Middleware for Distributed Context-Aware Systems*

Karen Henricksen[1], Jadwiga Indulska[2], Ted McFadden[1], and Sasitharan Balasubramaniam[2]

[1] CRC for Enterprise Distributed Systems Technology (DSTC)
karen@itee.uq.edu.au, mcfadden@dstc.edu.au
[2] School of Information Technology and Electrical Engineering, The University of Queensland
jaga@itee.uq.edu.au, sasib@tssg.org

Abstract. Context-aware systems represent extremely complex and heterogeneous distributed systems, composed of sensors, actuators, application components, and a variety of context processing components that manage the flow of context information between the sensors/actuators and applications. The need for middleware to seamlessly bind these components together is well recognised. Numerous attempts to build middleware or infrastructure for context-aware systems have been made, but these have provided only partial solutions; for instance, most have not adequately addressed issues such as mobility, fault tolerance or privacy. One of the goals of this paper is to provide an analysis of the requirements of a middleware for context-aware systems, drawing from both traditional distributed system goals and our experiences with developing context-aware applications. The paper also provides a critical review of several middleware solutions, followed by a comprehensive discussion of our own PACE middleware. Finally, it provides a comparison of our solution with the previous work, highlighting both the advantages of our middleware and important topics for future research.

1 Introduction

The proliferation of standalone and embedded computing devices in our work and home environments, combined with a variety of networking technologies, increases the importance of context-awareness in distributed applications. Context-aware applications adapt to changes in the environment and user requirements. This dynamic adaptation provides the degree of autonomy needed to free users from the current computer-centric model of human-computer interaction. For example, sensor-based "smart home" applications can unobtrusively support elderly people in everyday tasks, such as remembering to take medications or providing early detection of behavioural changes.

* The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training).

The complexity of developing context-aware applications makes middleware an essential requirement. The middleware solutions proposed so far for context-aware systems address basic issues traditionally addressed by middleware for distributed systems, including paradigms for coordination and communication between distributed components. They also offer support for gathering and managing context information, in order to simplify application development and promote sharing of context information and context sensing components. However, many additional requirements are not met. For instance, most solutions do not adequately support the deployment and configuration of new components, the dynamic reconfiguration of components, or user privacy.

In this paper, we evaluate the current state-of-the-art in middleware for distributed context-aware applications, including the middleware developed in our PACE (Pervasive, Autonomic, Context-aware Environments) project. Based on the evaluation, we also highlight a set of open research problems in this area.

The structure of the paper is as follows. In Sections 2 and 3, we characterise context-aware systems and introduce a set of requirements for middleware for these systems. In Section 4, we review a set of middleware solutions and analyse them with respect to the requirements. In Sections 5 and 6, we introduce our PACE middleware and demonstrate how the middleware is used to support the development of a context-aware vertical handover application. Finally, in Sections 7 and 8, we provide an analysis of our solution, a discussion of open research challenges, and a summary of the contributions of this paper.

2 Characteristics of Context-Aware Systems

Context-aware systems consist of a variety of distributed components. Early systems were relatively simple, and were often constructed simply as distributed application components communicating directly with local or remote sensors. Today, it is widely acknowledged that additional infrastructural components are desirable, in order to reduce the complexity of context-aware applications, improve maintainability, and promote reuse. Figure 1 illustrates the distributed components that can be found in many current context-aware systems. In addition to application components, sensors and actuators, shown at the two extremities in this layered model, these systems include:

- components that (i) assist with processing sensor outputs to produce context information that can be used by applications and (ii) map update operations on the higher-order information back down to actions on actuators (layer 1);
- context repositories that provide persistent storage of context information and advanced query facilities (layer 2); and
- decision support tools that help applications to select appropriate actions and adaptations based on the available context information (layer 3).

Programming toolkits are often also incorporated at the application layer (layer 4) to support the interactions of the application components with other components of the context-aware system.

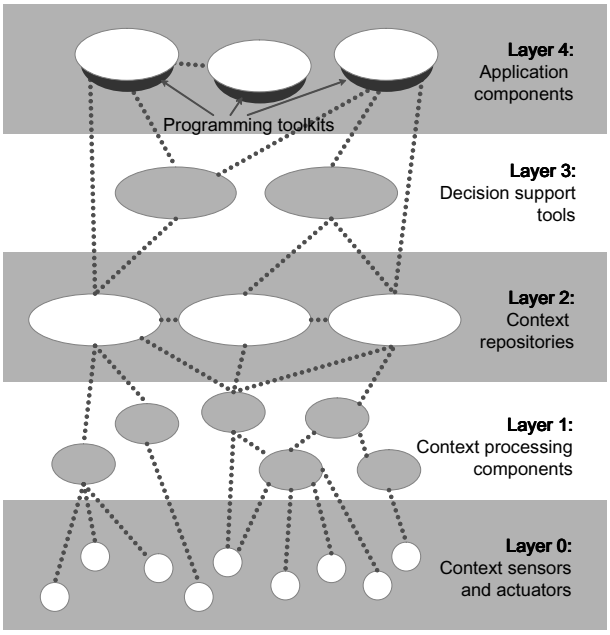

Fig. 1. Components of a context-aware system.

