An adaptive middleware framework for context-aware applications

Markus C. Huebscher and Julie A. McCann

This is a non springer version of the paper. For the full paper with equations and diagrams please see original published version : Received: 10 July 2004 / Accepted: 17 November 2004 / Published online: 19 August 2005_ Springer-Verlag London Limited 2005 or go to: http://www.doc.ic.ac.uk/~jamm/papers/puc05.pdf

Abstract
We describe a middleware framework for the adaptive delivery of context information to context aware applications. The framework abstracts the applications from the sensors that provide context. Further applications define utility functions on the quality of context attributes that describe the context providers. Then, given multiple alternatives for providing the same type of context, the middleware applies the utility function to each alternative and choose the one with maximum utility. By allowing applications to delegate the selection of context source to the middleware, our middleware can implement autonomic properties, such as self-configuration when new context providers appear and resilience to failures of context providers.

Keywords Utility functions, Context-awareness, Middleware

1 Introduction

In pervasive computing, one of the key features that distinguish pervasive applications from non-pervasive ones is context-awareness. Context-aware applications are aware of some aspect of the user, e.g. her location and activity, and of the physical environment in which the user is located, e.g. humidity and temperature, amount of ambient light and sound. Physical spaces that have been enabled with an infrastructure for pervasive applications are often called active spaces [10]. An example of active spaces with much ongoing research is that of smart-homes, where the aim is often to support people with health problems, such as elderly living alone at home, or to improve general lifestyle quality, e.g. a smart fridge that monitors its contents and warns the user of the products that are expiring or finishing. Supporting elderly people at home is a particularly important issue, as the ratio of people aged over 60 is increasing in many countries and nursing homes will not be able to cope with projected numbers of elderly people in the near future. A solution to this is to promote an independent lifestyle in private homes, and at the same time enrich the home with technologies that support these people. This includes recognising a crisis situation, supporting everyday activities and providing awareness of daily life and long-term trends [9]. This example illustrates the potential for pervasive computing applications, and also the importance of context information in these applications.

As already mentioned, context in pervasive computing refers to information about the user and her environment, e.g. user location and activity, environment noise level and ambient light. These different types of context information can each be determined in a variety of different ways. For instance, an office building may be ‘‘augmented’’ to continuously determine the location of the employees. This can be done using ultrasonic badges [5], RFID-tags with readers at the doors [13], Infrared badges that periodically emit unique IR pulses [12], or by other means. A variety of applications can then make use of the knowledge of the location of its users, e.g. by tracking colleagues [13] or teleporting one’s virtual desktop to the PC in front of us [5], wherever we may be. Sensors are extremely important to context-aware applications, as much of the context information involved in pervasive systems is derived from sensors [6]. While certain types of context information are static (e.g. a person’s birthdate), others are dynamic. Among these, the persistence of dynamic context information can be highly variable. For example, relationships between office colleagues typically endure for months or years, while a person’s location and activity often change from 1 min to the next. It is in these highly variable activities that sensors play a

fundamental role. Furthermore, sensors allow context sensing to be unobtrusive, not requiring the user to explicitly input context information [8].

1.1 Our approach to context delivery in a nutshell In our middleware framework, we consider situations where multiple sources of context information (i.e. types of sensors) may be available for the same type of context and introduce an abstraction layer, the context service, that hides a particular context provider from the applications and chooses for each application requesting context information one provider of context that is most appropriate among the currently available ones. The context service can then switch providers on behalf of the application, for example, when a new provider is introduced in the system at run-time, when a provider fails or when the quality of the currently used provider degrades (or vice versa when the quality of an available but unused provider improves greatly). In our middleware, we consider the notion of ‘‘context provider quality’’ to be application specific. Thus, when an application first contacts the middleware, it specifies its notion of quality in the form of a utility function. This function takes as input quality of context (QoC) attributes, which describe each provider—e.g. precision and refresh rate for location— and delivers a number that quantifies an application’s satisfaction with a particular context provider. Then, the middleware can apply the utility function to each provider and select at any time the provider with maximum utility. Adaptation arises in cases where a new provider appears, an existing one fails, or when dynamic QoC attribute values of a context provider change over time (Sect. 3.2 illustrates examples of adaptation).

