Merino: An Intelligent Environment Architecture for Scalable Context Processing

No Author Given

No Institute Given

Abstract. This paper describes the the Merino architecture for scalable management of an intelligent environment. We define requirements for such an architecture. We then review previous work that has informed our design and which has addressed aspects of some of these requirements.
The core of the architecture is its abstraction layers: the core *Sensors* and *Device* Layers; the next level of abstraction in the *Context* and *Device Abstraction* Layers; and the highest abstraction level, the *Smart Environment Agent* Layer. These are unified by the *Context Repository* and *User Model*. After describing this, we show how the scalable distribution of context and user model information is managed in terms of a hierachy of administrative domains. We describe the prototype implementation of Merino.

1 Introduction

The long term vision of pervasive computing has several facets. Firstly, taking the user's perspective, a pervasive environment offers the promise that people will be able to interact with their digital support tools *at any time and any place.* Such pervasive access to digital tools means that users should be able to work with these tools in the full range of their normal activities. So, for example, they will be able to access them in the many and diverse locations of daily activity: at home, in their car, on the bus or train, in the supermarket, and at work. They should also have access to these digital artefact's in less common situations, such as at the hotel on their holiday or at the convention centre on a business trip.

In addition to being accessible anywhere, the future pervasive computing environment must be *context-aware.* This includes awareness of such diverse elements as the user's location, their activity and the social context defined by the presence of other people. The way that a digital artefact should interact with the user depends upon that context.

Another critical aspect of the user's view is that the pervasive computing environment should support *two way flows of data.* This means that the user should be able to initiate activity with a digital artefact. In addition, it should be possible for such artefacts to initiate interaction with the user where this is appropriate.

The effectiveness of pervasively accessible digital artefacts will often be enhanced if those artefacts are *personalised* to their particular user. This means

that the same artefacts will operate differently for different people, even when they are in the same context. This means that the pervasive computing environment will hold and manage the relevant personal information about the user.

At quite a different level, users will only accept a pervasive computing environment if they feel that they have an adequate *sense of control.* This aspect applies to all the elements we have described: the user should be able to control the times and places that their digital artefacts should be able to operate; they should be able to control the operation of those artefacts in specific contexts; they should be able to determine when the system is allowed to take the initiative and finally, they need to have a sense of control over both personalisation processes and the personal information which underlies it. Of course, we are not suggesting that people will want to spend much time managing all these aspects. Indeed, people will generally want most aspects of the pervasive computing environment to be invisible, operating as needed and without the need for undue work from the user. However, the user must be able to exact just the level of control that they wish, when they wish to do so. For example, a person may make use of a reminder artefact for months. One day, it might work in an unexpected or irritating way. At that point, the user might decide that they want to be able to *scrutinise* the way that this artefact works so that they can then *control* it so that matches their preference better. Equally, one the user has scrutinised the way that they system came to act as it did, they might decide that it is acceptable.

This paper describes our work towards developing an infrastructure which will be able to support this user-view of the future pervasive computing environment. That environment will need *intermediate storage and processing* between the context gathering parts of the environment and the application programs at the heart of the digital artefacts. A key component of that infrastructure will be a distributed database for the information collected from the pervasive computing environment and used by the digital artefacts. This database must have several characteristics that pose considerable technical challenges. It must scale to global proportions. It should be accessible to the many and diverse devices within the pervasive computing environment. This means that it must be accessible via a light-weight access protocol. It must also be discoverable.

In this paper, we describe the Merino architecture which will underly the infrastructure for the vision we have described for the future pervasive computing environment. In Section 2, we review some of the work that has already been done. Section 3 described the high level view of the architectural layers and in Section 4, we describe the technical elements to realise that architecture. Section 5 discusses the current implementation work.

2 Related Work

This section attempts to identify Merino's position in relation to work that has been done in the complex area of gathering, processing, and representing context. In [1], Weiser described the dynamically changing environments as one of the

biggest challenges for context aware computing. Modularity, Decomposability, and scalability play a key role in design of ubiquitous systems. Despite that early insight, not many of the approaches so far have been able to successfully address these issues.