3 Middleware Requirements

In this paper, we refer to the components that reside between the layer 4 application components and the layer 0 sensors and actuators - together with the communications framework that binds the distributed components together - as middleware for context-aware systems. This middleware must address many of the requirements of traditional distributed systems, such as heterogeneity, mobility, scalability, and tolerance for component failures and disconnections. In addition, it must protect users' personal information, such as location and preferences, in accordance with their privacy preferences, and ensure that automatic actions taken by context-aware applications on behalf of users can be adequately understood and controlled by users. Finally, the large number of distributed components that are present in context-aware systems introduces a requirement for straightforward techniques for deploying, configuring and managing networks of sensors, actuators, context processing components, context repositories, and so on. A detailed summary of these requirements is provided in Table 1.

4 A Survey of Middleware for Context-Aware Systems

In this section, we review and analyse some of the proposed middleware solutions for context-aware systems. We focus on solutions that span multiple layers of the system architecture shown in Fig. 1; that is, we exclude single layer solutions

1. Support for heterogeneity	Hardware components ranging from resource-poor sensors, actuators and mobile client devices to high-performance servers must be supported, as must a variety of networking interfaces and programming languages. Legacy components may be present.
2. Support for mobility	All components (especially sensors and applications) can be mobile, and the communication protocols that underpin the system must therefore support appropriately flexible forms of routing. Context information may need to migrate with context-aware components. Flexible component discovery mechanisms are required.
3. Scalability	Context processing components and communication protocols must perform adequately in systems ranging from few to many sensors, actuators and application components. Similarly, they must scale to many administrative domains.
4. Support for privacy	Flows of context information between the distributed components of a context-aware system must be controlled according to users' privacy needs and expectations.
5. Traceability and control	The state of the system components and information flows between components should be open to inspection - and, where relevant, manipulation - in order to provide adequate understanding and control of the system to users, and to facilitate debugging.
6. Tolerance for component failures	Sensors and other components are likely to fail in the ordinary operation of a context-aware system. Disconnections may also occur. The system must continue operation, without requiring excessive resources to detect and handle failures.
7. Ease of deployment and configuration	The distributed hardware and software components of a context-aware system must be easily deployed and configured to meet user and environmental requirements, potentially by non-experts (for example, in "smart home" environments).

Table 1. Requirements for middleware for context-aware systems

such as context servers (layer 2) and models for context interpretation (layer 1). We also exclude solutions that are not general, such as those that deal only with location sensing and management.

4.1 The Context Toolkit

Dey et al.'s Context Toolkit [1] provides a set of abstractions that can be used to implement reusable software components for context sensing and interpretation. The context *widget* abstraction represents a component that is responsible for acquiring context information directly from a sensor. Widgets can be combined with *interpreters*, which transform low-level information into higher-level information that is more useful to applications, and *aggregators*, which group related

context information together in a single component. Finally, *services* can be used by context-aware applications to invoke actions using actuators, and *discoverers* can be used by applications to locate suitable widgets, interpreters, aggregators and services.

The toolkit is implemented as a set of Java objects that represent the abstractions described above. These provide a basic communication protocol based on HTTP and XML. The use of these Web standards allows for interoperation with components implemented in other languages, thereby providing basic support for heterogeneity. The toolkit's *discoverers* address component discovery, which is one of the requirements for mobility. However, the toolkit does not specifically address scalability, privacy, traceability/control[3], component failures, or deployment/configuration.

4.2 Context Fusion Networks

Chen et al. [3] propose the use of *Context Fusion Networks* (CFNs) to provide data fusion services (aggregation and interpretation of sensor data) to context-aware applications. CFNs are based on an operator graph model, in which context processing is specified by application developers in terms of sources, sinks and channels. In this model, sensors are represented by sources, and applications by sinks. Operators, which are responsible for data processing, act as both sources and sinks.

Chen et al. have implemented the CFN model in the form of *Solar*, a scalable peer-to-peer (P2P) platform which instantiates the operator graphs at runtime on behalf of context-aware applications. The Solar hosts (*Planets*) support application and sensor mobility by buffering events during periods of disconnection; they also address component failures by providing monitoring and recovery, as well as preservation of component states. However, Solar does not yet address heterogeneity, privacy, or monitoring and control of the system by users.

4.3 The Context Fabric

Unlike the previous two solutions, the Context Fabric (Confab) proposed by Hong and Landay [4] is primarily concerned with privacy rather than with context sensing and processing. Confab provides an architecture for privacy-sensitive systems, as well as a set of privacy mechanisms that can be used by application developers. The architecture structures context information into *infospaces*, which store tuples about a given entity. Infospaces are populated by context sources such as sensors, and queried by context-aware applications.