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 describes the structure of our middleware, focusing on context acquisition from sensors and delivery to applications. Section 3 describes middleware operation, including the use of utility functions for context provider selection and adaptation. Section 4 looks at the issue of trust in the context providers. We then look at important assumptions we make in our middleware design in Sect. 5, look at a related project in Sect. 6 and conclude with Sect. 7.

2 Middleware structure

Figure 1 shows the structure of context-delivery in our middleware framework. The figure does not show the adaptation components of the middleware, which are introduced in Sect. 3. In this section, we describe each layer in Fig. 1. The bottom layer consists of sensors. These are physical devices, such as wireless sensor networks, ultrasonic badges for location, RFID tags for identification, video cameras for tracking, etc., that produce raw sensor data. In the next layer, sensor logic aggregates and interprets these low-level data to produce context types as services. This is done in what we call context providers (CPs). While CPs could be written specifically for a particular type of sensors or set of sensors, we believe it should be possible to write CPs for a particular type of context that separate data analysis logic from sensor access logic to produce CPs that can be used with different types of sensors. This might, for example, be achieved if we were to integrate the Context Toolkit [3] (which will be briefly described in Sect. 5) as the context provision layer in our middleware. CPs provide context information as a service to the applications. However, we hide a particular CP from an application by introducing an extra layer of abstraction, the context service (CS). Context services retrieve context information from the CPs on behalf of the applications and deliver this information to the applications. This abstraction layer is useful because, should a different CP be better for an application than the currently used one, the CS can autonomously adapt by switching to the better alternative without having to involve the application. In the following section, we describe how the need for an adaptation is determined and the adaptation executed.

3 Middleware operation

In this section, we discuss the middleware functions. We start by describing how CPs and applications communicate with the middleware when they first enter the network. 3.1 First contact When a CP enters the network, it advertises itself to a directory service (DS) (see also Fig. 2a). This component in the middleware keeps track of all CPs that have registered themselves with it and are available as a source of context information for the applications. On first contact, the CP informs the DS of the context type it can deliver, and attaches QoC attribute values describing this provision. This set of QoC attributes is predefined

Fig. 1 Layers of context provision in our middleware

If the advertisement is successful, the DS and CP negotiate a maximum heartbeat time. Then, once registration is successful, a CP must regularly send a heartbeat to the DS within the maximum heartbeat time for the CP's entry to remain alive in the DS's registry. The heartbeat is a short message that implicitly informs the DS that this CP is still alive, and more importantly is used by the CP to send updated values for those QoC attributes that change over time.

We now move on to the applications. On first contact, an application connects to the DS requesting a particular type of context. If this type of context is available, the application delivers a utility function to the DS. This utility function takes as input QoC attributes (predefined for this type of context) and outputs a number that quantifies the application's satisfaction with a CP, whereby an increasingly positive value signifies increasing satisfaction (and analogously for negative values). Thus, the application delegates to the middleware the task of rating the various alternatives for this type of context, allowing the middleware to select the best alternative for the application, where best is application-specific and determined by the utility function. After receiving the utility function, the DS points the application to the CS that delivers this type of context information. The application can then poll for data or subscribe to event notification.

3.2 Delivery and adaptation

The delivery of context information is handled by one CS for each context type. A CS is initialised by the DS when the first CP for a particular context type is registered in the DS. Conversely, a CS is destroyed when there are no more CPs registered in the DS for the corresponding type of context. In this case, it may be necessary to notify applications that this context type is no longer available.

While a CS delivers context information to the application, it does not decide which CP to use. This decision is made by an adaptation engine which accesses the DS's database of current QoC values of each CP. Using this information, the adaptation engine can take an application's utility function, apply it to each CP and then pick the CP that maps to the highest utility value1. It then informs the corresponding CS accordingly.

While the adaptation engine performs the task of selecting the most appropriate CP for an application, the re-evaluation of utility functions in the adaptation engine is triggered by the DS. This will usually happen when the DS receives particular heartbeats from the CP, or when a lack of heartbeats indicates that a CP is not keeping its advertisement alive in the DS, due to its failure. Figure 2 illustrates various cases where a heartbeat (or lack thereof) in the DS triggers a re-evaluation of the utility functions in the adaptation engine and a possible switch Fig. 2 Adaptation scenarios to another CP in the corresponding CS. 1In the event of more than one CP with maximum utility, the DS picks one at random.

