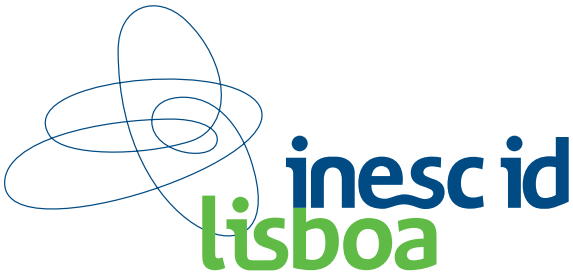

Technical Report RT/22/2011

Distributed Context-Aware Systems

Pedro Alves
INESC-ID/IST/Opensoft
pedro.h.alves@ist.utl.pt

Paulo Ferreira
INESC-ID/IST
paulo.ferreira@inesc-id.pt

May 2011

Resumo

Context-aware systems take into account the user current context (such as location, time and activity) to enrich the user interaction with the application. In the last decade, this topic has seen numerous developments that demonstrate its relevance and usefulness, a trend that was accelerated with the recent widespread availability of powerful mobile devices (such as smartphones) that include a myriad of sensors which enable applications to capture the user environment for a large number of people.

This paper examines the specific issues that occur in distributed context-aware systems, where the captured contextual information may travel a long distance before being used in an actual application. To better understand those issues, we propose a taxonomy derived from four layers that can typically be found in these systems: Capture, Inference, Distribution and Consumption. The common principles outlined in the taxonomy are then applied to some real applications, discussing the advantages and disadvantages of each approach.

Distributed context-aware systems

Pedro Alves
INESC-ID/IST/Opensoft
pedro.h.alves@ist.utl.pt

Paulo Ferreira
INESC-ID/IST
paulo.ferreira@inesc-id.pt

2011-05-17

Abstract

Context-aware systems take into account the user current context (such as location, time and activity) to enrich the user interaction with the application. In the last decade, this topic has seen numerous developments that demonstrate its relevance and usefulness, a trend that was accelerated with the recent widespread availability of powerful mobile devices (such as smartphones) that include a myriad of sensors which enable applications to capture the user environment for a large number of people.

This paper examines the specific issues that occur in distributed context-aware systems, where the captured contextual information may travel a long distance before being used in an actual application. To better understand those issues, we propose a taxonomy derived from four layers that can typically be found in these systems: Capture, Inference, Distribution and Consumption. The common principles outlined in the taxonomy are then applied to some real applications, discussing the advantages and disadvantages of each approach.

Contents

1 Introduction

While looking for better ways to achieve efficient communication and coordination on geographically dispersed teams, we have found that traditional mechanisms such as Instant Messaging and E-Mail are not enough to accomplish that goal. No one denies that these mechanisms and other forms of direct communication[1] have seen a tremendous evolution in the last years and are now essential for modern team collaboration; however they still lack several characteristics of face-to-face communication often found on co-located teams. For example, it ought to be possible, for every team member, to become aware of what is going on: who belongs to the team, who is online, who is working on what, who is responsible for what, etc. [Gutw 05]. This can be accomplished using direct communication where members continuously update their current status but since it requires too much effort from them, it usually ends up being dismissed. However, dismissal of such information is not an option, as it has been proven multiple times to be essential to efficient knowledge sharing and learning [Dour 92][Bjer 03]. To better understand this impact, we can refer to Sawyer's work [Sawy 98], who found that social processes such as informal coordination and ability to resolve intragroup conflicts accounted for 25% of the variations on software development quality.

It should be clear now that direct communication needs to be complemented with other communication forms, but which forms are those? Which forms will allow us to propagate to all team members this indirect, almost surreptitious knowledge? Research conducted in the last decade suggests that the answer may lie in context-aware applications [Schi 94], which are able to adapt their operations to current context without explicit user intervention. This context can be defined by a multitude of variables such as location, time or even emotional state. For example, a GPS-enabled device that is carried by every team member is sufficient to provide continuous location awareness without user effort (that is, without the need of direct communication). With increasing availability and decreasing cost of this kind of devices, it is expected that context-aware applications become more common in many areas, including team collaboration. Still, these devices (also known as sensors) can only go so far in this process - although they are able to capture context, they do not know how to distribute it among interested parties.

For most context-aware applications, sensors provide information either for personal consumption (e.g., tourist virtual guides [Chev 99]) or for community anonymous data gathering for statistical purposes (e.g., traffic information or health habits [Redd 07]). In both cases, the propagation of context is simple, often using a direct channel between the sensor and the data consumer. However, propagating context among the members of a group is a much different (and harder) problem, because we are entering the realm of distributed systems protocols such as broadcasting and publish-subscribe. In fact, with these distributed context-aware systems, we face the classical scalability problem of N producers (sensors) transmitting information to M consumers (client applications) with some aggravating factors like highly dynamic environments, different computing capabilities, user mobility and privacy issues [Lane 10]. Recent increased interest in social software applications [Wang 07] makes this challenge even more relevant.

Traditionally, distributed context-aware systems have been analyzed more from an usability perspective and not so much from the resource usage or scalability view point, but we believe this is bound to change. The success of early experiments with small groups will necessarily lead to experiments with larger or more dispersed groups, using a much larger quantity of sensors. Nothing prevents massive sensor networks that

[1] Communication which is explicitly initiated by someone and whose recipient is known, be it an individual or a group.

are now mainly used for environment monitoring and disaster prevention to be deployed in multi-national companies producing huge amounts of data, invaluable for learning and knowledge sharing. The underlying foundation is already in place: modern mobile phones (e.g, iPhone, Android) include a multitude of sensors ranging from accelerometers to proximity sensors (although still mostly ignored by the majority of the applications). A growing number of people maintain a virtual presence through most parts of the day, by logging in on their web-based email account or just browsing and commenting information online (many times within their social network, a major source of context data) just to mention a few.

All this information must often travel across multiple networks and servers, until it reaches their recipients. A clear vision of what is going on "in the pipes" is needed to understand the surrounding issues for distributed context-aware systems. Only then, we are able to improve upon them and envision more efficient architectures and protocols for this kind of applications. This requires to specifically analyze distributed context-aware systems and their unique problems and challenges. This article analyzes common principles of these systems and presents a taxonomy for comparing and contrasting them.

1.1 Other surveys

[Henr 05] includes a survey of middleware for context-aware systems. The article starts by describing common requirements for middleware in context-aware systems: *support for heterogeneity*, *support for mobility*, *scalability*, *support for privacy*, *traceability and control*, *tolerance for component failures* and *ease of deployment and configuration*. The authors then analyze some middleware solutions (e.g. Context Toolkit [Salb 99], Context Fusion Networks [Kotz 04]) with respect to the above mentioned requirements, but fail to provide a critical analysis of pros and cons of each solution. Also, these requirements are not design options but rather features that the middleware solution has already implemented or will implement in the future. For example, scalability is an obvious requirement for every distributed application, but what are the options to achieve it? The article lacks this analysis.

[Brat 07] is similar to [Henr 05] in the way it focus on certain *non-functional requirements* such as distribution, privacy and fault tolerance, comparing several context-aware frameworks (not applications) regarding these requirements. However, it doesn't show neither the possible options for implementing each requirement nor does it explain why certain requirements are met and others do not.

[Bald 07] starts with a layered model (Sensors, Raw Data Retrieval, Preprocessing, Storage/Management and Application) to explain the challenges found on each layer on a typical context-aware system. For example, according to the authors, the *Processing* layer is responsible for (1) reasoning and interpreting contextual information; (2) aggregating or composing different context data sources; (3) manage sensing conflicts that might occur when using multiple context data sources. It is also referred that this layer can be implemented directly in the application or in the context server and that the context server option may increase network performance and save limited client resources. It is also noteworthy that the *Storage/Management* layer provides two options for clients who want to access context: *synchronous* and *asynchronous*. In synchronous mode, the client is polling the server for changes while in asynchronous mode the client subscribes to the events it is interested in, and on occurrence of these events it is notified by the server. The article also classifies the possible options for modeling context, that is, for defining how context data is represented in a machine processable form. These options are *Key-Value Models*, *Markup Scheme Models*, *Graphical Models*, *Object Oriented Models*, *Logic Based Models* and *Ontology Based Models*.

Finally, it analyzes and compares the main context-aware frameworks with respect to architecture (how the different components are distributed), sensing (how sensor data is obtained), context model (how context data is represented), context processing (how higher-level data is inferred from raw sensor data), resource discovery (how components of the system are discovered, mainly sensors), historical context (if context can be stored for later analysis, like trends) and security and privacy (if it provides mechanisms for guaranteeing security and privacy of its users' data).

Contrary to other previous studies, in this survey we describe the possible options for context-aware application developers to choose on each layer instead of just enumerating the requirements. However, some of the compared dimensions are too abstract or mix different topics making it difficult to obtain relevant insights. For example, when comparing the architecture of the different systems, the authors of other surveys use the term *centralized middleware* as opposed to *blackboard model* or *widget based* (which can also be used in centralized middleware approaches). In fact, *blackboard model* pertains to how clients access the information and *widgets* are an abstraction on top of sensors - neither one is an architectural approach.

2 Basic Definitions

This section introduces some basic definitions related to context-aware systems that are useful in the following sections.

2.1 Context

The word "context" is subject to multiple interpretations and has been researched in various fields like psychology, philosophy and computer science [Bolc 07]. In the last field, specifically in the area of Computer Supported Collaborative Work (CSCW), context was initially perceived as user location [Schi 94, Brow 97] but, in the last years, it has been enriched with other sources of information such as identity, activity and state of people, groups and objects [Salb 99].

Still, the various definitions of context are usually synonyms of *environment* or *situation* which makes them difficult to apply in practice. In his seminal paper, Dey [Dey 00] came up with a definition of context that remains, to this date, one of the most accurate:

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*

By considering only the information that is relevant to the interaction between a user and an application, application developers can focus on a subset of the user environment data for a given application scenario. For example, the context needed by UbiqMuseum [Cano 06] to assist museum visitors is their location and native language. Other environmental information such as their body temperature or their marital status is not relevant to the interaction with UbiqMuseum.

Satyanarayanan [Saty 01] refers to context in its relation with pervasive systems that, with minimal intrusion, must be cognizant of its user's state and surroundings, and must modify its behavior based on this information. A user's context can be quite rich, consisting of attributes such as physical location, physiological state (such as body temperature and heart rate), emotional state (such as angry, distraught, or calm), personal history, daily behavioral patterns, and so on.

Chen [Chen 00], not satisfied by a general definition, defines context by enumerating examples of contexts:

- Computing context, such as network connectivity, communication costs, and communication bandwidth, and nearby resources such as printers, displays, and workstations.
- User context, such as the user's profile, location, people nearby, even the current social situation.
- Physical context, such as lighting, noise levels, traffic conditions, and temperature.
- Time context, such as time of a day, week, month, and season of the year.

These types of context are further refined by Dey [Dey 00]. He considers that there are certain types of context that are, in practice, more important than others for characterizing the situation of a particular entity: **location**, **identity**, **activity** and **time**. These context types not only answer the questions of who, what, when, and where, but also act as identity indexes into other sources of contextual information. For example, given a person's identity, we can acquire many pieces of related information such as phone numbers, addresses, email addresses, birth date, list of friends, relationships to other people in the environment, etc. With an entity's location, we can determine what other objects or people are near the entity and what activity is occurring near the entity. This characterization helps designers to choose which context to use in their applications, structure the context they use, and search out other relevant context.

2.2 Context-Aware Systems

Given the characterization of *context* in the previous section, we now describe how systems use that *context*.

Schilit [Schi 94] defined context-aware systems as systems that *adapt* themselves to context. He claims that it is not enough to be informed about context. However, some authors say otherwise [Pasc 98]. In fact, there is a classical debate in this area between "use context" advocates and "adapt to context" advocates. Although research has been more active on systems that adapt to context, even an application that simply displays the context of user's environment to the user can be considered context-aware even though it is not modifying its behavior.