Hong and Landay have implemented the infospace model using Web technologies, such that infospaces are identified by URLs and tuples are exchanged in an XML format. They provide a programming model based on *in* and *out* methods for transferring tuples into and out of infospaces. Privacy can be supported

[3] However, Newberger and Dey [2] did later address monitoring and control by providing an *enactor* extension to the Context Toolkit.

by adding operators to an infospace to carry out actions when tuples enter or leave the space; for instance, operators can be used to perform access control, notify users of information disclosure, and enforce privacy tags that describe how information can be used after it flows from one infospace to another.

As Confab focuses so heavily on privacy, it does not address traditional distributed systems requirements such as mobility, scalability, component failures and deployment/configuration. However, it does partially address heterogeneity, as it builds on platform- and language-independent Web standards. It also provides privacy-related traceability and control via the operator mechanism.

4.4 Gaia

Gaia [5] is designed to facilitate the construction of applications for smart spaces, such as smart homes and meeting rooms. It consists of a set of core services and a framework for building distributed context-aware applications. Gaia's event manager service enables applications to be developed as loosely coupled components, and can provide basic fault tolerance by allowing failed event producers to be automatically replaced. Gaia's remaining four services support various forms of context-awareness, and include: (i) a context service, which allows applications to find providers for the context information they require, (ii) a presence service, which monitors the entities entering and leaving a smart space (including people as well as hardware and software components), (iii) a space repository, which maintains descriptions of hardware and software components, and (iv) a context file system, which associates files with relevant context information and dynamically constructs virtual directory hierarchies according to the current context.

As smart spaces are typically small, constrained environments, Gaia does not address scalability (however, [6] canvasses the issues involved in federating spaces into large-scale "super spaces"). Similarly, privacy is not addressed by any of the basic services, but can potentially be provided by additional services [7], while user monitoring/control is outside Gaia's scope. Heterogeneity, mobility and component configuration can all be supported by Gaia in limited forms.

4.5 Reconfigurable Context-Sensitive Middleware

Yau et al. [8] propose a Reconfigurable Context-Sensitive Middleware (RCSM) for context-aware applications. The RCSM provides application developers with a novel Interface Definition Language (IDL) that can be used to specify context requirements, including the types of context/situation that are relevant to the application, the actions to be triggered, and the timing of these actions. The IDL interfaces are compiled to produce application skeletons; these interact at runtime with the RCSM Object Request Broker (R-ORB), which manages context acquisition, and the Situation-Awareness (SA) processor, which is responsible for managing triggers.

The R-ORB provides a context manager that uses a context discovery protocol to manage registrations of local sensors and discover remote sensors. When

Requirement	Context Toolkit	CFN/ Solar	Context Fabric	Gaia	RCSM
Support for heterogeneity	✓	×	✓	✓	✓
Support for mobility	✓	√	×	✓	×
Scalability	×	√	×	×	×
Support for privacy	×	×	√	×	×
Traceability and control	×	×	✓	×	×
Tolerance for failures	×	√	×	✓	×
Ease of deployment/ configuration	×	✓	×	✓	✓

Table 2. Middleware support for the requirements of context-aware systems. (**Key:** √ = comprehensive, ✓ = partial, × = none)

a context-aware application starts up, the discovery protocol is used to look for local or remote sensors that satisfy the application's context requirements.

The prototype described by Yau et al. does not satisfy the heterogeneity requirement, as it supports only C++ applications for the Windows CE platform; however, the IDL compiler could potentially be modified to produce skeletons for a variety of platforms and communication protocols. In addition, the context discovery protocol is not flexible enough to support mobility or component failure, and Yau et al. do not attempt to address scalability, privacy or traceability/control. The main strength of the approach comes from the use of an IDL to specify context requirements. This makes it possible to incorporate new types of context and context-aware behaviour by editing and recompiling IDL interfaces, and partially addresses ease of deployment and configuration.

4.6 Analysis

Table 2 summarises the capabilities of the surveyed solutions and shows that comprehensive solutions do not yet exist. A further shortcoming, which is not revealed in the table, is that none of the solutions provide decision support (layer 3 functionality). Our own middleware, which we discuss next, introduces decision support and addresses a large subset of the requirements listed in Table 1.

5 The PACE Middleware

Our middleware was developed as part of the PACE project, which investigates a variety of issues related to pervasive computing, including the design of context-aware applications and solutions for modelling and managing context information. An early form of the middleware was presented in [9]; however, further tools and components have been added subsequently as we developed further context-aware applications and uncovered additional requirements. Our current version of the middleware consists of:

– a context management system (layer 2);
– a preference management system that provides customisable decision-support for context-aware applications (layer 3);
– a programming toolkit that facilitates interaction between application components and the context and preference management systems (layer 4); and
– tools that assist with generating components that can be used by all layers, including a flexible messaging framework.

These components and tools have been developed according to the following design principles:

1. The model(s) of context information used in a context-aware system should be explicitly represented within the system. This representation should be separate from the application components (layer 4) and the parts of the system concerned with sensing and actuation (layers 0 and 1), so that the context model can evolve independently, without requiring any components to be re-implemented.
2. The context-aware behaviour of context-aware applications should be determined, at least in part, by external specifications that can be customised by users and evolved along with the context model (again, without forcing re-implementation of any components).
3. The communication between application components, and between the components and middleware services, should not be tightly bound to the application logic, so that a significant re-implementation effort is required when the underlying transport protocols or service interfaces change.