14 3.3 An example utility function

In this section, we describe an example utility function that uses Euclidean distance in the QoC space for rating the satisfaction of an application with providers of location information.

For simplicity in the description of the example, let us assume that the QoC space is defined by just two dimensions, precision and refresh rate of a location service. Without loss of generality, we can also define

that for every dimension a bigger number is better. Thus, we do not measure precision in metres, as a precision of 5 m is not as good as a precision of 1 m, but instead take the inverse of precision.

Now, our example utility function defines a reference point in the QoC space that represents desirable values for the QoC attributes in a CP. Consequently, this point defines an area in the QoC space where every point represents a CP whose QoC attributes are at least as good as the reference point (see shaded area in Fig. 3a). In other words, any point in this area (or hyperspace for the general n-dimensional case) will satisfy the application and will be preferred to points elsewhere in the QoC space.

If the application does not need better QoC than the reference point, it can choose any CP in this area at random. This can be achieved by setting the same satisfaction value for all CPs in the shaded area, whereby the middleware will then pick one at random. However, if the application wants the best available CP, then the utility function can return the Euclidean distance between the reference point and each CP's point in the shaded area. As a result, the middleware will pick the best alternative with respect to the reference point (CP2 in Fig. 3a).

Actually, an application may value one dimension more than another. For example, a frequently-lost-objects finder application for a smart-home will value precision more than refresh rate, as determining the location of the lost object with best possible precision is the top priority, while refresh rate is of relatively little importance, as a lost object in the home is unlikely to move much. Instead, an automatic light control system will value refresh rate more than precision, since the lights should turn on in a room the moment a user approaches an entrance to that room, and not seconds after the user has entered the dark room. Yet, the precise location of the user at the time of measurement is not that important. Applications may therefore introduce a weight for each dimension before computing the Euclidean norm2. This example also illustrates why utility functions and the notion of CP quality are application specific.

Now, we still need to consider the case where we only have CPs that do not entirely satisfy the QoC attributes of the reference point (Fig. 3b). Again, there are various sensible approaches, but a possible option is to pick, among these unsatisfying CPs, the one that minimises the distance from the hyperspace of satisfying CPs. This is equivalent to saying that we ignore how good the good QoC attributes are (mathematically, we set their distance in the distance vector to 0), and look only at the attributes that do not satisfy the reference point; we then minimise the Euclidean distance of only these attributes from the reference point. For instance, in Fig. 3b, the distance of CP1 is determined only by refresh rate (since precision satisfies the reference point), for CP2 the distance is given by both attributes, and in the case of CP3 the distance is determined by precision. The final utility values are made negative; so, when the middleware chooses the alternative with highest utility, it will pick the CP with minimal negative distance (CP1 in Fig. 3b).

The utility function mentioned above can already be predefined in the middleware as a parametrised function, so that an application on a resource-constrained device (e.g. a PDA or a mobile phone) only needs to select a predefined function and pass the appropriate parameter values instead of sending an entire utility function to define its QoC wishes.

3.4 Scaling the different dimensions

There's still a problem though we need to address. When computing the Euclidean distance, we are summing together the numbers that come from different measurement units, for example 1/m for precision with Hz from refresh rate. But, because the range of common values can be very different between dimensions, each dimension can influence very differently the final Euclidean distance in the QoC space. For instance, if we measure precision in 1/ m, we will get larger numbers than using 1/mm and therefore precision will weigh more on the utility value.

A possible solution would be to manually pick scaling factors for each dimension by looking at the likely range of values of each dimension, and then scaling these ranges to a common range, for example [0, 1]. However, this is tedious and, worse, subjective. A better solution is to use a simplified form of the Mahalanobis distance, where for each dimension (each QoC attribute), we compute the standard deviation over all available values of this QoC attribute from CPs, in fact preferably over all values that the

middleware has ever observed, which can be done incrementally with little constant storage. Then, we express the components of the distance between the CP point and the reference point as multiples of the standard deviation for each component. This basically means that we apply a scaling factor of 1/ri to the i-th component of the distance vector, where r is the standard deviation vector for all dimensions in the QoC space.