Dey [Dey 00] tried to conciliate both views with the following definition:

> *A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.*

In practice, this definition results in three main features that context-aware systems may provide:

- ***presentation* of information and services to a user** - systems that provide information to the user augmented with contextual information (e.g., phone's contact list enhanced with location information) or that provides services based on the current user's context (e.g., show me restaurants nearby);

- **automatic *execution* of a service** - systems that execute a service automatically based on the current context (e.g., automatically updating my status on a social network based on accelerometer data - sleeping, walking, running);
- ***tagging* of context to information for later retrieval** - systems that are able to associate digital data with the user's context (e.g., virtual notes that are attached to certain locations, for other users to see).

It is noteworthy that a context-aware application doesn't actually determine why a situation is occurring, but the designer of the application does. The designer uses incoming context to determine why a situation is occurring and uses this to encode some action in the application.

2.3 Distributed context-aware systems

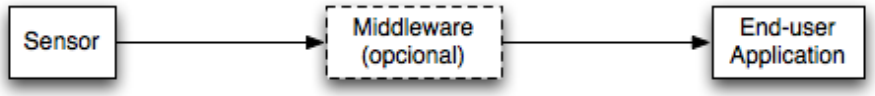

Figure 1: Typical context-aware architecture

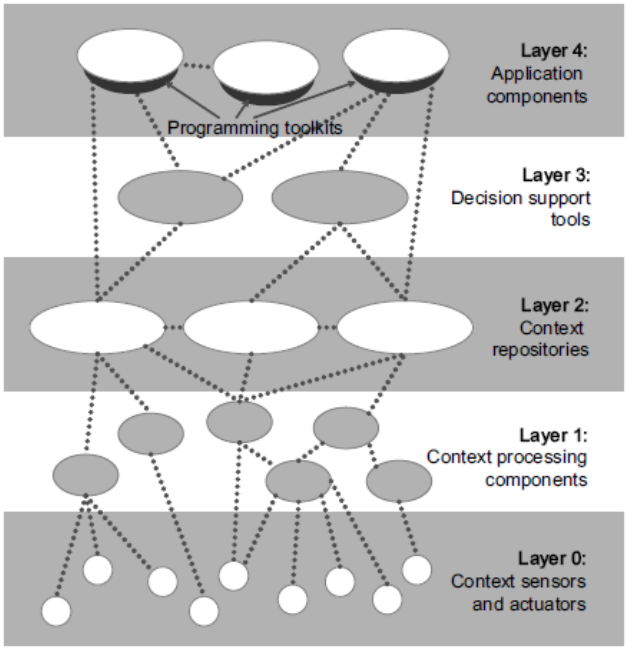

Figure 2: Layers of a context-aware application according to Henricksen

Context-aware systems can be described as *end-user applications* that use context information provided by *sensors*. From an architectural point of view, these two layers (sensors and end-user applications) are

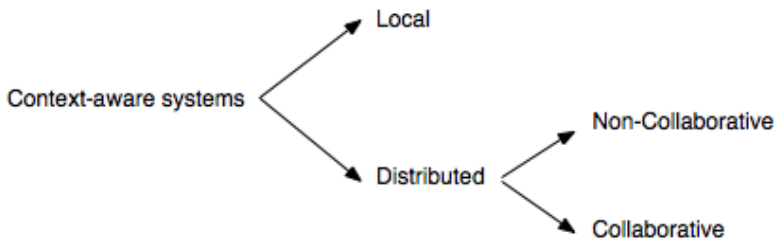

Figure 3: Types of Context-aware systems

mandatory, as they are the producers and consumers of context information. Between them, there is often a *middleware* layer to address communication and coordination issues between distributed components [Henr 05] (see Figure 1).

In simple context-aware systems, the end-user application communicates directly with the sensor (e.g., a cellphone that automatically switches off GPS when battery is below a certain capacity), removing the need for an intermediate layer. This was the case in early systems which were no more than 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. Henricksen [Henr 05] enumerates the following components, as shown in Figure 2:

- context sensors and actuators that provide the interface with the environment, either by capturing it (sensors) or by modifying it (actuators) (layer 0);
- 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);
- application components that are integrated in client applications using programming toolkits (layer 4).

Context-aware systems can be categorized into two large groups: local and distributed (see Figure 3). Local systems are systems in which sensors and applications are tightly-coupled (usually through a direct physical connection). For example, a cellphone application that sets the mode to silent while its owner is running is a local system, since the accelerometer that provides the "running" context is directly attached to the cellphone as well as the application that uses that context to activate the silent mode.

On the other hand, distributed systems do not have a direct physical connection between the sensor and the application. As a consequence of that loose coupling, it is possible to have multiple applications receiving information from the same sensor. Also, it is possible that multiple dispersed sensors produce information to be consumed by a single application. For example, a cellphone may broadcast to a group of friends that its owner is currently walking, to decrease the probability of incoming calls during that period. In this case, the accelerometer is not tightly coupled to the recipient of its information.

We can further divide distributed systems into two types: collaborative and non-collaborative. Distributed collaborative systems are systems that help two or more dispersed humans accomplish a common goal. For example, MyVine [Foga 04] provides real-time availability information within a group of colleagues, using speech detection, location, computer activity and calendar entries. Here, the common goals is team synchronization (actually, team synchronization is the most common goal of distributed collaborative systems). In contrast, non-collaborative systems support only individual goals. For example, UbiqMuseum [Cano 06] provides context-aware information to museum visitors. A portable device is provided to the visitors that, based on their current location and individual profile, shows relevant information in their preferred language. In this system, the goal is individual (the device shows information that is only relevant to its user) but the system is distributed since the location is inferred from Bluetooth emitters carefully dispersed throughout the museum.

In this article, we are focused in particular issues often found in distributed context-aware systems, as already mentioned in Section 1.

3 Taxonomy

As we explained in Section 1.1, the existent taxonomies for distributed context-aware systems are not well-suited for guiding the architectural decisions of application developers. We believe the main challenge of developing these systems is related to how the different layers communicate with each other and, mainly, how to overcome the problems that arise when these layers are spread across a distributed system. Thereby, we propose a taxonomy that:

- Clearly categorizes the architectural options available to the application developer, explaining the advantages and disadvantages of each approach;
- Analyzes the problems that are specific to distributed context-aware systems, whose (distributed) components have to communicate with each other, and how that affects availability and scalability of such systems;
- Exemplifies each option with real applications that were deployed and evaluated by end users, instead of relying on generic toolkits, frameworks and prototypes.

To develop this taxonomy, we divide the analysis into the four layers that we can find in the majority of these systems. These layers are traversed by context data, from the moment it is acquired from sensors in raw format to the moment it is consumed by the end-user application, as we can see in Figure 4.

First, context data is **captured** from the environment using sensors. This data is usually too detailed to be used directly by end-user applications, so a **context-inference step** is needed to obtain higher-level

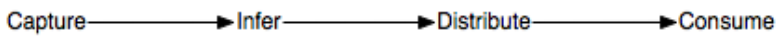

Figure 4: Layers of a context-aware system

aggregated data. For example, a GPS device captures geographical coordinates from which a place (city, building, etc.) is inferred. This step is also known as *feature extraction*. As the name implies, distributed context-aware systems have to deal with **distribution of context data** among its components in an efficient and scalable manner. Finally, client applications **consume** this information in order to provide relevant services to their users. Note that we intentionally left out the often referred in literature *storage layer* because, in this study, we want to focus on how context data is propagated. We now describe in more detail the available options to choose from in each of these layers.

3.1 Capture

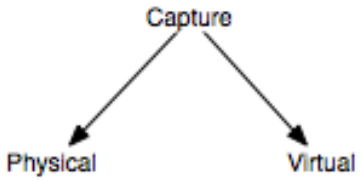

Figure 5: Capture layer

The Capture layer is responsible for acquiring context data from the environment, using sensors. Note that the word "sensor" not only refers to sensing hardware but also to every data source which may provide usable context information [Bald 07]. Sensors have been traditionally classified in two dimensions that, although differently named, are very similar. Prekop [Prek 03] call these dimensions *external* and *internal* and Hofer [Hofe 02] refers to *physical* and *logical*. We adopt the names proposed by [Indu 03], *physical* and *virtual* sensors, as depicted in Figure 5.

Physical sensors are hardware sensors capable of capturing physical data such as light, audio, motion and location. Location is, by far, the most researched type of physical sensor with numerous published studies from which we highlight the work of Hightower [High 01] and Indulska [Indu 03]. The later presents the following taxonomy:

- **Proximity vs Position** - Position sensors provide the location of an entity with coordinates (e.g. the latitude and longitude of a GPS system), while proximity sensors provide the location within a region (e.g. mobile phone cells). Note that both modes have different accuracy levels, ranging from centimeters to tens of meters.

- **Line of sight** - Some location sensors require a clear line of sight between them and the associated infrastructure. For example, GPS sensors need a clear line of sight to multiple satellites, which prevents these devices from being used inside buildings. Systems built upon communication mechanisms that

can cross clothes and walls place fewer constraints on device and infrastructure placement, but may not be appropriate when walls are, in fact, an important aid to infer the location (e.g. the room a certain person is in). This is similar to the distinction made by [Chen 00] on "Outdoors vs Indoors vs Hybrid systems".

- **Complexity trade-off** - Typically, location devices work coupled to an infrastructure (e.g., GPS devices and satellites), and their complexity is inversely proportional to the infrastructure's complexity. For example, a GPS device can be considered a simple device (if we consider only the receiver) but needs a complex satellite infrastructure. On the other hand, a location system based on multiple uncoordinated base-stations or beacons like Cricket [Priy 00] has a simpler infrastructure but the device has now the responsibility of computing the position.

- **Identification** - Many sensor devices incorporate some unique ID that must be transmitted to the associate infrastructure to infer their location (e.g. Wifi) raising privacy and ethical issues. Sensors that compute the location on the device itself (e.g. Cricket [Priy 00]) allow greater end-user control over publishing their location in the system.

Virtual sensors acquire context data from software applications, operating systems and networks [Indu 03]. Detecting new appointments on an electronic calendar or watching the file system for changes are examples of virtual sensors. These sensors can also be used to infer *location*. Indulska [Indu 03] gives some examples like using a travel-booking system or the IP of the active device[2] to perceive where the user is currently located. Although virtual sensors have not been subject to the same level of research of their physical counterparts, they offer a promising alternative as people spend increasing time using computers, smartphones and similar devices, where their identity, location and activity may easily be tracked with simple software. Actually, it is usually cheaper to develop and deploy a virtual sensor than a physical one, since the required infrastructure is already in place.

The literature also refers to a third type of sensor: the *logical sensor*, which combines physical or virtual sensing data with information from other sources (like databases) in order to produce higher-level context data [Indu 03, Bald 07]. We think the distinction between logical sensors and context inference is not very clear and prefer to include in this layer only sensors that capture information in raw format, without further processing. Indulska [Indu 03] argues that a distinction is made between logical sensors and the fusion of sensor data. According to him, logical sensors work with data from particular sensor systems and, e.g., do not try to resolve conflicts. We still think this is a weak distinction because since, by definition, logical sensors gather data from multiple sources, it is impossible to guarantee that there won't be any sensing conflicts.

Different sensors can provide different types of context. According to Dey [Dey 00], the most important types of context are: *location*, *identity*, *activity* and *time*, corresponding roughly to the primal questions: *where*, *who*, *what* and *when*. When designing a context-aware system, it is important to know which types of context the application will want to observe and use the appropriate sensors. In Table 1 we show examples of physical and virtual sensors grouped by the type of context they are able to capture.

In Table 2, we show some context-aware applications and their corresponding sensors. Table 2 reflects a general trend in context-aware applications: while early applications were predominantly based on location

[2]The device where user's activity was last detected

	Physical sensors	Virtual sensors
Location	*Outdoor*: Global Positioning System (GPS), Global System for Mobile Communications (GSM); *Indoor*: Bluetooth, 802.11 cells	Networked calendar system, Travel-booking system, User's login on location-aware computer, IP subnet
Identity	*Based on something you are*: Fingerprint reader, retina scanner, microphone; *Based on something you have*: smartcard reader, RFId	Various authentication schemes at the operating system or application level
Activity	Mercury switch, accelerometer, motion detector, thermometer, UV-sensors, camera	Keyboard or mouse activity, application usage
Time	Clock	Operating system timer

Table 1: Types of context and corresponding physical and virtual sensor

sensors, researchers have been recently using other types of sensors to infer richer context information like user activities ("walking, running") and even mood ("sad", "happy") [Sant 09].