The following sections provide an overview of the components and tools that make up the middleware. In Section 6, we illustrate their use in the development of a context-aware system that supports vertical handover of media streams.

5.1 Context Management System

In our middleware, the context management system fulfils the requirements of layer 2 as discussed in Section 2: that is, it provides aggregation and storage of context information, in addition to performing query evaluation. It uses a two-layered context modelling approach, in which context can be expressed both in terms of fine-grained facts and higher-level situations which capture logical conditions that can be *true*, *false* or *unknown* in a certain context. All information is stored in the fact representation, but can be queried by either retrieving specific facts based on template matching, or evaluating situation definitions over a set of facts. Our context modelling approach has been well documented in previous papers [9, 10], and therefore is not described in detail here. However, an example fact-based context model will be shown later in Section 6.

The context management system consists of a distributed set of context repositories. Each repository manages a catalog, which is a collection of context models consisting of fact type and situation definitions. Applications may

define their own context models or share them with other applications. Context-aware components are not statically linked to a single repository, but can discover repositories dynamically by catalog name (and potentially also other attributes). Several methods of interacting with a context repository are currently permitted, in order to support a range of client programming languages and platforms; likewise, a variety of discovery mechanisms can be used, including context-based discovery, which allows for matching based on context attributes.

Each repository is capable of performing access control, although this feature can be switched off if it is not required. The access control mechanism allows users to define privacy preferences that dictate the circumstances (i.e., situations) in which context information can be queried and updated. The privacy preferences are stored and evaluated by the preference management system, which we describe in the following section.

Our current prototype consists of a context management layer running on top of a relational database management system. This is written in Java using JDBC[4] to query and manipulate a set of context databases[5]. It provides clients with the following interfaces:

- *query*: supports situation evaluation and retrieval of facts matching supplied templates;
- *update*: allows insertion, deletion and modification of facts, as well as insertion of new situation definitions;
- *transaction*: allows clients to create read-only transactions within which a sequence of queries can be executed against a consistent set of context information, regardless of concurrent updates;
- *subscription*: allows monitoring of situations and fact types, using callbacks to notification interfaces implemented by clients; and
- *metadata*: allows clients to discover the fact types and situations that are defined by models in the catalog.

In addition to invoking methods on repositories using Java RMI, clients can use a Web interface (based on XML and HTTP) or programming language stubs generated from a context model specification. The latter method can potentially accommodate arbitrary programming languages and communication protocols; currently, we generate stubs for Java and Python, using Elvin [11], a content-based message routing scheme, as the underlying communication paradigm. One of the benefits of Elvin is that it allows for complex interactions (including 1:N and N:M communication, not only 1:1 as supported by RMI and HTTP), which allows (for example) queries and updates to be simultaneously routed to multiple context repositories. We discuss the stubs further in Section 5.5.

Currently, our context repositories behave independently of one another; however, we are developing a model for replicating context information across

[4] http://java.sun.com/products/jdbc/

[5] Note that this is not the most efficient implementation in terms of query/update time and throughput, but we have found the performance adequate for all of the context-aware applications we have developed so far.

several repositories and allowing clients to cache their own context information for use during disconnections.

5.2 Preference Management System

A preference management system provides layer 3 functionality that builds on functionality of the context management system. It assists context-aware applications with making context-based decisions on behalf of users. Its main roles are to provide storage of user preference information and evaluation of preferences - with respect to application state variables and context information stored by the context management system - to determine which application actions are preferred by the user in the current context. Applications can connect to, and store their preference information in, one or more preference repositories.

The preferences are defined in terms of our novel preference model, which allows the description of context-dependent requirements in a form that enables them to be combined on-the-fly to support decisions about users' preferred choice(s) from a set of available candidates. For example, the preference model can be used to decide which mode of input or output should be employed for particular users according to their requirements and current contexts. A detailed description of the preference model is outside the scope of this paper, but further information can be found in earlier papers [9, 10, 12].

The benefits of a preference-based approach to decision-making are that customisation and evolution of context-aware behaviour can be supported in a straightforward manner; preferences can be shared and exchanged between applications; and new types of context information can be incorporated into decision-making processes simply by adding new preferences, without the need to modify the application components.

The implementation of the preference management system bears strong resembles to that of the context management system, and therefore we discuss it relatively briefly. It provides the following interfaces:

- *update*: allows new preferences to be defined and grouped appropriately into sets (for instance, by owner and purpose);
- *query*: provides preference evaluation based on the information stored in the context management system;
- *transaction*: allows a set of preference evaluations to occur over a consistent set of context information, regardless of concurrent updates occurring within the context management layer; and
- *metadata*: allows retrieval of preference and preference set definitions.