3.5 Learning an application's satisfaction

We have extended the middleware to consider the situation where an application does not send a utility function. Indeed, it may be unwilling or incapable of doing so, for example, an application running on a portable and resource-constrained device that needs to minimise the amount of wireless communication to preserve battery power. In such a case, the middleware can still try to predict whether the consumer will be satisfied with a particular CP and therefore make a good choice on behalf of the consumer, as opposed to simply choosing one blindly at random.

So, effectively, the middleware can learn to understand whether an application that does not provide a utility function will be satisfied with a particular CP or not, i.e. the predicted decision is only a binary yes/no. Since we want to learn something, there has to be some input into the learning model from the application. But, since the application did not send a utility function in the first place, the feedback from the application must be as simple as possible. Keeping this in mind, we propose the following strategy. At the very beginning, the middleware has no specific information about the application and selects a CP that is hopefully satisfying, e.g. by looking at the CP that has been selected most by utility functions of other applications. Then, when the CS sends context information to the application (which includes attached QoC attribute values), the application is requested to send a binary feedback, a positive or negative acknowledgement as to whether the CP is satisfying. This feedback can be sent at the application's leisure at any time after receiving the context information, for example, piggybacked onto the next message sent to the DS. Given this input, the DS must now learn to predict whether the consumer will be satisfied with a given CP.

Because one of our major goals is real-time adaptation, we cannot afford to use artificial intelligence learning techniques with high complexity. Indeed, the simpler the learning technique, the less performance loss we can expect from the DS. As a consequence of this point, we believe relevance based decision tree learning models (RBDTL) to be appropriate, as decisions tree learning algorithms are among the simplest, and yet most successful forms of learning algorithms [11]. Decision tree learning builds a classification tree that is traversed from the root down and, at every internal node, a child is selected according the value of some property. The decision is then determined by the value of the leaf node reached in the tree. In our case, the properties in the internal nodes are the QoC attributes, and the decision is yes/no (for a CP's satisfiability).

Because most 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 at the internal nodes. Various approaches have been suggested to accurately and efficiently solve this problem [4]. ''Relevance based'' means that the learning model first identifies those properties (i.e. QoC attributes) which affect the decision at all (which may be only a small subset of all available properties) and then uses only these to construct the classification tree.

Relevance based decision tree learning models performs reasonably well if the application's satisfaction can be represented as a classification tree, and particularly well if only a small subset of the QoC attributes affect the decision. For instance, deciding whether a CP's point in the QoC space is contained within the hyperspace spanned by a reference point, as in the utility function example in Sect. 3.3, can be easily modelled with a classification tree. However, if the application's satisfaction cannot be modelled as a classification tree, an unpredictable tree may be produced that is continually updated at every application feedback. Further, note that we assume that the application's model of satisfiability is static. Tree-learning models don't work well in the dynamic case, as the current tree is continuously updated and can become more and more complex.
It might seem much better to directly ask the application which available CP it prefers. However, because some of the QoC attributes are dynamic, we would need to ask the application again and again whenever

an attribute of a CP changes. This again requires a lot of communication, which is undesirable. Our learning extension is thus a compromise between performance requirements in the middleware and the desire not to require applications to send a utility function to the middleware.

4 Addressing the trustworthiness of CPs

After establishing the design and operation of our middleware framework, we tried to address the important issue of trust. Our middleware selects the most appropriate CP for an application by applying the application's utility function on the QoC attributes advertised by each CP. However, the selection will only truly be as intended if the advertised QoC attributes are reliable, and therefore the correct operation of our middleware relies on accurate QoC values by the CPs. This means that, not only must a CP determine accurate QoC attribute values, it must also send them expeditiously to the directory service in a heartbeat, to prevent the directory service from using outdated QoC values in a utility function.