3.2 Infer

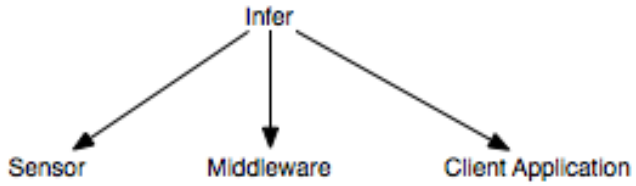

Figure 6: Infer layer

The *Context-Inference* layer, also known as the Preprocessing layer [Bald 07], is responsible for reasoning and interpreting raw context information provided by sensors on the Capture Layer. Most of the times, sensorial information is too fine grained, with too much detail for the needs of end-user applications. A classical example is location information provided by GPS devices. Generally, end-user applications do not need to know the exact latitude and longitude of an entity, being much more interested in knowing the place (city, street, etc.) in which the entity is located. A transformation is needed to reach a higher level of abstraction like transforming GPS coordinates into the name of a street. This transformation is commonly known as context inference, because it usually involves some kind of reasoning. Some authors [Schm 99] introduce a sub-layer called *feature extraction*. *Feature extraction* refers to the act of cataloging raw sensorial data into relevant features (e.g. from the readings of a luminosity sensor extract level, flickering, wave-length, etc.). Further inference can be made using these features. [Zand 10] uses the term "context provider" as something that convert any kind of input data (either sensorial or web-based content) to an RDF-based context description.

	Description	Sensors used
GUIDE [Chev 99]	Information for city visitors	Wifi (location)
[Welb 05]	Mode of transit: Walking, running, riding a vehicle	Clock, GSM/Wifi (location), Accelerometer
ContextContacts [Oula 05]	Enhanced phone's contact list	GSM, Phone activity, Bluetooth (nearby environment)
UbiqMuseum [Cano 06]	Assists museum visitors	Bluetooth (location)
AwareMedia [Bard 06]	Support coordination at an operation ward	Bluetooth (location), video camera, shared calendar
[Stie 06]	Help workers perform critical and complex assembly tasks in a car production environment	Body-worn, car-mounted and tool-mounted accelerometers
BikeNet [Eise 07]	Cyclist experience mapping	Magnetometer, Inclinometer, Speedometer, Microphone, GPS, GSR Stress Monitor, CO^2 meter
CenceMe [Milu 08]	Social activities (dancing, lunching)	Microphone, GPS, Camera, Bluetooth (nearby environment), Accelerometer
SoundSense [Lu 09]	Music events' sharing	Microphone, Camera, GPS
Upcase [Sant 09]	Daily activities (working, driving, sleeping, resting, walking outside)	Luminosity, Microphone, Temperature, Accelerometer

Table 2: Some context-aware applications and corresponding sensors

Also, this layer is normally associated with classification techniques, mostly borrowed from Artificial Intelligence algorithms, like Kohonen Self-Organizing Maps (KSOMs) [Van 01], k-Nearest Neighbor [Van 00] and Neural Networks [Rand 00]. These techniques have been successful applied to some context domains such as inferring the user activity (walking, running) from an accelerometer [Welb 05].

Context inference techniques is an active topic of research which we do not detail here. Since we are analyzing *distributed* context-aware systems, we classify these systems based on *where* the inference occurs.

Usually, the type of information inferred from sensorial data is specific to each application. For example, consider the data provided by an accelerometer. An application might use that information to know whether a user is walking or running [Welb 05], while another may be more interested in detecting screw tightening [Stie 06]. In these cases, it makes sense to move the inferring task to the application because of the specific semantic needs it tries to accomplish. However, transforming raw sensorial data into high-level context information can be a resource demanding task, unsuitable for applications that run on constrained devices like cellphones. To avoid this problem, some systems use a middleware component (e.g. SOLAR [Chen 02], PACE [Henr 05]), running in a machine with higher capability, that is responsible for context inference. In these systems, the processing is moved off the applications into a server, allowing the use of very basic devices for client applications. In other systems, the sensors are capable of inferring high-level context data

themselves. For example, the Activity widget provided by the Context Toolkit [Salb 99] senses the current activity level at a location such as a room, using a microphone. Instead of producing raw audio data captured by the microphone, it provides a high-level attribute "Activity Level" with three possible values: *none*, *some* or *a lot*.

In summary, context may be inferred in the sensor, in a middleware component or in the end-user application. These three locations, depicted in Figure 6, are intimately related to the following properties of a context-aware system:

- **Network bandwidth consumption** - Since context inference transforms fine-grained sensorial information into coarse-grained high-level data, it effectively reduces the amount of information needed to represent a context message. Thus, moving the inference layer closer to the sensor results in less network bandwidth consumption. This is more significant in distributed context-aware systems with devices dispersed along a WAN, unreliable or low-bandwidth connections, etc.

- **Complexity (CPU/Memory consumption)** - As already noted, context inference mechanisms can lead to high CPU and RAM consumption, specially when sophisticated AI algorithms like Neural Networks [Widr 94] or Decision Trees [Quin 86] are used. Even if the device is capable of computing the information, it can lead to excessive and unsustainable battery consumption. Since resource constrained devices are, in these cases, unsuitable for context inference, the developer has to deploy the inference engine in a resourceful machine like a server. Most physical sensors are attached to low-capability devices (to increase mobility) and do not provide context-inference mechanisms. However, virtual sensors often run on servers or desktop computers (where they can access virtual context information from databases, shared calendars, etc.), so they are able to provide higher-level context information than their physical counterparts.

- **Reusability** - Context inference reusability makes sense for context types like location, where the output is sufficiently *standard* to be used by multiple applications. For example, inferring the street name from geographical GPS coordinates is a recurring requirement in location-aware systems and doesn't make sense to (re)implement in every end-user application. Similarly to network bandwidth, moving the inference layer closer to the sensor increases reusability. In fact, reusability is referred as one of the main benefits of Context Toolkit' Widgets [Salb 99].

- **Personalization** - Somehow opposed to reusability is the ability to personalize the inference engine to better suit individual needs. For example, CenceMe [Milu 08] is a phone application that allows its users to associate a certain movement (like drawing an imaginary circle with the phone) to some meaning or activity (e.g., going to lunch). Although it is possible to personalize a context engine outside the application, such system would suffer from scalability issues trying to cope with the individual needs of its users. Another important issue related to personalization is the mediation of ambiguity [Dey 02]. Sometimes, context inference includes dealing with ambiguous data and explicit user mediation is needed (the application prompts the user to resolve ambiguity) [Foga 07], again adapting the inference mechanism to suit individual needs.

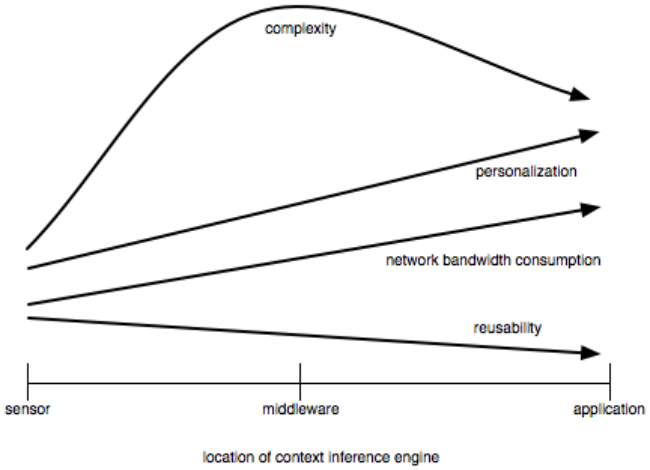

Figure 7: How context-aware systems properties are affected by the location of the context inference engine

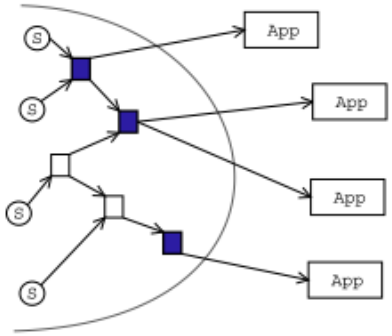

Figure 8: The Solar system pushes application-specific processing into network proxies (represented by dark squares) that can be combined and reused by other applications (S=Sensors; White squares=Generic context processors)

Figure 7 summarizes how these properties are affected by the location of the context inference engine. Moving the context inference from the sensor to the application achieves better personalization but higher network bandwidth consumption and decreased reusability. The most complex inference engines should be moved into the middleware where they have access to better hardware resources.

Some systems use an hybrid approach where the inference engine reside in multiple places. Miluzzo [Milu 08] proposes a split-level classification for its CenceMe system, pushing some classification to the phone and other to the backend servers (middleware). The phone's classification output is sent to the server which then applies a second more complex classification. This design achieves a good balance between network bandwidth consumption (which is reduced because the phone sends already classified information instead of raw data) and CPU/memory consumption (complex classification is moved off of the phone).

Chen [Chen 02] proposes an interesting solution to achieve personalization without sacrificing reusability. In the Solar system, applications can push its application-specific processing into the network as a proxy. These proxies run on servers in the network and form an *operator graph*, where multiple sensors can feed the

proxies which can be combined with other proxies and reused by multiple applications, as shown in Figure 8. The Solar middleware is, in fact, composed of multiple application-specific inference engines developed in a modular way such that multiple applications can combine and reuse them to satisfy their requirements.

3.3 Distribution

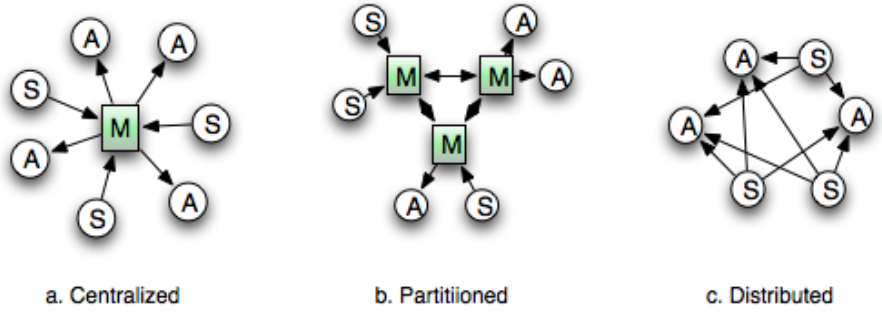

Figure 9: The three types of distributed organization for context-aware systems (M: middleware; S: sensor; A: application)

Independently of how context was captured or inferred, it has to be distributed among the system components (sensors, middleware, application). Some articles refers to this distribution scheme as the system *architecture*, although the term *architecture* usually comprises other things such as the system modules, repositories, sub-layers, etc. Here we focus solely on how context is distributed in the system.

The most common approach for distributed context-aware systems uses a centralized middleware to broker between dispersed sensors and client applications (see Figure 9a). For example, CenceMe [Milu 08] shares personal context obtained from mobile phones though an HTTP server. All users of the CenceMe application rely on this server to send and receive context information to/from other users. This approach is useful to relieve resource-constrained devices from cpu and memory-demanding tasks and simplifies the communication since the devices (sensor or application) only need to establish a channel between them and a single component (the server). However, it provides a single point of failure and thereby lacks robustness. Even if the server doesn't fail, its scalability is limited - as more devices use the system, its performance will start degrading. Also, it is not well fitted to scenarios where network connection is intermittent such as mobile phones. Unless the application uses local caching mechanisms, it may stop working when the connection with server drops.