In a similar manner to the context repositories, the preference repositories respond to requests from clients over a variety of communication protocols. However, Java clients need not interact directly with repositories; instead, they are provided with a Java programming toolkit that assists with discovery of, and interaction with, repositories. We describe the toolkit in the following section.

5.3 Programming Toolkit

The programming toolkit complements the functionality of the preference management layer by implementing a simple conceptual model for formulating and carrying out context-based choices. The model provides a mechanism for linking application actions with candidate choices. It also allows one or more of the actions to be automatically invoked on the basis of the results of evaluating the choices with respect to preference and context information, using the services of the preference and context management systems.

A significant benefit of the toolkit is that it makes the process of discovering and communicating with the preference and context management systems transparent to applications. It also helps to produce applications that are cleanly structured and decoupled from their context models, and thus better able to support changes in the available context information. These changes can result from evolution of the sensing infrastructure over time, or problems such as disconnection or migration from a sensor-rich environment to a sensor-poor one.

The toolkit is currently only implemented in Java, using RMI for communication with remote components; however, it could be ported to other programming languages and communication protocols in the future.

5.4 Messaging Framework

To facilitate remote communication between components of context-aware systems - which may be either application components or middleware services such as the ones described in Sections 5.1 and 5.2 - we provide a flexible messaging framework. In the tradition of middleware such as CORBA, the framework aims to provide various forms of transparency, such as location and migration transparency. It maps interface definitions to communication stubs that are appropriate for the deployment environment. These stubs are considerably simpler for the programmer to work with than the APIs of the underlying transport layers, and can also be automatically re-generated at a later date, allowing for substitution of transport layers without modifying the application.

Stubs can be generated for a variety of programming languages and communication protocols (including message-based and RPC-based protocols). To date, however, we have focused on producing Java and Python stubs for the Elvin publish/subscribe content-based message routing scheme. Elvin is particularly appropriate for building context-aware systems because it decouples communication from cooperation. Because it delivers messages based on matches between message content and the subscriptions of listeners, rather than based on explicit addressing, it is able to tolerate mobility, support complex interactions (not only 1:1 interactions as in the case of RPC/RMI), and allow for spontaneous interactions between components without the need for an explicit discovery/configuration step. The ability to add new listeners into the system on-the-fly is also useful for debugging and generating traces.

In the future, we plan to extend the messaging framework to other protocols appropriate for context-aware systems (for example, context-based routing schemes such as GeoCast [13], which performs routing based on location).

5.5 Schema Compiler Toolset

The final piece of our middleware is a set of tools capable of producing custom components to assist with developing and deploying context-aware systems, starting from context models specified using the two-layered context modelling approach that we briefly outlined in Section 5.1. The tools take input in the form of a textual representation of a context model (a context *schema*), perform checks to verify the integrity of the model, and produce the following outputs:

- SQL scripts to load and remove context model definitions from the relational databases used by our context repositories;
- model-specific helper classes to simplify source code concerned with carrying out context queries and updates; and
- context model interface definitions compatible with the messaging framework.

The first output simplifies the deployment and evolution of context models. By automating the mapping of context models into the database structures stored by the context repositories, errors that might arise during the hand-coding of SQL scripts or JDBC code to manipulate the repositories can be avoided. Similarly, updates to context models can be supported simply by re-generating and re-executing the scripts. In the future, we envision extending the tools to produce alternative scripts for context repositories that are not SQL-based.

The second output is designed to simplify the programming of components that query or update a context model, and includes classes that represent basic value types, fact types and situations defined by the model. By programming with these classes, rather than the generic APIs provided by the context management layer, type checking becomes possible at compile time and standard IDE features such as code completion can better be exploited.

The final output is used to produce stubs for transmitting/receiving context information over communications infrastructure such as Elvin. The context transmitters can be used by layer 0 and layer 1 components (sensors, actuators and processing components) to transmit context information to one or more context repositories. Similarly, the context receivers can be used at layer 2 to listen for context updates that require mapping to operations on context repositories.

Further information about the context schema toolset can be found in [14].

6 Case Study: Vertical Handover

We now illustrate how our middleware assists with the development of distributed context-aware systems, using a vertical handover application as a case study. This application represents just one of the context-aware applications we have developed using the middleware; others are described in earlier papers [9, 12]. The application is concerned with adapting the streaming of media to a mobile user according to the context. The adaptation occurs at the application layer rather than the network layer (e.g., using Mobile IP) because of stringent Quality of Service (QoS) requirements. The application adapts by handing

over the stream, either between network interfaces on a single computing device or between interfaces on different devices. A handover can potentially occur in response to any context change; for example, the user moving into range of a network that offers higher bandwidth than the current network, or the signal strength of the current network dropping.

The handover process is managed by adaptation managers and proxies. The adaptation managers use the context management system to monitor significant context changes, in order to determine when a vertical handover should occur, and to which network interface. The proxies perform the handover process. One proxy is co-located with the transmitter, while other proxies are located within the same networks as the receivers. The transmitter's local proxy (proxy-transmitter) is responsible for redirecting the stream when it receives a handover instruction from an adaptation manager. During the handover process, the proxy-transmitter transmits the stream to both the original and the new proxy. This is referred to as doublecasting. The proxies within the receivers' networks are responsible for forwarding the streams to the receiver(s) executing on the client device(s). When the handover is complete, the proxy-transmitter stops transmitting to the original receiver proxy.