However, in practice, this is not a reasonable assumption. Indeed, a CP may be unable, perhaps only for a temporary amount of time, to deliver accurate QoC values. As an example, let us consider a CP that uses a video camera to determine the gait of a person, i.e. whether the person is standing, walking, sitting or lying. This type of context information is of paramount importance in a smart home that supports the elderly and must recognise a crisis situation. Now, if the probability of correctness (poc) QoC value depends on the amount of ambient light in the room, the video camera can use a light metre to estimate its poc. However, if the poc decreases when there is low contrast between the background colour and the colour of the person's clothes, and the video camera is unable to detect this aspect, then the reliability of the advertised poc will drop, at least until the person moves to another spot where the background colour is different, or where different lighting conditions increase this contrast, or of course until the person changes her clothes.

To allow the applications to take into account, the reliability of a CP's advertised QoC values, we introduce an additional QoC attribute: trustworthiness. This is an aspect of context that has already been introduced by Buchholz et al. in [1], on which we have based our own QoC attributes. Trustworthiness informs us how likely it is that the remaining QoC information provided by the CP is correct. This must not be confused with probability of correctness, which gives the probability that the actual context information delivered is correct. Further, unlike other QoC attributes, trustworthiness is not provided by the CP itself, but is determined by an entity external to the CP, in our case the adaptation engine (the cost of updating a CP's trustworthiness incrementally on receiving feedback is small in our approach). 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.

We now describe our approach to estimate a CP's trustworthiness. We request that applications that are capable of judging the quality of the context information they receive against the advertised QoC send a complaint or praise feedback to the CS accordingly. The CS then forwards the feedback to the adaptation engine to adjust this CP's trustworthiness. In some cases, an application could use explicit user feedback to determine the correct context, or implicit user reaction to context-sensitive application adaptation. The following example shows a context-based adaptation of a graphical user interface (GUI) which involves the user without explicit user interaction for context quality feedback.

Suppose a smart home inhabitant wears a ‘‘smarthome’’ remote control that possesses a display that allows the user to control various appliances and aspects of the home. Further, when not currently in use, the display continuously adapts in the background to the current location of the user. Now, consider the case where the remote control adapts its GUI 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 this same user and notifies interested parties (including the remote control). Furthermore, the user manually switches the remote control's GUI to display an inventory of the contents of the refrigerator. In such a case, putting all this information together, we can safely say that the location information was highly inaccurate and that because of that, the remote control made an incorrect context-based adaptation. However, ‘‘putting all this information together’’ is not necessarily trivial: when the refrigerator senses that it is being opened, this information contributes to validating a location CP's

delivered information, even if the refrigerator itself is not interested in the user's location. Thus, while useful information may be directly or indirectly acquired from the user to determine the trustworthiness of CPs, it potentially requires the cooperation of all smart appliances in a smart space and is often not trivial. Consequently, utility functions must also cope with the case where no trustworthiness information is available (in which case a special default value for trustworthiness is used as input in the utility function).

Moreover, using application feedback to determine trustworthiness introduces an additional trust problem: we must determine how far we can trust the feedback of applications. We have come up with a possible solution to solving these problems using Bayesian parameter learning, where we feed application complaints/praises into a beta distribution to evaluate probabilistically a CP's trustworthiness. We also use majority voting of multiple applications' feedback to estimate the trustworthiness of each application. Details must be omitted for lack of space, but may be presented in a later paper.

In the ongoing work, we are trying to see if we can use Bayesian networks as a means to combine context information from all available providers, while taking into account the trustworthiness of each provider, and outputting a final context value for the application with its own trustworthiness. We are currently looking at a general approach that is applicable to simple types of contexts with discrete values, such as gait (e.g. walking, standing, sitting, lying) or location as room ID. For complex contexts with continuous values, it is likely that a very context-specific approach is necessary, e.g. [7] for position location.

5 Important assumptions in our approach

For our middleware framework to work, we make a series of fundamental assumptions, which we describe here.

Firstly, we assume that it is possible to abstract context delivery from the sensors and sensor logic that derive the context from raw sensor data acquired by monitoring the user and her environment. This means that it should be possible for each type of context to define a standard API for applications to access the context information while keeping clear semantics for the information delivered (i.e. what does the information exactly mean). This is an assumption that most pervasive middleware appear to make. A seminal project in this area was the Context Toolkit [3], which was influential to our own framework. The