Some systems try to solve some of the above mentioned problems by using multiple servers [Fitz 99, Carz 01]. In this approach, usually associated with a partitioned or federated architecture (some authors call it an *acyclic peer-to-peer architecture*), each device still communicates with only one server, but different devices may use different servers, as shown in Figure 9b. The servers communicate directly with each other, propagating information from devices in one partition to the other partitions. This distributes the load among the servers, increasing the system scalability. For example, the ContextContacts application [Oula 05], which enhances the traditional cellphone contact list with status information such as location and phone speaker state, uses the XMPP partitioned infrastructure to distribute this information. There are multiple XMPP

servers (one for each domain) and the users of each domain connect to the associated server, which then propagates information to the other servers. The main challenge on partitioned systems is dividing its users. For example, XMPP assumes each user id is like an email address (e.g. "alice@wonderland.com") from which it can infer the domain and the associated XMPP server. But other systems may use different user identifications from which deriving a partition is much harder. Also, context transmission delay may increase due to another hop in the communication path (the server to server communication). Note that, as in the centralized model, the lack of redundancy constitutes a limitation in assuring connectivity, since the failure of a server isolates all the devices in that server's partition. A promising approach that is particularly relevant to location-aware systems is the use of geographical partitions. Under this approach, each server is responsible for a geographical region. For example, Chakraborty [Chak 07] is researching how to evolve BusinessFinder from the current centralized model to a geographically partitioned model. BusinessFinder, a "Yellow Pages" application which searches vendors nearby the current user location, is particularly suited to this model, since in most cases the client and the vendor will share the same server and no extra hops will be necessary to propagate context.

Finally, some systems use a completely distributed middleware through a peer-to-peer topology [Jova 01], where each device communicates directly to the other devices (see Figure 9c). In this case, the middleware is usually embedded in the device itself. Peer-to-peer systems do not suffer from the single point of failure, since each device acts as a client and a server. Also, it is more resilient to network problems as there are multiple paths to communicate. For example, the Hydrogen framework [Hofe 02] proposes an architecture for mobile devices in which local context information is combined with remote context from nearby devices. This *context sharing* can be used to pair two devices with complementary information like a thermometer and a GPS receiver communicating through Bluetooth. However, these systems have to employ complex algorithms to ensure that context information produced by the sensors is delivered to all interested applications. Contrary to centralized and partitioned models, the components of these systems are in a constant discovery process of new sensors and applications that wish to communicate with them. Also, depending on the routing algorithm, the time necessary to propagate context may increase when multiple hops are needed to connect the source and the destination nodes.

Context-aware systems that follow a decentralized (peer-to-peer) architecture are usually related to a geographical distribution of their nodes in order to provide efficient location-based services. Some examples of these systems include GHT [Ratn 02], GeoPeer [Arau 04] and IGM [Cowz 09]. The big advantage of these systems over traditional peer-to-peer architectures is that their nodes know their location and the location of nearby nodes thus allowing efficient geographical routing. Araujo [Arau 04] proposes two context-aware applications that can be easily implemented on top of GeoPeer: (1) *Geographically-scoped multicast*, a service that consists in disseminating a notification to all nodes located inside a given geographic region (e.g., an alarm about some natural disaster) and (2) *Geographically-scoped queries*, a service used to collect information from nodes inside a given geographic region (e.g., environmental monitoring of geographical areas by connection of the relevant sensors to the GeoPeer nodes). Note that, in these systems, operations are not limited to a user local region, therefore users can perform operations on the entire network, e.g. the user querying Indian restaurants in Dublin may be currently in New York. Operations can also be performed by proxy nodes i.e. a node in Dublin may perform and aggregate results of other nodes in Dublin and return theses results to the user in New York [Cowz 09].

Typically, these systems have been applied to sensor networks or geographical service directories, where context is not tied to human behavior thus limiting the potential scope of its applications. For example, services like "Find friends dancing near me" would be good candidates to be implemented on top of decentralized networks (where the nodes would probably be cellphones). Also, to the best of our knowledge, none of the proposed decentralized context-aware applications to date support collaborative features.

Although most systems fall into just one of these categories, there are hybrid systems that combine different distribution models, like AwareMedia [Bard 06], an awareness tool to help hospital staff coordination. AwareMedia is developed using SIENA [Carz 01], which features an hybrid approach that mixes the partitioned and the distributed models. In SIENA, devices are distributed within partitions, with each partition having an associate server. Communication between partitions is made using server-to-server direct connections as usual. However, the devices within each partition are organized in a peer-to-peer model. Instead of communicating only with the partition server, they are also able to communicate directly between them. BikeNet [Eise 07] also implements a hybrid approach that combines a centralized model with a peer-to-peer model. Bicycle sensors may transmit context data directly to a central server or to other sensors in passing by bicycles, which are than transmitted to the central server.

3.4 Consume

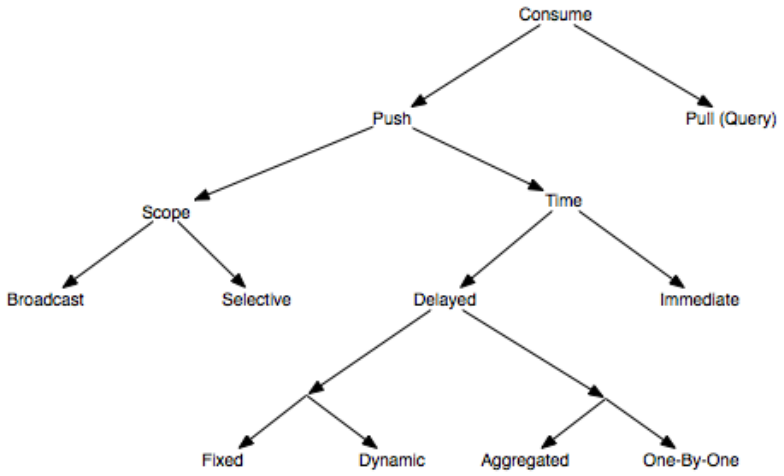

Figure 10: Consume layer

After context information has been captured from sensors, inferred into higher-level data and distributed through the network, it will be consumed by client applications. In this section, we describe how client applications obtain this context information in a distributed context-aware system, categorizing the possible options and presenting examples of such options. Figure 10 presents a taxonomy of approaches for dealing with context information consumption.

Context-aware systems can be either Push or Pull-based. In *pull-based systems*, the consumer (client application) queries the component who has the information (sensor or middleware) for updates. This can be done manually (e.g., the user clicks the "refresh" button) or periodically (e.g., checking for updates every 5 minutes). In *push-based systems*, the component that has the information is responsible for delivering it

to interested client applications. The main distinction between these approaches lies on who initiates the communication. If a client application initiates the communication the system is pull-based, otherwise it is push-based.

Pull-based mechanisms are simple to implement because the client application only has to query a certain component with a certain periodicity. In the extreme scenario, the application can simply delegate the responsibility of refreshing the information in the user. Also, this periodicity (polling interval) can change according to the application current needs. For example, an application running on a cellphone can increase the polling interval to save battery life, when the battery power falls below a certain threshold. However, this mechanism suffers from two problems that can be critical in a context-aware system. First, context information reaches the application with a delay. In the worst case, this delay equals the polling interval. If the polling interval is defined as 5 minutes then there will be cases where the application receives context information from 5 minutes ago, compromising its usefulness. The application can minimize this disadvantage by reducing the polling interval but then, the second problem may start occurring - most queries will be unnecessary since there isn't new context information available. Since the application doesn't know when there will be updates, it has no alternative as to keep asking until there is new information, leading to wasted network bandwidth and cpu consumption.

Push-based mechanisms are more complex then pull-based mechanisms. Since the component who has the information is also responsible for delivering it, this component has to know how to reach every possible consumer that is interested in that information. Usually, a permanent connection with all the consumers is necessary. If the system has a large number of consumers, its performance may degrade. Furthermore, these systems have poor scalability as every new consumer degrades the performance even worse. There are two measures to overcome the scalability problem: *reduce scope* and/or *relax delivery time*. We describe these measures in the following sections.

Although the *Push vs Pull* issue is mainly analyzed from the distributed systems perspective, context-aware systems raise additional challenges on deciding between both approaches. Cheverst [Chev 02] argues that applications that use push-based mechanisms may surprise the user, disrupting its current task. However, he also refers that allowing the application to present inconsistent data (as happens in the pull-based approach) may confuse the user. He tested this rationale with a location-aware visitor guide [Chev 01] that was originally designed following the pull model, whereby the onus was on the users to request the presentation of context-aware information.[3] Cheverst developed a push-based version of this application that immediately reacts to context updates (e.g., location changes) presenting the user with constantly updated information about nearby attractions. The subsequent evaluation shown that visitors were comfortable with the push-based version, because it required less effort to learn and use that the pull-based version.

3.4.1 Scope

In many context-aware systems, users are only interested in a subset of the system's available context information. For example, users of AwarePhone [Bard 04] (an application that enhances the traditional phone contact list with contextual information such as location and availability) are only interested in context updates for the people in their contact list. In this system, if person A has person B in his list of contacts, A is notified if B moves from one location to another, but other people in the system without

[3]For example, by tapping on the information button

B in their contact list are not notified. AwarePhone is built on top of the AWARE framework [Bard 04], which provides an event-based infrastructure, as long as the underlying communication channel allows it. Currently, the supported communication channels are: Java RMI, PHO (special-purpose optimized channel for cellphones) and HTTP. HTTP, due to its stateless request-response nature, is the only one that doesn't support push-based context propagation. ContextPhone [Raen 05] uses a similar mechanism on top of the XMPP protocol [Sain 05].

These are examples of *selective scope* systems, that is, systems that propagate only a subset of the available context information based on user's selection. Most context-aware systems propagate all the available context information, thus falling into the *broadcast scope* category (see Figure 10).

Using scope is an effective technique to reduce the number of pushed messages in the system, thus reducing the required outbound network bandwidth on the component that has the information and improving overall scalability.

Although selective scope filtering is usually applied on the middleware layer, some systems apply the filter on the sensor. Elvin [Sega 00] introduces the concept of *quenching*, a mechanism that allows sensors to know whether client applications are interested in their information and only sending information in that case.[4]

3.4.2 Time

Another way to improve push-based systems scalability is to assume that certain context information doesn't need to be immediately delivered. Users tolerate some lag as long as the information does not require urgent attention. For example, a context-aware system that shows friends near me isn't required to be continuously up to date. On the other hand, an application that shares current availability among a group of friends (e.g., to help decide whether a person can call a certain friend without interrupting anything) looses its usefulness if context information is not propagated as soon as possible.

These issues have been studied in the context of *optimistic replication algorithms*. Such algorithms increase availability and scalability of distributed data sharing systems by allowing replica contents to diverge in the short term [Sait 02]. Distributed context-aware systems are, in fact, a large distributed database of context information. In addition, most of them use a crude form of single-master replication - all updates originate at the master and then are propagated to other replicas, or slaves. Applying this general definition to context-aware systems, we have context information replicated among multiple nodes with sensors acting as the masters and end-user applications acting as slaves. Since sensors are the only components who *write* context information, these systems do not suffer from conflicting updates, one of the main problems found in optimistic replication algorithms.

Improved scalability is just one of the advantages of using delayed context propagation. This technique also allows for greater availability and network flexibility, working well over slow, unreliable or intermittent connection links. Such property is essential in mobile environments in which devices can be synchronized only occasionally. For example, Rich [Rich 03] proposes a system for managing user context across multiple devices that survives intermittent device connectivity by applying optimistic replication techniques.

In spite of this, the majority of distributed context-aware systems use the immediate approach pushing context information to end-user applications as soon as possible.

[4] In fact, Elvin is not specific to context-aware applications and quenching can be applied to any publish-subscribe system.

Delayed context propagation can be further categorized based on the *interval between updates* and the *amount of information transmitted on each update* (see Figure 10):

- **Fixed vs Dynamic** - Systems which postpone context propagation must decide when to actually push the updates. This decision can be made based on a fixed criteria such as time period (push the update every x seconds) or amount of retained information (push the update when there is more than x Kbytes of context information). For example, ReConMUC [Alve 11b] uses both criteria to improve the scalability of a context-aware IM application.