2.1 Projects and Major Case Studies

Many projects have attempted to cover a narrow path from low level system devices, such as sensors and actuators, to complex, context-aware applications at the top. The Aware Home Project [2], for example, attempts to create a home environment that is aware of its occupants' activities and locations. The project is a conglomeration of a number of building blocks. Components include indoor location tracking, activity recognition, large-scale projective displays, wireless sensor networks, etc. These components are built on top of a toolkit [3] that forms the core API of the context awareness infrastructure. While this toolkit provides a versatile environment for individual intelligent homes, it lacks scalability for larger, more structured intelligent environments.

Project Oxygen [4] is an umbrella project at the Massachusetts Institute of Technology (MIT) that supports research aimed at replacing the PC with ubiquitous computing devices. This project mainly deals with user interaction and its emphasis is to use speech and vision for as a natural user interface. As such, it deals less with the structure of the intelligent environment itself, but more with its perception of and by the user. Another project situated at the MIT, the Intelligent Room, also deals with user interaction. They use a Java based programming language called Metaglue as a coordination framework for multi-agent systems [5]. Both projects are orthogonal to a system infrastructure for intelligent environments and could be easily modified to work on top of a more generic context architecture.

The projects described so far provide toolkits, APIs, or agent frameworks that allow applications to utilise context information. CoolTown [6] attempts to extend this approach by giving people, places, and things a presence on the world wide web. Their rationale is that the convergence of web technology, wireless networks, and mobile devices provides a natural structure for embodying physically related objects into web servers. They suggest either manual entry or specialised resolvers that can, for example, be linked to the location of a device to get the URL for a specific web presence. Our architecture uses a similar approach but attempts to present a unified, more scalable interface that works for global, inter-connected intelligent environments as well as individual local environments.

2.2 Context Infrastructures

In addition to projects and case studies, some attempts have been made to provide environments and infrastructures for context information. Elvin [7] is an I/O oriented infrastructure that connects sensors and actuators to applications. It relies on the content of messages to route them between different nodes

within a network. This framework was modelled as part of a Locales Framework for Computer Supported Cooperative Work [8]. A centralised event notification service provided by Elvin allows consumers such as applications to subscribe to state changes with in the environment or the system. This centralised notification service works well within boundaries defined *a priori*, but does not scale well in dynamically changing environment structures.

In [3], Abowd *et al* presented an abstraction for context information that is based on the widget user interface model. Sensors, actuators, or abstract context information is represented by a widget that can send or receive events from applications or GUI elements. This allows context to be handled in a manner similar to user input. Based on their concept, the authors examined the feasibility of a context-based infrastructure using this abstraction [9]. The authors concluded that their approach did not completely support the transparent acquisition of context for applications and that some sort of resource discovery was needed to effectively hide such details from the application. Grim *et al* later presented a similar abstraction model that uses a task/tuple architecture to handle context information [10]. This architecture attempts to avoid scalability problems through restrictions to a closed system that requires conscious design upfront. It does not provide the flexibility to account for changes later on.

2.3 Scalability

Neither of the above architectures have been designed for dynamically changing distributed structures. While suited to the well defined, but constrained environments they have been design for, these architectures typically lack decomposability of scalability in more complex scenarios. Kantor and Redmiles attempt to address this dilemma by introducing a distributed group based subscription service [11]. They introduce a scalable cross application subscription service that allows applications to obtain awareness information from multiple sources. While their approach is flexible enough to allow applications to locate and subscribe to context information provided by other applications, their approach is restricted to client polling which makes it unsuitable for a large number of application scenarios.

At this stage, despite many attempts to solve these problems, decomposability and scalability remain the biggest challenges for distributed, intelligent environments. In the following sections we present a system architecture that can overcome these difficulties and work as a robust, underlying infrastructure for context-aware, intelligent applications.