Context Toolkit is a framework aimed at facilitating the development and deployment of contextaware applications. It builds upon the very assumption we have just mentioned: it abstracts context services, e.g. a location service, from the sensors that acquire the necessary data to deliver the service. Thus, an application has a standard API to access the context data, and may be deployed in a different environment, where another provider of the same context type is used, possibly based on very different sensors. Yet, no change to the application code is needed to accommodate the new type of context provision. Our second assumption is that, in a smart space, we will be likely to find different context providers for the same type of context. For example, there are many ways of locating a person indoors. In a smart home, there may be one method, such as the active bats [5], that is specifically designed for location. But if that method fails, for example, the batteries on the active bat die out, then there may be many other types of sensors that were not specifically designed for location but can nevertheless deliver location information with varying degrees of quality (where the notion of quality is application specific). In the Context Toolkit, 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.

Our third, and perhaps the most controversial assumption, is the use of QoC attributes as input into the utility functions. The utility functions are predefined by the applications, and each application sends or selects a utility function without prior knowledge about the alternatives available. This means that it must be possible, for each type of context, to define a set of descriptive attributes that satisfactorily describes the quality of the context information delivered by a provider, i.e. the descriptive attributes do not just

describe the information delivered, but describe the capabilities of the provider as well. Further, for each type of context, all parties—i.e. context provider developers and application developers—must agree on a standardised set of descriptive attributes. This set would be immutable, at least until a new version of the standard would be released, which would require applications and context providers to migrate to the new standard. However, a new standard would be to some extent backward-compatible: old utility functions would simply not use QoC attributes defined in the new standard, and new utility functions would have to set a default value (of their choice) for QoC attributes that a provider is not able to deliver, either because it does not implement the new standard or is simply incapable (perhaps temporarily) of delivering this particular QoC attribute. It is arguable however, whether a utility function could still adequately serve its purpose if some of the QoC attributes it uses are not supported by the context provider and some of the provider's QoC attributes are not supported by the application's utility function. This makes backward compatibility a delicate matter when a new version of the standard is released. Therefore, for our middleware approach, it is preferable to assume that a standard set of QoC attributes can be decided and standardised a priori for each type of context, and no major changes are likely to occur in new releases of the QoC attributes standards.

Our third assumption is not too uncommon in pervasive computing, though. A related example, if not exactly in the same context, is Universal Plug and Play (UPnP) [14], a service discovery and access protocol. 18 The UPnP Forum is also working at creating standard ''templates'' for how devices and services in the home are described and accessed. While vendor-specific information is allowed, devices must first of all comply with a standard template that defines a set of so-called ''state-variables'', which are means for applications and services to exchange information. The range of allowed values for each state-variable is defined in the standard template. UPnP is mainly an industry-effort, and this shows that the industry is interested in standardising such things as the description of a temperature sensor. However, much work still remains to be done in service description ontologies and also the semantic web, which is related to this issue.

Finally, our entire framework only makes sense if there is full trust in the middleware. Specifically, the CPs must trust that the DS and adaptation engine deal correctly with the QoC attributes that each provider delivers to the middleware. Also, the applications must trust that the adaptation engine applies their utility functions correctly and deals correctly with their feedback to compute each CP's trustworthiness. Trusted computing technologies, such as TCM4, may be able to give applications and CPs the assurances they need.

6 Related work

Cohen et al. [2] have proposed iQueue, a data-composition framework for pervasive data. iQueue allows applications to create data composers, specify a composer's data sources using functional data specification, and specify a composer's computation through application specific code (in Java) or a library of built-in primitives. Their framework is implemented in Java and applications must be implemented in Java as well, as they are tightly coupled with the composers, which are the central elements of the iQueue programming model. Once a composer is activated, the iQueue run-time system selects data sources satisfying the data specifications, dynamically reselects data sources as appropriate, mediates between diverse data formats, and manages network placement of composers. As a result, the iQueue system enables applications to focus on the semantics of composition by facilitating the mechanics of composition. For instance, a sensor failure is automatically detected and rebinding initiated when the sensor's previous service announcement expires without a new one having arrived, a common failure detection scheme used in service discovery protocols. The goal is very similar to ours, although our approach is somewhat different.

