service, from the sensors that acquire the necessary data to deliver the service. Thus, in a different deployment environment, the provider of a context type can be exchanged with another provider that may use a fundamentally different type of sensors, yet the application does not need to be modified to access this different source of the same context. However, there is no mechanism that allows context services to adapt and react to failure or degradation of the underlying sensor infrastructure, e.g. by switching to an alternative means of acquiring the same type of context. In fact, the situation where multiple means of acquiring the same context may be dynamically present in the system is not considered. We started with this assumption and then addressed the issues of self-management and adaptation.

There have also been other middleware approaches to context-awareness middleware. For instance, [7] describes an infrastructure that provides context-awareness support to applications. It uses first order logic predicates to model contexts and allows deduction of higher-level contexts using rule-based approaches. Effectively, the application can delegate context-awareness to the infrastructure, which takes care of obtaining the context, evaluating the rules (provided by the application) and calling the actions. In contrast, in our solution applications delegate to the middleware the choice of most appropriate context provider.

As to adaptation in middleware, Bhatti and Knight [1] have proposed a QoS middleware for multimedia distribution in the Internet that also uses a utility function for determining the best alternative for delivering the media to the application, for instance choosing the most appropriate codec for audio delivery considering the current properties of the network, i.e. bandwidth and latency. However, the adaptation context of their middleware is quite different. QoS middleware typically address the issue of adapting the delivery of data across the network in response to changes in network properties. In our case, we are choosing the source of the data given various alternatives with varying quality (where quality is quantitatively an application-specific concept). Furthermore, our alternatives are not fixed, but are dynamically determined by the currently available context providers in the network and their (possibly dynamic) QoC attributes. This requires a service discovery protocol that reacts quickly to changes in the providers, so that the adaptation engine has an accurate view of the providers. Moreover, while Bhatti and Knight describe a fixed utility function for deciding adaptation, we use application-specific utility functions, as an application's QoC wishes are specific to the purpose and im-

plementation of the application, and therefore cannot be statically assumed in advance, as with network delivery properties, but must be either provided by the application or learned through application feedback. An interesting design aspect in their middleware is how they engineer their utility function to prevent state-flapping, i.e. the situation where the network QoS state is near a utility function's threshold, causing the middleware to continuously flap between two adaptation states.

## 7. CONCLUSIONS

Our goal was to introduce notions of autonomic computing in a middleware for context-aware applications. While there has been much research on autonomic computing, it has not been specifically applied to middleware for context-aware applications. Yet, particularly in a smart-home environment, self-management of the system is important because we cannot assume that there will be a technical administrator present round the clock to solve problems that may arise and fine-tune the system.

In our proposed solution we have assumed that CPs are able to estimate their QoC (or at least most of them). It remains to be seen in practice how well this assumption holds.

Although our middleware was designed in the context of smart-home application scenarios and context-awareness, it can be generalised to the case where there are multiple service providers for the same service, e.g. printing services, and based on the applications' wishes we choose the optimal service provider for each application, e.g. nearest printer, best-quality printer. Then, if the currently used service (e.g. a printer) should fail (e.g. paper jam, no toner) the middleware can automatically switch to the best available alternative. In general, we believe utility functions to be better suited to adaptive service provision than query-based service discovery, as they rank all available alternatives by their utility and allow the middleware to autonomously determine the best alternative on current service failure.

## 8. REFERENCES

- [1] S. N. Bhatti and G. Knight. Enabling QoS adaptation decisions for Internet applications. *Computer Networks*, 31(7):669–692, 1999.
- [2] T. Buchholz, A. Küpper, and M. Schiffers. Quality of context: What it is and why we need it. In *Proceedings of the Workshop of the HP OpenView University Association 2003 (HPOVUA 2003)*, Geneva, 2003.
- [3] A. K. Dey and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16:97–166, 2001.
- [4] E. Frank and I. H. Witten. Selecting multiway splits in decision trees.
- [5] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer, IEEE*, 34(8):57–66, Aug. 2001.
- [6] S. Meyer and A. Rakotonirainy. A survey of research on context-aware homes. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 159–168. Australian Computer Society, Inc., 2003.
- [7] A. Ranganathan and R. H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput.*, 7(6):353–364, 2003.
- [8] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.