  Context propagation can also occur using a dynamic criteria, such as the importance or urgency of the context information. For example, consider a location-based system which propagates context more frequently if the recipient is geographically near the origin. In this system, people in the same building could receive context information form each other almost immediately while context information from people in vicinity but not in the building would be received with a certain delay.[5]

- **Aggregated vs One-By-One** - Since there is a delay in context propagation, information must be retained and may start accumulating. So, when the system decides to push the update, there may be a considerable number of messages to transmit. The most simple approach is to transmit the messages one-by-one, as would be the case if there wasn't any delay. However, it can also be extremely inneficient.

  Consider the case of a moving person which is carrying a GPS-enabled device constantly updating his location to his friends. Consider also that this system uses a delayed push technique to improve scalability, with a fixed period of 1 minute (that is, a maximum delay of 1 minute between update propagation). During each minute, the system may accumulate a large number of location positions (since the person is moving). If, after each period, the system transmits the location messages one-by-one, it consumes network bandwith without necessity because, in fact, only the last location is relevant to his friends. In this case, the system could aggregate all the location positions in just one (the last). Even when all the retained messages are relevant, they can still be aggregated in one big message, achieving higher levels of compression and much less round-trips in the connection path [Alve 11b].

  Note that, in context-aware related literature, the term "aggregation" is often associated with the combination of information from different sensors, along with conflict resolution. That *multiple sensor aggregation* should not be confused with the aforementioned aggregation which is specifically related to accumulated messages on systems that use a delayed push approach.

[5] Context from people outside the vicinity would not be received at all, but this falls into the selective scope approach.

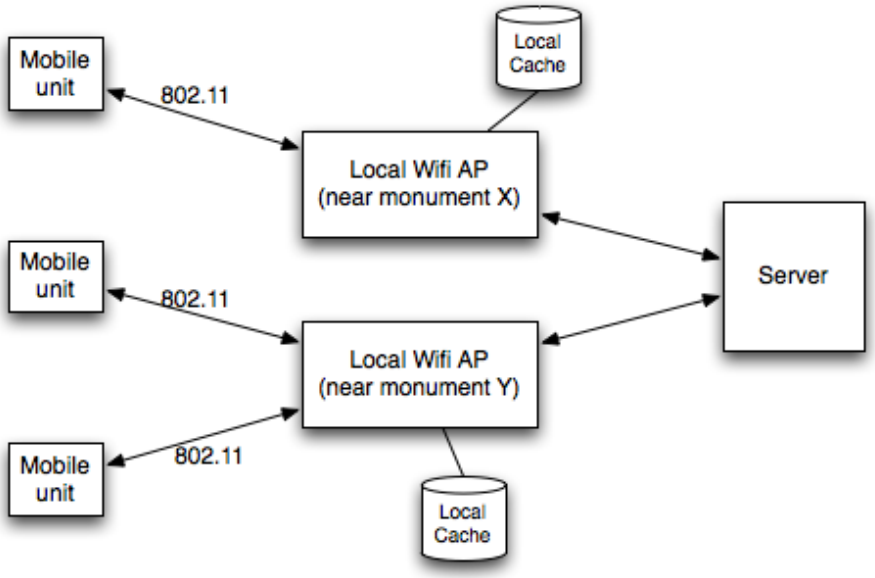

Figure 11: GUIDE architecture

4 Systems

Now that we have defined a taxonomy for distributed context-aware systems, we analyze some of the most relevant systems in this area, providing an overview of their functionalities and architecture and classifying them according to the above mentioned taxonomy.

4.1 GUIDE

GUIDE [Chev 01] provides city visitors with a hand-portable context-aware tourist guide. The system has been successfully deployed in the city of Lancaster and is publicly available to visitors who wish to explore the city. The city's major attractions are covered with a cell-based wireless communications infrastructure (based on the 802.11 *wifi* protocol). Each wireless access point (AP) has an associated *cell-server* and is responsible for (i) broadcasting location beacons to provide positioning information and (ii) disseminating both static and dynamic information to mobile GUIDE units. These *cell-servers* have local storage capabilities and act as a proxy cache to the central GUIDE web server (see Figure 11).

Each mobile unit stores the user profile (preferred language, etc.) and, combined with location information provided by the location beacons, provides relevant information to its user about nearby touristic attractions.

GUIDE uses a purpose-built information model [Chev 99] that represents places, such as attractions and key buildings, within the city. This model is translated into HTML by a local web server resident in every GUIDE unit.

This system had an interesting evolution since it started has a distributed non-collaborative system and was subsequently extended to support collaboration. In particular, two collaborative features were implemented: (1) Create a comment and rating for association with a particular attraction; (2) Realize when another GUIDE user is (or has recently been) physically located at a particular attraction. In order to

provide these features, GUIDE units were changed to periodically acknowledge their location to the APs, in order to share their location to other users. To protect their privacy, users are able to specify if they do not mind having their location recorded but wish to remain anonymous (in that case, the userId that is passed to the server is a random one) and if they do not wish to have their location recorded at all.

Since acknowledging every location beacon would result in large network bandwidth consumption, only 1 in 10 beacons is acknowledged by the mobile unit.

In summary, GUIDE captures context (location) through physical sensors (Wifi). Nearby attraction names are inferred from the IP address of the corresponding AP, in the server (middleware). Context information is distributed through a centralized topology with an intermediate caching layer that reduces the dependency between the mobile units and the central server. Information about touristic attractions is obtained by querying the server (pull) but the location of the mobile units is detected using a selective, periodic, push-based approach. It's selective because the location beacons are only emitted to a subset of all the mobile units in the city (only those in the AP wireless range). It's periodic (fixed delay) because the beacons are pushed to the mobile units periodically. As already noted, only 1 in 10 location beacons are acknowledged by the mobile unit, which can be considered a form of context aggregation.

4.2 UbiqMuseum

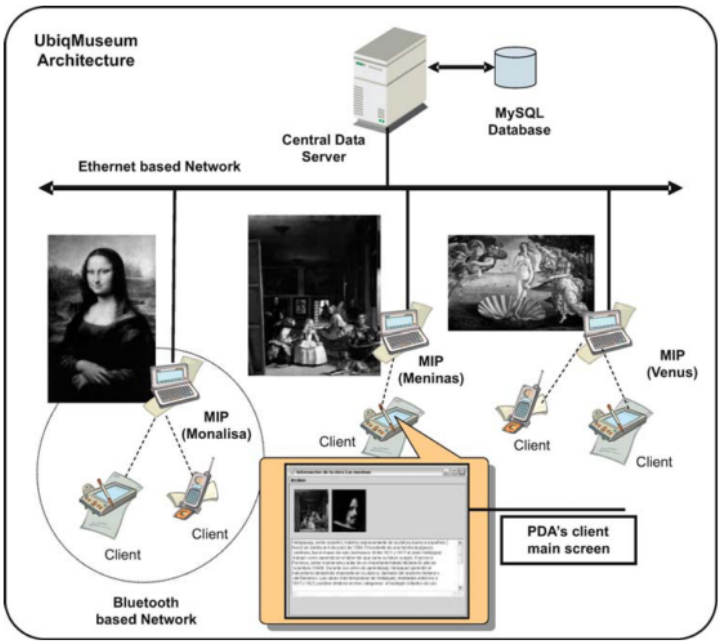

Figure 12: UbiqMuseum architecture

UbiqMuseum [Cano 06] is similar to GUIDE but inside a building (usually a museum). This system provides relevant information to museum visitors through mobile devices based on proximity to pieces of art, using Bluetooth to give proximity in-buildings context. The short wireless range of Bluetooth technology is suitable to much finer-grained location than Wifi and, unlike GPS, works inside buildings.

UbiqMuseum's architecture features an edge wireless network based on Bluetooth technology used by mobile devices integrated with a core network based on a fixed Ethernet and wireless 802.11b LAN. The core network connects mobile clients with servers. The system considers three types of software entities: client applications, museum information points (MIPs), and the central data server. MIPs are associated with one or more pieces of art or objects (see Figure 12).

A mobile client, while wondering around the museum, continuously searches for new MIPs through Bluetooth inquiry process (Service Discovery Protocol). Then, mobile clients send the user profile to the MIP, also using Bluetooth. The MIP forwards the request to the central server (the authors do not specify which protocol is used in this communication) and receives information about that piece of art, which updates its local cache and forwards to the mobile client.

This system does not provide any collaborative feature and context (location) is never persisted, so there is no need for privacy mechanisms.

In summary, UbiqMuseum is very similar to GUIDE: physical sensors (Bluetooth) capture location information, which is inferred into pieces of art by the server (middleware) and distributed along a centralized topology (again, since the MIPs can cache information, the dependency of the central server is reduced). The main difference w.r.t. GUIDE is the location detection mechanism, which is pull-based. The mobile clients are responsible for periodically discovering and querying nearby MIPs.

4.3 ContextContacts

ContextContacts [Oula 05] enhances the traditional cellphone contact list with cues of the current situation of others, such as location, time spent in that location, phone speaker state and number of persons near the phone (Figure 13 shows the application's graphical interface). The goal is to help the caller to decide if the callee can be interrupted and, in that case, which communication channel to use.

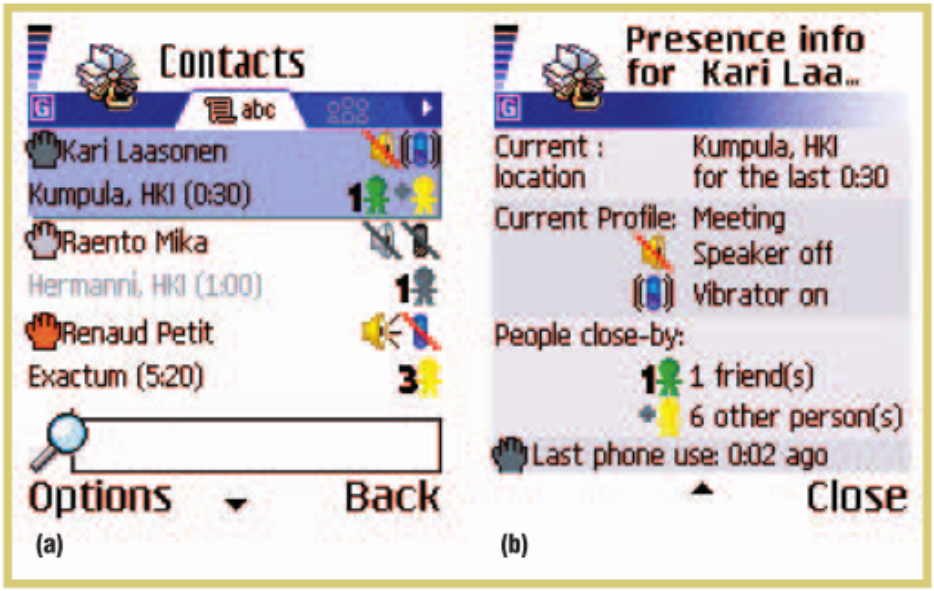

Figure 13: ContextContacts graphical interface

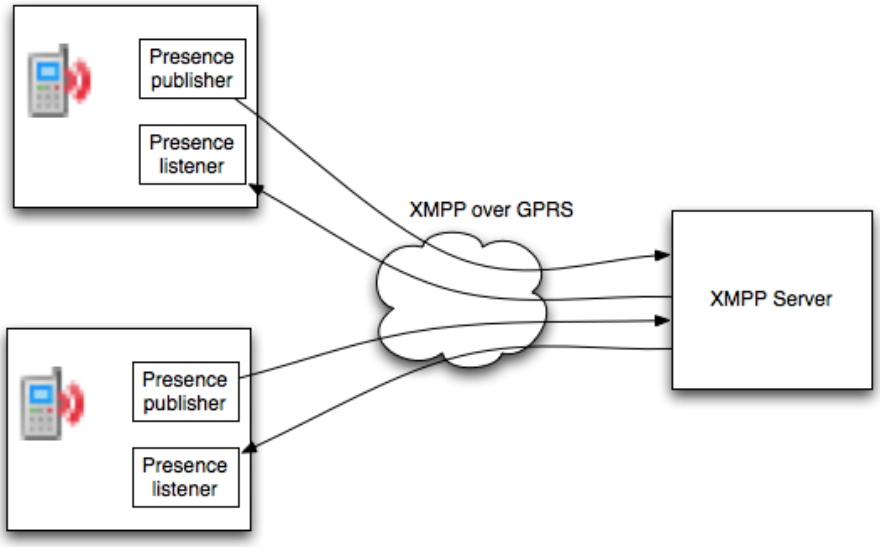

Figure 14: ContextContacts architecture

Unlike the previous examples, this application captures other types of context besides location, like phone activity (speaker and vibrator state, calls), sensing how many people are nearby using Bluetooth, etc. Location is obtained through the GSM cell id.