3 Architecture

The core of our work is based upon an architecture which offers the right levels of abstraction for the services needed in the future pervasive computing environment, which we call an *Intelligent Environment.* Fig 1 shows the service layers of our Intelligent Environment's architecture. The figure has two main parts. At the

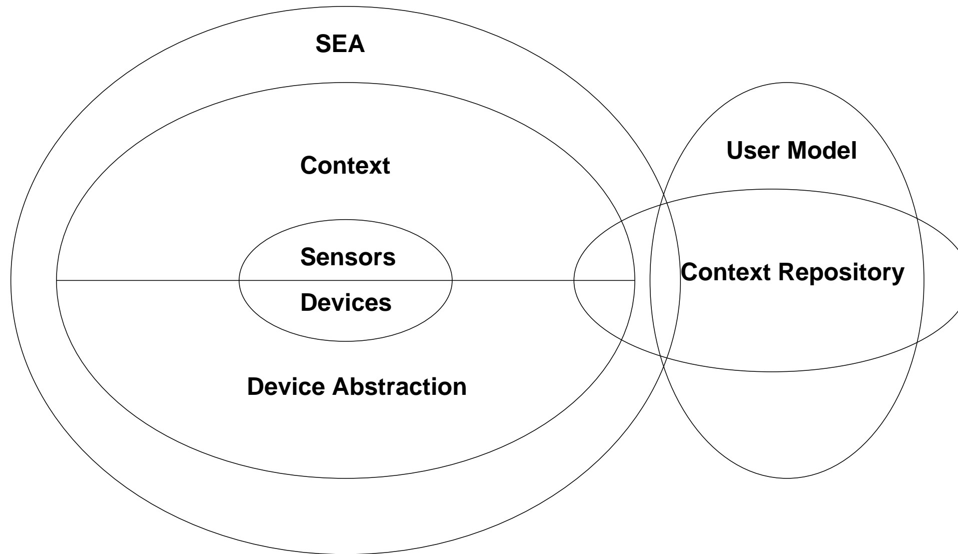

Fig. 1. Service layer description

left, we have concentric ellipses for the layers of abstraction. These manage the movement and transformation of information between the raw physical hardware devices within the environment and the application programs that implement digital artefacts, which users can access within the Intelligent Environment. The other main part of the figure is the Context Repository and User Model at the right.

Since both of these main elements are tightly coupled, we will explain the architecture by first showing how raw sensor information is managed, making its way to the Context Repository and User Model. Then we will describe these elements in more detail. Finally, we will describe how the whole architecture supports user interaction with a digital artefact.

We begin with the *Sensors* Layer in the upper part of the innermost service layer. This represents the range of sensors, which interrogate the physical and virtual environments. For example, some sensors may detect the location of a particular mobile phone. Others may detect a user's keyboard activity. A simple sensor might detect movement in an area. This layer is the part of the architecture that is directly connected to the variety of devices that are able to collect information that is needed for an Intelligent Environment.

We now move from this low level sensor to the next layer in the figure, the *Context* Layer. It must perform the core tasks of filtering and aggregating the raw sensor data from the lower layer. It collects data from one or more sensors and converts this to a higher level, closer to the needs of the digital artefacts. For example, there may be several temperature sensors in a room. Each of these produces data at the core *Sensors* Level; the *Context* layer combines these,

perhaps with a majority voting scheme to provide a single consensus temperature value for the room.

We now introduce the *Context Repository*, a key element, that unifies and manages the whole environment. The *Context* Layer interacts with the Context Repository. This holds collections of arbitrary context objects. Taking a very simple example, the *Sensors* Layer may detect that the user's keyboard is active. This is passed to the *Context* Layer which lodges the information directly into the *Context Repository.*

Context objects contain information about a sensor, a room, a device or any entity that has a role in the system. Taking a very simple example, a movement detector has an associated object containing its status: movement detected or not. For a mobile phone, there may be an object that contains information about the device such as its phone number, its Bluetooth MAC address, its owner and other capabilities.