They use a mechanism similar to our heartbeat, in that a data source issues advertisements periodically, but also whenever properties of the data source, e.g. quality of information, change. Based on this information, a binding manager decides either to maintain its current binding or to rebind to some other data source (its function is similar to our adaptation engine). However, they don't describe in detail how this decision is made, although it would appear that they use Boolean predicates over the values of the properties of the data source. Instead, we present a mathematical model based on applications' wishes that evaluates each application's quantitative satisfaction with regard to any particular data source.

Service discovery is also somewhat similar to our work. A data resolver receives advertisements from member data sources. The binding manager registers with the data resolver to receive data-source-change notifications from the data resolver, issued when a new advertisement for the data source reports a change in the value of some property. They mention the need for rapid adaptation, but need to trade-off rapid change and failure-detection for scalability. They plan to federate data resolvers using a Gryphon, a high-performance, scalable wide-area content routing (publish/subscribe) network, to replicate advertisements. As future work, they plan to include external data sources such as databases, news feeds and SOAP web services.

7 Conclusions

We have proposed an adaptive middleware framework for the provision of context information for context aware applications. In particular, context provision adapts to changes in the quality of information advertised by the providers, to new providers entering the network and present ones failing.

We realise that we have not addressed many issues in context delivery, such as the architecture and structure of the CPs, the semantics of context and context composition, but focused on adaptation of information delivery in the presence of alternatives. Thus, we have proposed a generic framework which we hope could be integrated as a context delivery component into a more holistic middleware for pervasive applications, as certain basic techniques, such as service advertisements using heartbeats, are becoming a recurring pattern in distributed service-oriented architectures, particularly in pervasive computing.

We would like to conclude by suggesting that the use of utility functions for service selection can be advantageous also for other types of services. For instance, consider the case of printing services. A user might wish to print to the printer with shortest printing time (within a certain distance of the user), which would depend on the current print queue length and printer speed (dynamic and static descriptive attributes of the printer, respectively). With a utility function, the middleware could redirect the print job to another printer if, while it is enqueued at a printer, that printer suffers a paper jam or the toner finishes. Therefore, suddenly another printer would give the fastest printing time, and the middleware could use the utility function to automatically select the new printer to switch to. In general, we believe that utility functions are a good solution for specifying service selection and enabling automatic service adaptation whenever the alternatives can be ranked. This way, if the best service fails, or gets worse, the middleware can autonomously switch to the new best alternative on behalf of the application.

References

1. Buchholz T, Ku¨ pper A, Schiffers M (2003) 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
2. Cohen NH, Purakayastha A, Wong L, Yeh DL (2002) iQueue: a pervasive data composition framework. In: Proceedings of the third international conference on mobile data management (MDM), pp 146–153, 8–11 January 2002
3. Dey AK, Abowd GD (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum Comput Interact 16:97–166
4. Frank E, Witten IH. Selecting multiway splits in decision trees
5. Harter A, Hopper A, Steggles P, Ward A, Webster P (2002) The anatomy of a context-aware application. Wireless Netw 8(2–3):187–197
6. Henricksen K, Indulska J, Rakotonirainy A (2002) Modeling context information in pervasive computing systems. In: Proceedings of the first international conference on pervasive computing. Springer, Berlin Heidelberg New York, pp 167–180
7. Hightower J, Fox D, Borriello G (2003) The location stack. Technical report, University of Washington
8. Meyer S, Rakotonirainy A (2003) A survey of research on context-aware homes. In: Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003. Australian Computer Society, Inc., pp 159–168
9. Mynatt ED, Essa I, Rogers W (2000) Increasing the opportunities for aging in place. In: Proceedings on the 2000 conference on Universal Usability. ACM Press, New York, pp 65–71
10. Roman M, Hess C, Cerqueira R, Ranganathan A, Campbell R, Nahrstedt K (2002) A middleware infrastructure for active spaces. Pervasive Comput IEEE 1(4):74–83
11. Russel S, Norvig P (2003) Artificial intelligence: a modern approach, 2nd edn. Prentice Hall, New Jersey
12. Stanford V (2002) Using pervasive computing to deliver elder care. IEEE Pervasive Comput 1(1):10–13
13. Trumler W, Bagci F, Petzold J, Ungerer T (2003) Smart doorplate. Personal Ubiquitous Comput 7(3–4):221–226
14. UPnP Forum (2003) UPnP device architecture 1.0.1, 6 May 2003 20