From the architectural perspective, the system is built on top of ContextPhone [Raen 05], a generic platform to help develop context-aware applications for cellphones. This platform consists of four modules: sensors (acquire and infer context data such as location and phone use), system services (error logging, service recovery, etc.), communications (HTTP/XMPP over GPRS, Bluetooth, SMS, GPS over Serial, etc.) and customizable applications (applications that seamlessly augment or replace built-in applications). ContextPhone is an example of a customizable application, with three specific components: (1) the presence publisher which gathers and sends relevant sensory data to other users via XMPP; (2) the presence listener which receives sensory data from others and integrates it into the applications' user interface and (3) application customizations such as the adaptation of the built-in contacts list application (see Figure 14).

Although Bluetooth is used, it cannot be considered a communication channel since its purpose is to gather sensory data related to people nearby. All communication is done using XMPP push-based presence protocol. This protocol establishes a two way permanent channel (TCP) between the client and the server and propagates presence information based on a publish-subscribe model [Plal 03].

Since this system provides collaborative features to help synchronization of a group, its user's privacy must be preserved. In ContextContacts, privacy is managed using self-awareness mechanisms: the application provides a separate view showing exactly how others see the user at the moment. However, the only way to control what is revealed is to switch off the application. The authors say they are planning better control of which information is shared with whom.

In summary, although the system captures a wider range of context information (not only location) it is still mostly captured though physical sensors. Location is inferred from the GSM cell id by the client application associated with that phone. That is, the client application infers the higher-level location infor-

mation (e.g., city) before propagating that information for the other devices. Since ContextContacts relies on XMPP protocol, is benefits from its partitioned architecture, with the advantages outlined in Section 3.3. Also, since XMPP uses a publish-subscribe model, the system is push-based, with immediate and selective propagation. It is selective since context is only propagated to the users that are on the contact list (not every user in the system).

4.4 BusinessFinder

BusinessFinder [Chak 07] may be viewed as a "Live Yellow Pages" service, that factors in the actual mobility of both the requester (the customer seeking a service) and the vendors (e.g., the electrician or plumber offering the service) to perform on-demand matching. The system is targeted to *nomadic* vendors whose location is always changing as opposed to traditional local static services like restaurants and gas stations. Users trigger the system by sending an SMS like "Find me the nearest plumber".

To increase adoption, BusinessFinder doesn't include a client application to be installed in cellphones. Therefore, it doesn't have direct access to cellphone sensors. Instead, it relies on the services provided by the cellphone operator infrastructure, either using a Parlay Gateway (a generic interface to common mobile services such as location, SMS processing, call control) or a SIP Presence Server (see Figure 15). The interaction with these interfaces can be query-based (pull) or push-based (change-triggered notifications). BusinessFinder also maintains a database of vendors profiles including skills, availability and rating.

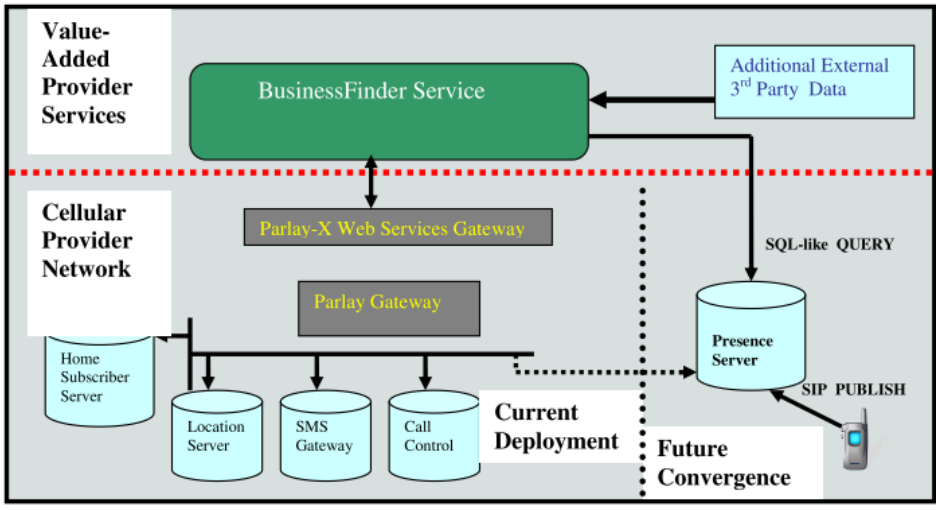

Figure 15: BusinessFinder architecture

The system was implemented following the centralized model (a single Presence Server or Parlay Gateway interacting with a single Location Server). The authors are currently researching an alternative distributed architecture, partitioned by geographical areas. In this architecture, all vendors currently resident in a specific zone transmit their presence updates to the corresponding server. The major challenge in this model occurs when there are no available vendors in the local server and the query needs to be routed to alternative servers from nearby areas. The authors propose a *Resource-Aware Query Routing (RAQR)* algorithm [Chak 06] that avoids broadcasting or query flooding by maintaining a *gradient-offer* table in each

server that is near resource exhaustion. This *gradient-offer* table is a local representation of the available resources on nearby servers, and can be used to reserve certain remote resources anticipating local resources exhaustion (see Figure 16).

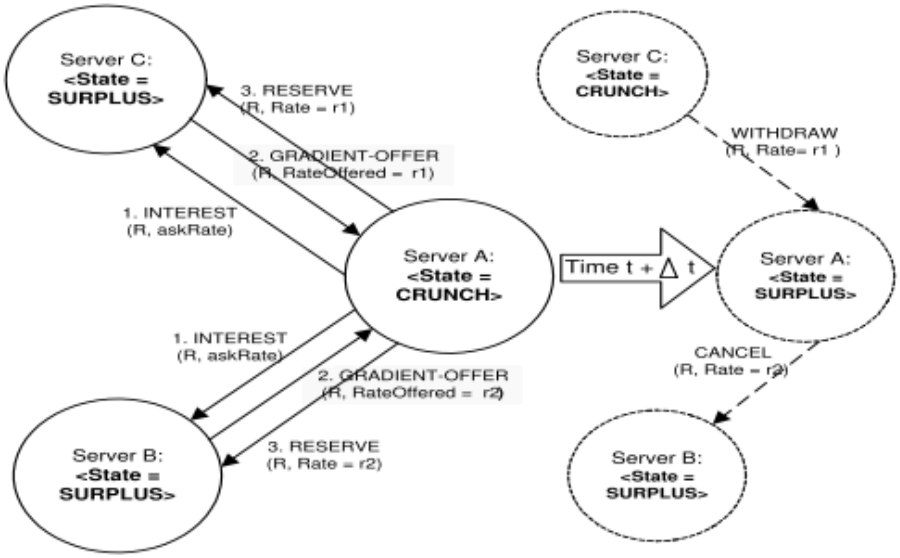

Figure 16: RAQR Basic Steps

Regarding privacy, although the system continually keeps track of its users location, that information is never directly revealed unless a match occurs between client and vendor. Even in that case, all subsequent interactions are explicit and identities are only revealed in the last step - upon matching the system sends an SMS to the requester like "Vendor at distance 10.85 kms is available. Reply Connect Yes/No".

In summary, BusinessFinder uses the physical sensors associated with the cellphone (although indirectly through the cellular operator infrastructure). Also, inference is provided by the cellular operator infrastructure with is organized in a centralized architecture. Context can be consumed either using the pull mechanism or the push mechanism. In the last case, the events are broadcasted immediately.

4.5 AwarePhone

AwarePhone is similar to ContextContacts in the way it augments the traditional phone contact list with richer context information, but it is implemented over a different architecture - the AWARE framework [Bard 04]. One of the interesting characteristics of this framework is its support for direct communication (e.g., IM) besides context propagation. In fact, the core idea in the AWARE architecture is to combine CSCW system components for providing social awareness among collaborating users with Ubiquitous Computing components for obtaining context-awareness.

In particular, AwarePhone is an application for cellphones that displays a contact list with enhanced context information (personal status, activity and location) and supports simple text messaging. It uses diverse sensors such as Wifi, IR, Bluetooth, Calendar (from which the current activity is inferred) and Status (virtual sensor: the user manually sets its status). Location can be inferred from Wifi, IR or Bluetooth depending on the environment.

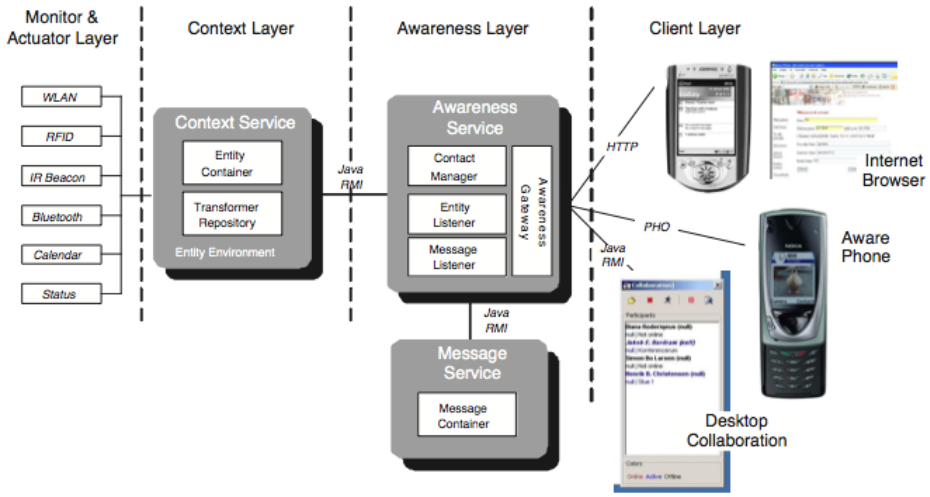

Figure 17: AWARE architecture

The AWARE architecture is divided into four layers (see Figure 17):

- **Monitor & Actuator Layer** - This is the sensor layer. Currently, AWARE supports some location sensors (Wifi, Bluetooth, IR beacon, RFId), an activity virtual sensor based on the calendar, and a personal status sensor manually set by the user.
- **Context Layer** - This layer manages context information about relevant entities in the real world, such as people, places and things, and distributes this information to the awareness layer using either a request-response or publish-subscribe pattern.
- **Awareness Layer** - This layer has three responsibilities: (1) maintain information about users subscribing to the AWARE system, and about whom they want to maintain social awareness; (2) handle connections to clients using different protocols, respond to their requests and know how to notify them about relevant events and (3) manage a message broker than enables users to post messages to each other. The message broker is an optional component which can be replaced by the standard SMS service, but the authors claim that its "embedability" provides a smoother transition from context awareness to direct communication.
- **Client Layer** - This layer includes all end-user applications that use this framework, including a phone client, a desktop client and an internet (browser) client. This layer communicates with the Awareness Layer through a gateway which is able to transform protocol-specific requests into internal AWARE API calls.

The Awareness layer uses the context infrastructure by connecting to a Context Service and registering itself as a listener to relevant entities. Then, it starts receiving notifications of relevant changes. Applications running on devices can now treat these events appropriately, i.e. updating the user-interface or notifying users.