The outer layer, the *Smart Environment Agent* Layer, takes information from the *Context* Layer, via the *Context Repository* and the *User Model* to construct higher level context information. This *Smart Environment Agent* Layer is the level of abstraction in the architecture is where the highest level of context information is handled. The same information from the lower layers may be reused by several *Smart Environment Agents* which operate at this level. Different agents may make different interpretations of the same information. For example, one digital artefact might interpret location information by very rigorous standards: it may require high levels of certainty about the user's location before it will treat the user's location as known. Another artefact might require lower certainty standards. This would mean that the same sensor data, interpreted by the *Context* Layer, then stored in the *Context Repository* would be regarded as giving the user's location for one artefact while the other would consider the user's location as unknown.

The figure shows the *User Model* overlapping the *Context Repository.* Indeed, our architecture treats these as quite similar. The main difference is that the *User Model* contains information about people and that is tagged to their identity. By contrast, the *Context Repository* holds context information from the *Context* Layer. This includes information from sensors which is used to model the environment. It also includes information from sensors that can model aspects of people. For example, information from movement detectors, Bluetooth detectors and keyboard activity monitors all provide information about people. While that information is anonymous, it can be kept in the *Context Repository.* Once it can be associated with an individual, it should be kept in the *User Model.* The reason that we distinguish these two forms of information is that the *User Model* has to be treated as personal information about people. This means that it is subject to additional constraints. In particular, the security and privacy of this data will need to conform to the requirements of emerging legislation, both national and international. Associated with this, we need to ensure that it is managed in ways that enable the user to scrutinise the information kept and the

ways that it is managed, interpreted and used. The user must also be able to maintain a sense of control over these aspects of their user model.

The *Device* and *Device Abstraction* parts of the architecture are motivated by the need to send data to low level devices in the pervasive computing environment. All but the simplest devices will have an object in the context repository containing attributes of the device. These attributes include device characteristics such as the phone number or Bluetooth MAC address in the mobile phone example given earlier. A Smart Environment Agent may wish to communicate with the phone to change its state, for example to request that it switch to silent mode during a meeting. The device abstraction layer includes agent programs that carry out this communication on behalf of the SEA.