6.1 Implementation

The architecture of the system, including both application components and supporting middleware components, is shown in Fig. 2. In the remainder of this section, we demonstrate how our middleware was used to implement the system.

Context Model. The context model used by the vertical handover prototype is shown in Fig. 3. The main objects described by the model are computing devices, networks, network interfaces, streams and proxies. The model captures associations between computing devices and network interfaces, proximity between devices, mappings of streams to proxies and network interfaces, basic QoS information related to network interfaces (current signal strength and bandwidth), and other type and configuration information. Much of the information is user- or application-supplied (i.e., static or profiled in the terminology of our context modelling approach); however, proximity between devices is sensed using wireless beacons, and current network connectivity, signal strength and bandwidth are all sensed by monitors running in the network.

The context model, and its instantiation at run-time with concrete facts, is managed by a set of context repositories as shown in Fig. 2. Each local network may contain one or more repositories. In the example system architecture shown in the diagram, two of the local networks possess their own context repositories, while the other network does not. However, the design of the system is such that many other configurations are also possible.

The schema compiler toolset described in Section 5.5 was used to map the context model to appropriate database structures when the context repositories were deployed. The toolset was also used to generate context transmitter and

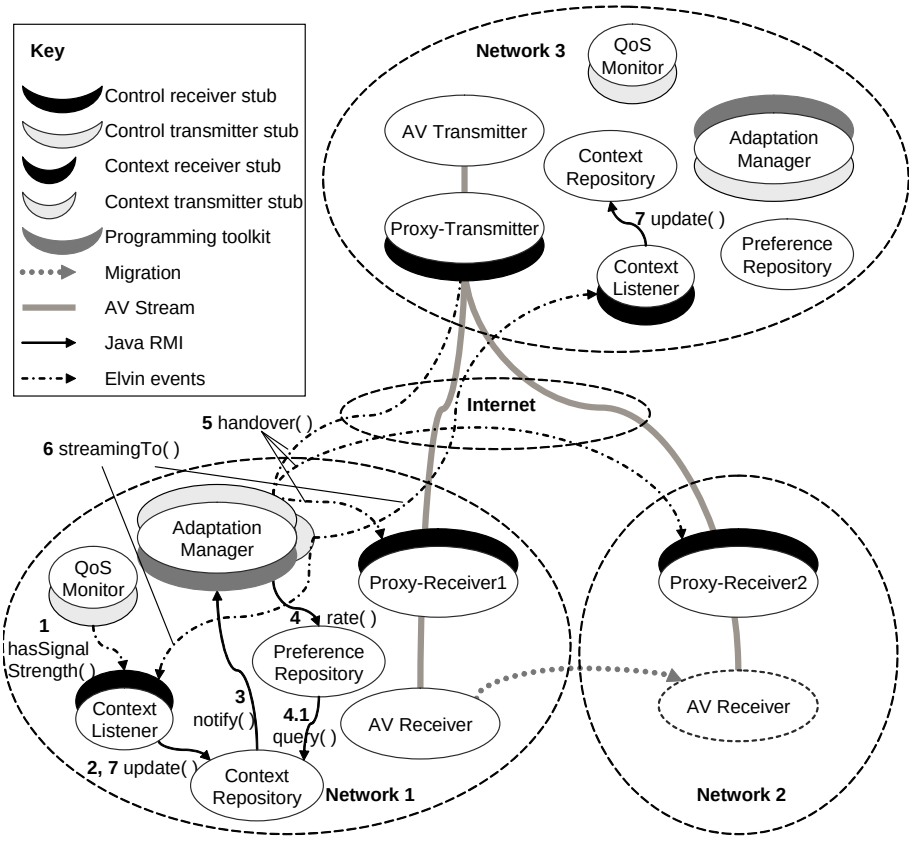

Fig. 2. Vertical handover architecture.

receiver stubs, which are used by the wireless beacons (not shown in the diagram), network monitors and adaptation managers (shown for networks 1 and 3) to report context information to the context repositories over Elvin, via context listeners that map the Elvin notifications to RMI context repository updates.

Adaptation managers. The context-aware functionality of the application is concentrated within the adaptation managers. These are the components that are responsible for determining when handover is required, according to the current context and user preferences. Therefore, the adaptation managers are the components that interact with the context and preference repositories.