End-user applications can use a variety of protocols to communicate with the AWARE framework, such as HTTP or Java RMI. In the specific case of AwarePhone, a special-purpose PHO protocol was designed

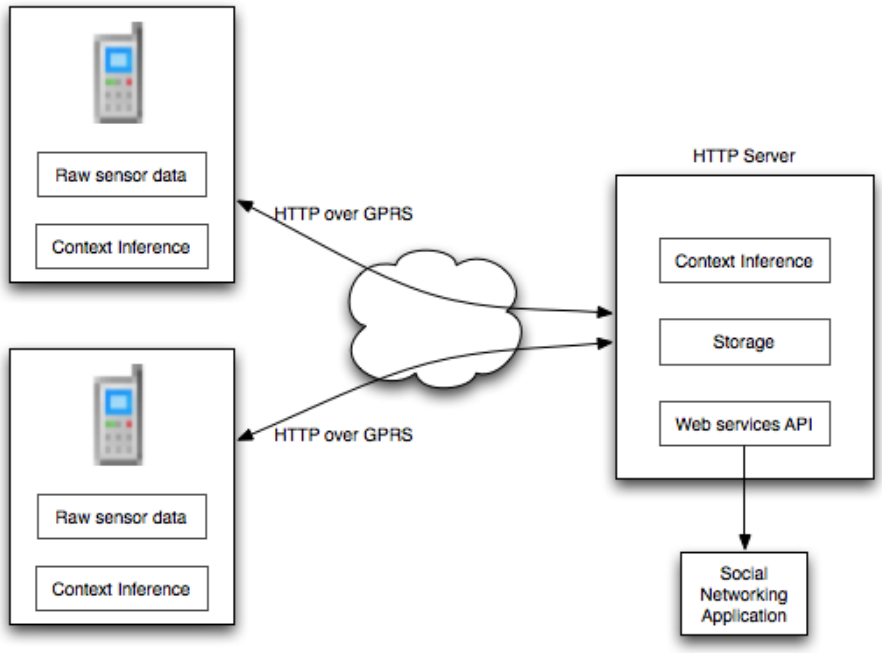

Figure 18: CenceMe architecture

[Bard 04]. This protocol uses a compact string format over TCP/IP. The communication between the Awareness Layer and the Context Layer uses the Java RMI protocol. Note that event-notification (push) is only possible when using PHO or Java RMI.

From the user's perspective, the main difference between ContextContacts and AwarePhone is that the last one provides richer context information and mechanism for direct communication. Note that even though AwarePhone is a collaborative system, there is no support for privacy mechanisms although the authors refer a need for such mechanisms.

In summary, AwarePhone captures context from both physical and virtual sensors. Higher-level context is inferred by the transformer repository (middleware, see Figure 17). The system relies on a centralized architecture, and information can be consumed either using pull or push mechanisms (depending on the underlying communication protocol, as previously explained). The available push mechanism propagates context information immediately and selectively (only to those users on the contact list).

4.6 CenceMe

CenceMe [Milu 08] exploits off-the-shelf sensor-enabled mobile phones to automatically infer people's sensing presence (e.g., dancing at a party with friends) and then shares this presence through social network portals such as Facebook. While systems like ContextContacts and AwarePhone are much more focused on productivity features (e.g., reducing the chance of interrupting someone), CenceMe is focused on connecting people by sharing their personal status in a social network, usually for entertainment purposes.

The personal status is very sophisticated and goes well beyond traditional location. The phone's microphone is used to capture audio samples from the environment (e.g. detecting if the user is in a dance club)

and the built-in accelerometer is used to infer current activity like sitting and walking. Bluetooth, GPS and Camera are also used (the last one to take random photos of the environment surrounding the user). Richer sensors such as microphone and accelerometer imply more complex inference engines, using CPU-intensive operations such as Discrete Fourier Transform (DFT) and a machine learning algorithms.

CenceMe is implemented in a traditional centralized architecture, where an application installed on the mobile phones transmits and receives context information to/from a HTTP server (see Figure 18). However, this system proposes an interesting twist over other similar systems called *split-level classification*. The idea is to push some of the context-inference process to the mobile phone, and some to the server, in order to improve scalability. This architecture offers some advantages: (1) supports *customized tags* (any activity, gesture or audio that the user can bind to a personal meeting, such as drawing an imaginary circle with the phone to indicate "going to lunch"); (2) provides resiliency to cellular/Wifi dropouts, by computing and buffering context information when there is no radio coverage; (3) minimizes the sensor data the phone sends to the server improving the system efficiency by only uploading classification derived-primitives rather than higher bandwidth raw sensed data; (4) reduces energy consumed by the phone by merging consecutive uploads and (5) eliminates the need to send raw sensor data to the backend, enhancing the user's privacy.

The protocol used for communication is HTTP over Wifi or GPRS, depending on network availability. Data exchange is initiated by the phone at timed intervals whenever the phone has primitives to upload. The phone also periodically pings the server with control messages. Messages from the server are piggybacked on both phone initiated messages. To extend the battery life of the phone when running the CenceMe application, data upload and sensing components may be slightly delayed, minimizing sampling and context propagation while maintaining the application's responsiveness. Right now, this delay must be set manually by each user. Also, although it increases the battery lifetime of the phone it was detected to have a negative impact on the performance of the classifiers.

It is noteworthy that split-level classification allows for some aggregation of messages. By uploading to the server higher-level context data instead of raw sensor data, the application is effectively aggregating context without loosing relevant context information. Also, to reduce the number of data cellular connection, the application merges consecutive uploads to the server.

In any system with social features, user's privacy must be assured. CenceMe includes a privacy settings GUI that allows the user to enable or disable the five sensing modalities by user with whom they are willing to share some information. The authors also refer enhanced privacy as a consequence of split-level classification, because raw sensor data is no longer transmitted to the server. It is noteworthy to refer that since all transmitted data is published to social network applications, the onus of setting appropriate privacy rules is somehow transferred to these applications, since CenceMe is only the "messenger".

In summary, CenceMe uses diverse physical sensors to capture context. Through split-level classification, part of the context is inferred in the client application and another part is inferred in the server. This is necessary because inferring high-level information from microphone and accelerometer data is a resource-demanding task. Context information is distributed in a centralized architecture, and consumed using pull mechanisms (the cellphones query periodically the CenceMe server).

4.7 BikeNet

BikeNet [Eise 07] was developed by the same group that developed CenceMe to improve the cycling experience with miscellaneous sensors attached to the bike. During a ride, these sensors continuously capture environmental and personal data such as current speed, average speed, distance traveled, path incline, heart rate and galvanic skin response. This data is transmitted to a central server for further visualization and comparison with friends/competitors through a web-based portal.

The system is organized into three tiers: the backend server tier, the sensor access point (SAP) and the mobile sensor tier (see Figure 19). The mobile sensor tier includes all the sensors attached to the bicycle, including an accelerometer, thermistor, photodiode and microphone. The SAP tier acts as a gateway from the sensor tier to the backend server. SAPs can be static and wired directly to the Internet (e.g. wireless 802.11 router), or can be mobile using a wide area radio access network (e.g. mobile phone with GSM/GPRS). Static SAPs my be distributed across a cycling route. When the bicycle comes within the range of these SAPs, sensor data is uploaded to the backend server. This is the default mode, where cyclists go on trips, collect sensed data, and upload their data when they return home. Obviously, this implies a delay between the time the sensor data packet is generated and the time this packet reaches the central server.

This delay can be reduced using a mobile SAP. If the rider goes cycling with its cellphone, it can be used as a real-time interface between the sensors and the central server. In this case, the only delay is the time it takes for the sensor data to be transmitted over the GPRS connection, and it is possible to follow the ride in real-time using the web portal (e.g., updating the current location on a map). In any case, BikeNet distributes context information using a centralized topology, either using a *delayed sensing mode* or an *immediate continuous sensing mode*.

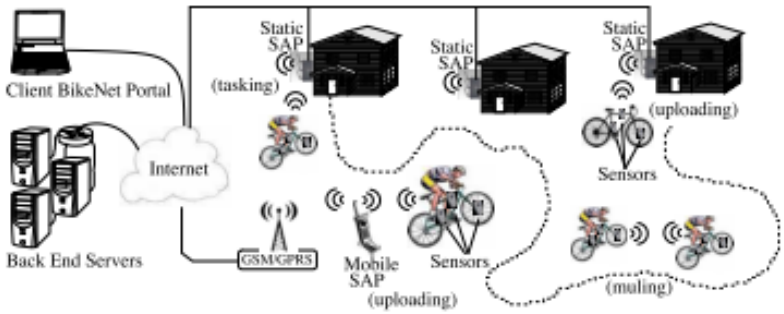

Figure 19: BikeNet architecture

However, this system introduces an interesting networking solution that takes advantage of the opportunities that arise as a result of the uncontrolled mobility of the cyclists, using a *muling* exchange protocol (which is in fact a simplified epidemic protocol [Deme 88]). In the muling exchange, sensed data is transferred between mobile sensors outside the wireless range of either a mobile or static SAP. This occurs when two bicycles equipped with BikeNet pass by each other. The sensors of one bike are able to detect the presence of another bike and establish a direct wireless connection between them. The idea is to collect sensed data from other bicycles found along the ride, in addition to our own data. When entering the wireless range of static SAP, all data is uploaded to the server. That is, the first bicycle to stop uploads sensed data of the other (still running) bicycles, decreasing the already mentioned delay that occurs when using static SAPs.

In addition, if one of the bicycles participating in the muling exchange has access to a mobile SAP (when the cyclist brought its cellphone), sensed data is uploaded immediately further reducing the delay. This opportunistic protocol creates, in fact, small peer-to-peer networks that allow the system to efficiently work in disconnected or intermittent clients. Note that, in this system, only one-hop muling is allowed - sensed data can be replicated but replication of mulled data is not supported. The reason argued by the authors is that if unlimited replication was allowed, it would be impossible for the client to know the number of copies of its data in the system.

The three context propagation modes (static SAP, mobile SAP and muling) are depicted in Figure 19 along the overall architecture.

4.8 Tickertape

Figure 20: Tickertape interface showing some scrolling messages/notifications

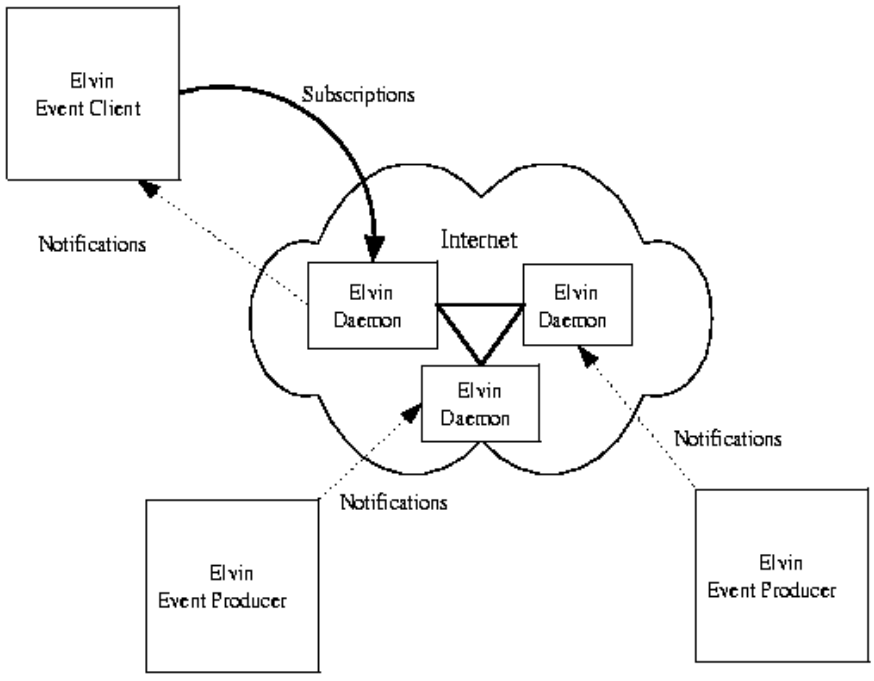

Figure 21: The Elvin architecture for content based routing

Tickertape [Fitz 99] is a desktop application that displays notifications on topics that the user has previously subscribed to. Its interface, shown in Figure 20, consists of a single resizable rectangular window, showing small, colour-coded messages scrolling from right to left. Context is propagated using Elvin [Sega 97], a generic notification service following the publish-subscribe paradigm. Users can subscribe to groups of

messages (e.g., "work" messages) and can filter messages with those groups based on their content (e.g., messages that contain the word "tickertape").