4 Distributed Realisation

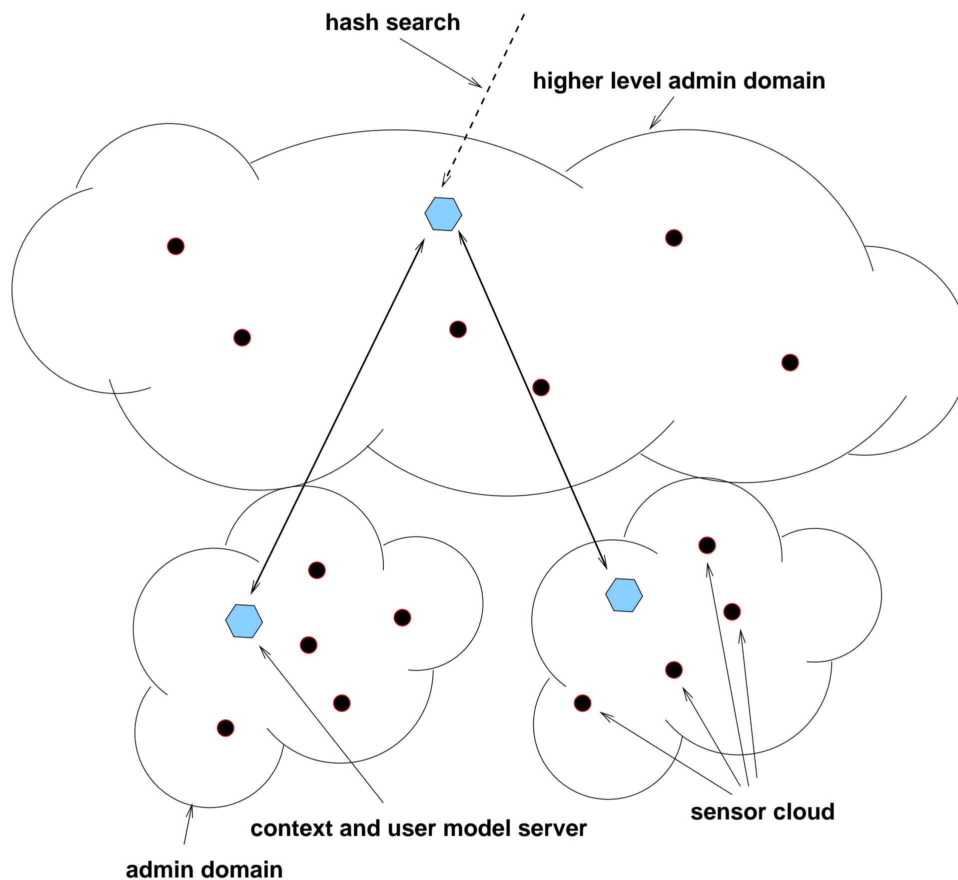

Fig. 2. Context Repository

Where Fig 1 shows the abstraction layers of the architecture, Fig 2 shows the realisation of the architecture in terms of its servers and protocols. Each cloud in Fig 2 represents one administrative domain: this may be a small building, a floor of a large building or the collection of locations that constitute one administrative unit which resides in space scattered over several floors of several buildings. In some cases, where there is a particularly rich environment, a single room may constitute its own administrative domain. This could apply in rooms such as an operating theatre, where there is a complex of activity by several people.

To simplify the description, Fig 2 shows elements for the processes underlying movement of sensor information. (There is a corresponding flow in the opposite direction for output devices.) Within each cloud of Fig 2, there are several sensors, shown as dots. These might be hardware or software sensors. For example, a movement detector is one simple hardware sensor. A simple example of a software sensor is a keyboard activity detector. These operate at the *Sensors* Layer shown in Fig 1.

Each administrative domain has one or more servers, shown as hexagons in Fig 2. Each of these holds part of the *Context Repository* and the *User Model*. Both the context-objects and components of the user model are stored on servers which are distributed across administrative domains.

The exact location of a particular context object is on the server that manages the appropriate administrative domain. For example, if a person's mobile phone was issued by their employer, the context object for that phone would be kept in the employer administrative domain. By contrast, in the case of a person who privately owned a mobile phone, the context object would be kept on their home server.

A corresponding distribution of the *User Model* applies. In this view of the architecture, we again distinguish context information from user modelling information as the later is tightly linked to an individual. So, it needs to meet different demands for privacy and security. Even so, the location is defined in the same general manner as the context objects: a person's private information would logically reside at the their home server while there work-related information may be kept at a server at work. One important difference between the management of context objects and user model components is that we anticipate that people will be far more likely to want to know the location of the parts of their user model. Moreover, people are likely to want to be able to control the choice of this location. Indeed, legislation may require that they be able to do this. These aspects parallel those aspects of the *User Model* discussed in the last section.

In general, a single server would hold a diverse collection of context objects. For example, a simple light sensor object would hold an intensity value. A wireless access point object might contain a list of currently active MAC addresses. Objects can also be containers. For example, a room object might contain a list of objects corresponding to the physical sensors in the room. It may also hold a list of person objects for people currently in the room.

Objects in the repository are updated from the *Context* Layer in response to changes in the environment. For example, a sensor may respond to the detection of a given active badge by updating two objects: the object corresponding to the location; and the object for the active badge.

In addition to this automatic updating of context objects, a program in the *Smart Environment Agent* layer may request notifications. For example, a program may register to be notified when the location attribute of the active badge object changes. This may, in turn, trigger further changes in other objects.

Communication within each administrative cloud and between clouds is carried out using logical multicast. A particular object has a unique ID and is located using a search that starts in the local cloud and expands to adjacent and higher level clouds. After an administratively defined limit on the search is exceeded, the search switches to a distributed hash table algorithm to locate the address of the server that holds the object.

Suppose that the lower left cloud of Fig 2 was a person's main workplace administrative domain and the lower right one was at an administrative domain for another department. If the person was at the other department, the sensors in their main workplace should indicate that they are not present. Sensors in the other department would detect an unknown object and ask their local server for the address of the server for this object. The local server multicasts the request to clouds at the same level and discovers the correct home server for the object. The location information in the object is then updated appropriately.