According to the design principles outlined in Section 5, the adaptation managers are not tightly coupled to the context model. The only direct interaction that occurs between the adaptation managers and the context repositories is in the form of subscriptions/notifications, which allow the managers to learn about significant context changes and report new streaming configurations. The

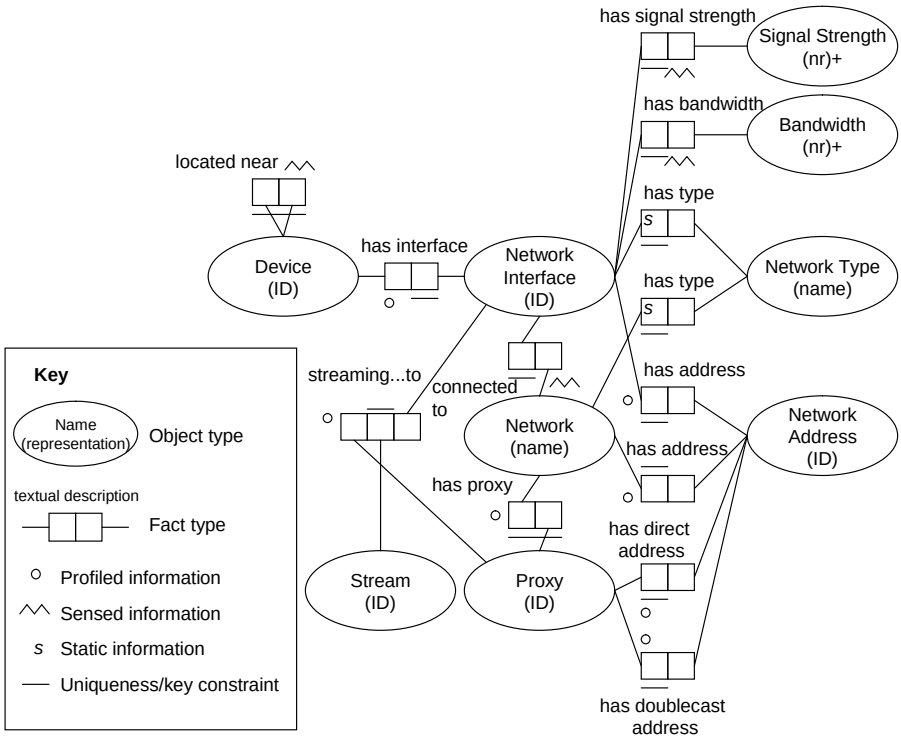

Fig. 3. The context model used in the vertical handover application.

subscriptions monitor the state of the sensed fact types. When an adaptation manager is notified of a change (for example, a drop in signal strength, as shown in Fig. 2), it uses the programming toolkit described in Section 5.3 to connect to a preference repository, re-evaluate the user's preferences, and determine whether the current network interface is still the preferred one (step 4 in the figure). The bulk of the context evaluation occurs during this step, as a side-effect of the preference evaluation (step 4.1). New context information can be easily incorporated into the evaluation simply by extending the user preferences (i.e., the implementation of the adaptation manager does not need to change).

When an adaptation manager determines that a handover is required, it communicates with the proxies (step 5) and the context listeners (step 6) using Elvin. The manager first transmits a handover instruction to the proxies using Elvin stubs produced by the PACE messaging framework described in Section 5.4. After instructing the proxies to perform the handover, the adaptation manager updates the stream state information stored in the context repositories (i.e., the "streaming...to" fact type shown in Fig. 3), using a context transmitter stub to transmit the information to the context listeners.

7 Analysis

In this section, we briefly analyse the PACE middleware with respect to the requirements set out in Table 1 and compare it to the earlier solutions surveyed in Section 4. Based on the analysis, we also highlight some areas for future work.

Heterogeneity. As the PACE messaging framework can generate stubs for a variety of programming languages (and, in the future, transport layers), it offers strong support for heterogeneity. The PACE middleware is also capable of accommodating legacy components, such as the transmitters and receivers in the vertical handover system. Thus, PACE's support for heterogeneity is more comprehensive than the solutions surveyed in Section 4. However, Yau et al.'s solution bears some similarities to our approach, and could be extended to generate skeletons for a variety of platforms as discussed in Section 4.5.

Mobility. The use of Elvin within the messaging framework facilitates component mobility, as demonstrated in the vertical handover system, and often removes the need for component discovery. Local context and preference repositories can be dynamically discovered by mobile context-aware components using a variety of service discovery protocols. PACE therefore provides a level of mobility support that is comparable to the best solutions surveyed in Section 4. In the future, we plan to extend PACE's current support for mobility by introducing caching/hoarding models for context and preference information, to allow mobile components to store local copies of information that is relevant to users.

Scalability. Our current implementation of the PACE middleware does not address scalability or performance, and the same can be said of almost all of the solutions surveyed in Section 4. This is unsurprising, as all have been developed as research prototypes. As future work, we intend to develop models for federating context and preference managers across large scale systems and a large number of administrative domains.

Privacy. We address privacy by providing access control for sensitive context information. Thus, our middleware provides more privacy support than all of the surveyed solutions, with the exception of Confab. However, controlling access to context information addresses only one aspect of privacy. To address other aspects, we intend to add access control to our preference management system and combine this access control with context-based authentication [13]. Further information about our current work on privacy can be found in [15] and [16].