Fig 2 does not show the *Device Abstraction*, *Device* or *Smart Environment Agent* mapping. In the case of the *Device* Layer, the architecture treats these similarly to the sensors shown in Fig 2. Like the sensors, the devices are associated with an administrative domain.

In the case of the *Smart Environment Agent* layer, such agents can be located anywhere in the computing fabric.

5 Prototype

As a demonstration application we have implemented a tracking system that uses Bluetooth-equipped telephones as a form of active badge. This is an appealing approach since the mobile phones we are using have the characteristics of many common and important classes of sensors and devices. In addition to sensor data similar to an active badge, they can collect images via built-in cameras and they have considerable compute power. They can and do support a range of interaction modalities as well as services. These elements are likely forerunners of devices people will carry in the future.

We have equipped one floor of our building with six Bluetooth access points that are used as sensors for Bluetooth devices. When a Bluetooth equipped phone comes in range of an access point, it is detected. In terms of Fig 1, these Bluetooth access points are the sensors of the prototype. In terms of Fig 2, the floor we have equipped is the local cloud.

Note that we do not depend on the phone and access point making a formal Bluetooth association. We simply use the fact that the access point has detected the presence of the Bluetooth phone. At the context layer of the Fig 1 architecture view, we simply take the Bluetooth MAC address of the telephone as a unique ID for the context repository object that stores location information about the telephone.

The *Context* layer has a sensor agent, which polls the access points every 30 seconds. When a new MAC address is detected, it locates the local context repository server; this is usually cached. It then requests the address of the home server for the object corresponding to that MAC address. Typically the address will be in the local "cloud" or administrative domain since most phones belong to people who work on our floor. If not found in the local cloud, the request is multicast to adjacent clouds and to higher level clouds. After a small number of steps (using a time-to-live value) this process stops and the server reverts to using the distributed hash table algorithm to find the home server for the object.

Once the home server address for the object is found, the sensor agents contacts the server directly. It requests that the object be updated with new location information.

Agents or applications may register with the server and ask to be notified when an object is updated.

In our current prototype, only a single cloud has been implemented. The Chord distributed hash table algorithm has been explored using a simulation. We expect to soon deploy Bluetooth access points at other locations on our campus and on other campuses as well as in homes. These additional points will have associated repository servers and administrative clouds.

The current implementation of the logical multicast uses the Elvin [7] content-based routing system to carry messages between agents and servers.

Our prototype implementation includes an application program that tracks people as they move around our building and shows a floor plan with names appearing and moving as people move. It operates by requesting notification whenever the location field of a set of objects representing mobile phones is updated. It in turn updates the display.

6 Discussion and Conclusions

We began this paper with a discussion of some of the elements of a long term vision of pervasive computing. This took the user's perspective to define some of the requirements for an Intelligent Environment that give the user the benefits of pervasive digital artefacts. We now review these requirements and indicate how the Merino architecture enables this vision.

Merino must support two way flows of data, from sensors to the main computational elements and also, in the opposite direction from the digital artefacts to the devices in the environment. We have shown how Merino does this in Fig 1 and its associated descriptions. Essentially, Merino's levels of abstraction support the management of these information flows.

From the user's perspective, Merino needs to support a ubiquitous intelligent environment which gives users the ability to interact with digital artefacts any where and at any time. This calls for scalable solutions to the distribution of context information, user models and device access information. We illustrated the Merino architecture approach to this in Fig 2 and its associated description.

Merino needs to support context-awareness. This involves the combination of the aspects in Fig 1 with its abstractions and the operation of distributed information flows of Fig 2. The former is critical to the flexible management of context, with the possibility of a rich set of possible interpretations of context data as well as flexibility in the management of information miving to the devices in the environment. The second figure indicates how Merino ensures scalability of context management so that the user can move around their world and still be served effectively.

The final element that we identified was the importance of personalisation and the need for user control over the personal data that will be held as part of the Intelligent Environment. This was a prime concern in the design of Merino and this was reflected in the discussion of both Figs 1 and 2.