Traceability and Control. We showed in Section 4 that traceability and control are not addressed at all by previous middleware except in relation to privacy. The PACE middleware begins to address this problem. The use of Elvin facilitates the generation of traces, as event listeners can be added on-the-fly and

event traces can be tailored by adjusting the Elvin subscriptions. Our preference model also provides a basic mechanism for user control and customisation. In the future, we envisage opening up the service layers to clients to allow inspection (and manipulation) of context and preference evaluations. Traces of these evaluations can be selectively revealed to users to explain system behaviours.

Tolerance for failures Our solution's failure tolerance ranks behind that of Solar but ahead of the remaining solutions surveyed in Section 4. Although our middleware does not yet detect or repair failed components, its use of Elvin allows a loose coupling of components, minimising the impact of disconnections and failures. In addition, our context and preference models were both designed with the assumption that context information will generally be imperfect. This introduces some tolerance for failed sensors, sensing errors, and so on.

Deployment and Configuration. Finally, the PACE middleware provides more advanced support for component deployment and configuration than previous solutions. Specifically, the messaging framework simplifies the deployment of components on top of a variety of platforms, while the schema compiler toolset facilitates the deployment of new context models. However, further extensions to the middleware are needed to facilitate the scalable deployment and configuration of infrastructural components such as sensors.

8 Conclusions

In this paper, we showed that middleware is essential for building context-aware systems and introduced a list of requirements that this middleware must address. We also analysed the current state-of-the-art in the area and provided a comprehensive discussion and evaluation of our own PACE middleware. Our solution ranked the best or equal best for the majority of the requirements (heterogeneity, mobility, traceability/control and deployment/configuration), and above average for two of the remaining three requirements (privacy and tolerance for failures). A further advantage of the PACE middleware is that it provides decision support (i.e., layer 3 functionality), unlike the other solutions we surveyed. However, many problems have not yet been adequately addressed by our work or that of the broader research community - for example, scalable deployment, configuration and management of sensors, caching and hoarding of context information to support mobility, and mechanisms for revealing aspects of the system state to facilitate user understanding and control.

References

1. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction **16** (2001) 97–166

2. Newberger, A., Dey, A.: Designer support for context monitoring and control. Technical Report IRB-TR-03-017, Intel Research Berkeley (2003)
3. Chen, G., Li, M., Kotz, D.: Design and implementation of a large-scale context fusion network. In: 1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous), IEEE Computer Society (2004) 246–255
4. Hong, J.I., Landay, J.A.: An architecture for privacy-sensitive ubiquitous computing. In: 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys), Boston (2004)
5. Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: Gaia: A middleware infrastructure for active spaces. IEEE Pervasive Computing, Special Issue on Wearable Computing **1** (2002) 74–83
6. Al-Muhtadi, J., Chetan, S., Ranganathan, A., Campbell, R.: Super spaces: A middleware for large-scale pervasive computing environments. In: Workshop on Middleware Support for Pervasive Computing (PerWare), PerCom'04 Workshop Proceedings, Orlando (2004) 198–202
7. Al-Muhtadi, J., Ranganathan, A., Campbell, R., Mickunas, M.D.: Cerberus: A context-aware security scheme for smart spaces. In: 1st IEEE International Conference on Pervasive Computing and Communications (PerCom), Fort Worth (2003) 489–496
8. Yau, S.S., Huang, D., Gong, H., Seth, S.: Development and runtime support for situation-aware application software in ubiquitous computing environments. In: 28th Annual International Computer Software and Application Conference (COMPSAC), Hong Kong (2004) 452–457
9. Henricksen, K., Indulska, J.: A software engineering framework for context-aware pervasive computing. In: 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer Society (2004) 77–86
10. Indulska, J., Henricksen, K., McFadden, T., Mascaro, P.: Towards a common context model for virtual community applications. In: 2nd International Conference on Smart Homes and Health Telematics (ICOST). Volume 14 of Assistive Technology Research Series., IOS Press (2004) 154–161
11. Segall, B., Arnold, D., Boot, J., Henderson, M., Phelps, T.: Content based routing with Elvin4. In: AUUG2K Conference, Canberra (2000)
12. McFadden, T., Henricksen, K., Indulska, J., Mascaro, P.: Applying a disciplined approach to the development of a context-aware communication application. In: 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer Society (2005) 300–306
13. Navas, J.C., Imielinski, T.: Geographic addressing and routing. In: 3rd ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), Budapest (1997)
14. McFadden, T., Henricksen, K., Indulska, J.: Automating context-aware application development. In: UbiComp 1st International Workshop on Advanced Context Modelling, Reasoning and Management, Nottingham (2004) 90–95
15. Henricksen, K., Wishart, R., McFadden, T., Indulska, J.: Extending context models for privacy in pervasive computing environments. In: 2nd International Workshop on Context Modelling and Reasoning (CoMoRea), PerCom'05 Workshop Proceedings, IEEE Computer Society (2005) 20–24
16. Wishart, R., Henricksen, K., Indulska, J.: Context obfuscation for privacy via ontological descriptions. In: 1st International Workshop on Location- and Context-Awareness. Volume 1678 of Lecture Notes in Computer Science., Springer (2005) 276–288