The essential layers of the Merino architecture are the core *Sensors* and *Device* layers which reside in the physical environment, collecting information about the user and acting on the environment, with on behalf of the user or in order to interact with the user. At a higher level of abstraction, Merino defines the corresponding *Context* and *Device Abstraction* Layers. Finally, the *Smart Environment Agent* Layer holds the programs that constitute the digital artefacts which are supported by Merino's Intelligent Environment. The *Context Repository* and *User Model* are the persistent stores of the information from the *Context* and *Smart Environment Agent* Layers and it is available to the *Device Abstraction* and *Smart Environment Agent* Layers.

Our current prototype system has implemented parts of the architecture. It implements the information flows from the sensors, at the *Sensors* layer, into the *Context* layer which makes simple interpretations of the sensor information and initiates messages with context information to go to the *Context Repository.* We have build one application, a simple people tracking interface.

Our description of the Merino architecture has not addressed a number of important issues. For example, when a new device is attached to the network it must be able to configure itself and find its local context and user model server. In our current prototype we hand configure the sensors. In the future, we plan to use ZeroConf to acquire an IP address and multicast DNS for server discovery. We have considered the Jini system [12] for this but while Jini has the capability to dynamically add and delete services in a changing network infrastructure, it does this using Java API signatures for resource discovery. This scales very well in small to medium size home and corporate networks. It does not scale so well in heterogenous, distributed environments that are part of multiple spheres of control.

Another area we have not discussed concerns the characteristics of the messaging protocol used to update context objects. In our prototype we use the

Elvin message routing system as a transport layer for messages. The question of ordering semantics for update messages is simplified by the use of an *accretion* approach to modifying the context objects.

We believe that the Merino architecture provides a framework for managing context data that enables sophisticated pervasive computing services in a scalable manner.

References

1. Weiser, M.: The computer for the twenty-first century. Scientific American **265** (1991) 94–104
2. Kidd, C.D., Orr, R.J., Abowd, G.D., Atkeson, C.G., Essa, I.A., MacInyre, B., Mynatt, E., Starner, T.E., Newstetter, W.: The Aware Home: a living laboratory for ubiquitous computing research. In: Proceedings of the Second International Workshop on Cooperative Buildings, Pittsburgh, USA (1999)
3. Salber, D., Dey, A.K., Orr, R.J., Abowd, G.D.: The Context Toolkit: aiding the development of context-enabled applications. In: Proceedings of the 1999 Conference on Human Factors in Computing Systems, Pittsburgh, PA (1999) 434–441
4. Brown, E.S.: Project Oxygen's new wind. Technology Review (2001)
5. Phillips, B.: Metaglue: A programming language for multi-agent systems. Master's thesis, MIT, Cambridge, MA (1999)
6. Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B.: People, places, things: Web presence for the real world. Technical Report HPL-2000-16, HPLabs, http://www.cooltown.hp.com/ (2000)
7. Fitzpatrick, G., Mansfield, T., Kaplan, S., Arnold, D., Phelps, T., Segall, B.: Instrumenting the workaday world with Elvin. In: Proceedings of the ECSCW'99, Copenhagen, Denmark, Kluwer Academic Publishers (1999) 431–451
8. Fitzpatrick, G.: The Locales Framework: Understanding and Designing for Cooperative Work. PhD thesis, Department of Computer Science and Electrical Engineering, The University of Queensland (1998)
9. Dey, A., Abowd, G., Salber, D.: A context-based infrastructure for smart environments. In: Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE'99), Dublin, Ireland (1999) 114–128
10. Grimm, R., Anderson, T., Bershad, B., Wetherall, D.: A system architecture for pervasive computing. In: Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark, ACM, SIGOPS 2000 (2000) 177–182
11. Kantor, M.: Creating and Infrastructure for Ubiquitous Awareness. PhD thesis, University of California, Irvine, California, USA (2001)
12. Waldo, J.: Jini architecture overview. Technical report, Sun Microsystems, Inc. (1998)