Unlike the already mentioned context-aware systems, Tickertape relies solely on virtual sensors, mainly information sources (Usenet news, CNN stories, etc.) and chat messages. Since these sensors produce high-level information there is no need for an inference engine in this system.

As already refered, Tickertape is developed on top of Elvin, a content-based notification server using a partitioned architecture, as shown in Figure 21. Multiple Elvin Daemons (servers) listen to events coming from Event Producers and deliver them to interested clients, based on their subscriptions. Communication between clients and servers is asynchronous and can use a variety of protocols ranging from raw TCP and UDP to HTTP and HTTPS.

From a collaborative point of view, Tickertape provides mechanisms for shared context-awareness in a group of people but also for direct communication (clicking on the Tickertape opens a pop-up dialogue where users can write messages).

In summary, TickerTape captures all context information from virtual sensors, it doesn't use any inference engine, it relies in a partitioned architecture and context information is pushed immediately and selectively (based on user-defined filters) to the client application.

4.9 NESSIE

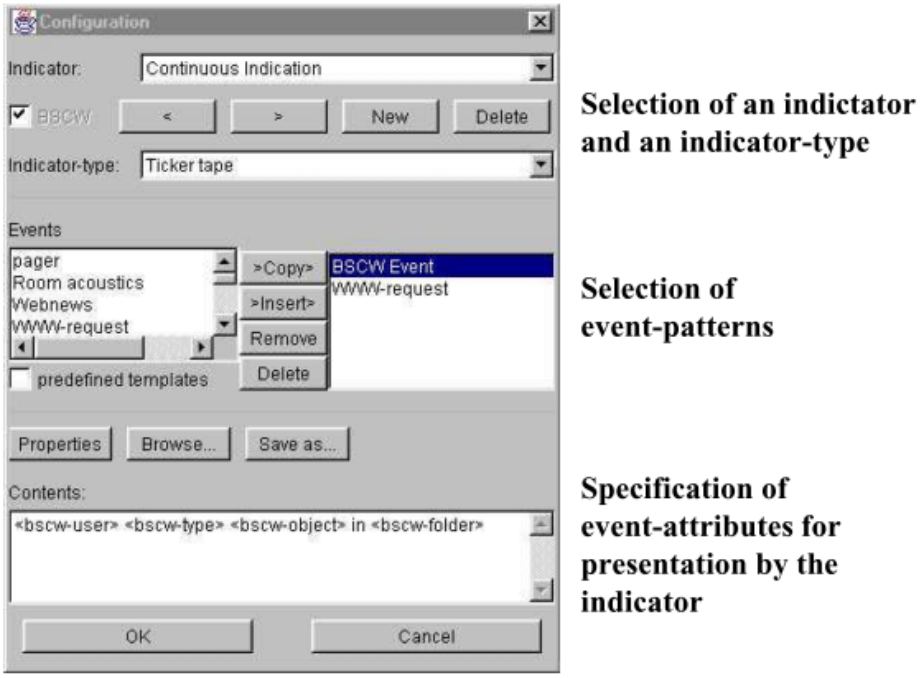

Figure 22: NESSIE client configuration

NESSIE [Prin 99] is an awareness environment for cooperative settings in an office. A NESSIE client application runs on a PC, presenting events taking place in the office and showing locations of others in places of interest (e.g., shared information or social places). Information can be presented though multiple

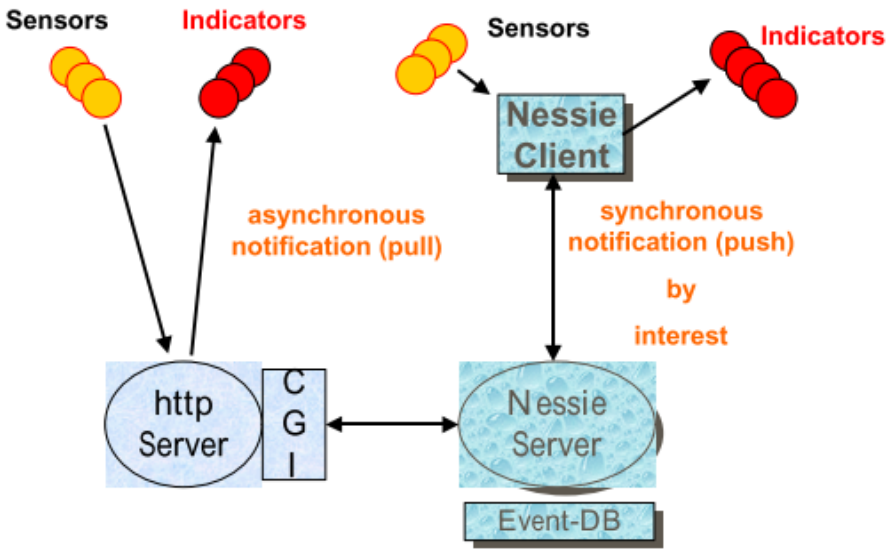

Figure 23: NESSIE architecture

interfaces like simple windows with events listings, background images and sounds or tickertape-like [Fitz 99] applications. In short, NESSIE matches user-defined filters with context information captured in an office (see Figure 22), mainly through virtual sensors that detect modifications on shared artifacts (e.g., documents).

NESSIE uses mostly virtual sensors, although some physical sensors have also been integrated. Some virtual sensors allow the application to know the *information space location* of every user. *Information space location* refers to a virtual location that can be visited by users, like a web page, a document or a virtual world's room. In addition, physical sensors like video image motion recognition, infrared-motion detection and acoustic microphones are used to detect the presence of people in a room.

NESSIE follows a traditional client-server architecture, in combination with an event database. The server supports two methods of production and provision of events: asynchronous (pull) server interaction (HTTP calls) and synchronous (push) notifications based on registered interest profiles by the NESSIE client (see Figure 23). It also introduces the concept of *Indicator*, an application with a GUI (Graphical User Interface) that shows context information. Even though sensors may communicate directly with the NESSIE client (i.e., without going through the Nessie Server), we believe the most common case to be sensors communicating with the server using the pull mechanism and then the server propagates this information to interested NESSIE clients (based on user profiles) using the push mechanism.

Regarding privacy, NESSIE events can include an access control attribute listing the users who are allowed to receive that event. Also, to prevent misuse, NESSIE provides a *reciprocity* mechanism - it discloses the users who have registered interest in the events produced by a certain user ("when you see me, I see you"). This information is useful not only for the support of reciprocity. It further informs a user about the list of people who will become aware of a certain activity. This is important when a certain reaction is expected on the activity that raised the event.

Regarding inference, the NESSIE server includes an event transformation module that allows the transformation of submitted events.

In summary, NESSIE uses both physical and virtual sensors to capture context information which may be subject to further inference in the server. Data is propagated though a centralized architecture and uses both push and pull mechanisms. If the client application (*indicator*) consumes information from the HTTP server, it does so using the pull mechanism. On the other hand, the NESSIE client communicates with the NESSIE server through a push mechanism. In the last case, the NESSIE server pushes information to the client immediately and selectively (based on user-defined filters).

4.10 Summary

Applying our taxonomy to the surveyed systems, we reach the results shown in Table 3.

	Capture	Infer	Distribute	Consume
GUIDE	Physical	Middleware	Centralized (1)	Pull (attraction information) / Push: selective, fixed delay (location)
UbiqMuseum	Physical	Middleware	Centralized (1)	Pull
ContextContacts	Physical	Client application	Partitioned	Push: selective, immediate
BusinessFinder	Physical	Middleware	Centralized (2)	Push: broadcast, immediate
CenceMe	Physical	Middleware / Client application (3)	Centralized	Pull
AwarePhone	Physical / Virtual	Middleware	Centralized	Pull / Push: selective, immediate
BikeNet	Physical	Middleware	Centralized / Peer-to-Peer	Pull
TickerTape	Virtual	N/A	Partitioned	Push: selective, immediate
NESSIE	Physical Virtual	Middleware	Centralized	Pull / Push: selective, immediate

Table 3: Classification of the surveyed systems

We can see that most analyzed applications use physical sensors on the Capture layer, except for AwarePhone (which uses an activity virtual sensor based on the user's calendar) and TickerTape (which uses virtual sensors for information sources such as CNN and chat messages).

Regarding the Inference Layer, most applications infer context in the Middleware. ContextContacts infer the location directly in the Client application, based on the current GSM cell id. CenceMe uses an hybrid approach where part of the inference is done in the Client application installed in the cellphone and the other is done in the Middleware.

We can also observe that most applications use the simpler Centralized distribution model, although ContextContacts and TickerTape use a Partitioned model (the former using XMPP and the later using Elvin has their underlying messaging infrastructure). BikeNet uses an hybrid model that combines the Centralized approach with a Peer-to-Peer approach (the muling exchange protocol).

Finally, the applications are very diverse in how they consume information. Some of them, such as ContextContacts, BusinessFinder and TickerTape use a Push approach while others such as UbiqMuseum and BikeNet use a Pull approach. The remaining applications use a mix between the two models. Of the applications that use a Push model, the majority opted for selective immediate propagation, with only one implementing a broadcast model (BusinessFinder) and another implementing a fixed delay propagation instead of an immediate one (although it was just a periodic beacon to infer the location). In fact, we can observe that none of analyzed applications used a delayed propagation approach in spite of the advantages presented in Section 3.4.2.

(1) Since the location inference is based on multiple access points (either Wifi or Bluetooth) with a caching mechanism, these systems may be considered semi-partitioned but they must rely on a central server, at least while the caches are empty.

(2) The authors are planning a partitioned version of BusinessFinder

(3) This system uses split-level classification: some classification is executed in the phone and other in the backend servers (middleware).

5 Conclusions

This section concludes the article by summarizing the main challenges faced by developers of distributed context-aware systems.

Context information in distributed systems usually flows through four steps: *capture*, *infer*, *distribute* and *consume*. This division makes the analysis of such systems much easier, because it fits well within typical context-aware architectures and uses simple verbs to denote actions that evolve the state and location of the context information. This article focuses on these actions and tries to answer the question: "what happens to context since it is produced until it is consumed?".

The *capture* step is what triggers the process. Traditionally, context has been captured using physical sensors and with the sole objective of providing the user's location. This is changing in two fundamental ways. First, the range of available physical sensors is now much wider, encompassing such devices as accelerometers, thermometers and photodiodes. Many of these sensors are now bundled with off-the-shelf cellphones. As more of these sensors are used in the *capture* step we can expect much larger volumes of sensed data to be distributed to the system components, with the associate risk of network bandwidth exhaustion. Second, as more people use computers on everyday tasks (for work or leisure), the number of available virtual sensors is also increasing. Time-tracking desktop applications are able to record every user action on a computer. Social network portals such as facebook or twitter expose personal status updates, location and behaviors of millions of users. Face-to-face communication is being replaced by instant messaging, email and video-conferencing. All these applications are potential virtual sensors and may generate so much information that the problem is no longer the available network bandwidth but rather the user attention bandwidth. The fundamental tradeoff in this step is between richer context information and information overload. The sensors at our disposal are more ubiquitous and cheaper than ever, but is the system infrastructure able to cope with all this information?

One possible solution to context information overload lies in the *infer* step. This step is able to transform large volumes of raw sensed data into much smaller high-level information. For example, consider an inference engine that continually transforms megabytes of raw audio data captured from a microphone in a room into a binary representation "occupied" or "empty". However, larger and more complex sensed data requires a more resource-demanding inference engine. Even modern smartphones with last-generation CPUs are not appropriate to run complex inference algorithms which would quickly deplete battery power. As such, this engine is usually deployed in servers with large capacity. Although this solves the performance problem, it creates other problems, such as reduced personalization and increased risk of privacy breach.

The third step deals with context distribution. There are three main options to solve this problem, although some systems combine two of them: using a central server (centralized), using multiple central servers, each one responsible for a subset of the system clients (partitioned) and a peer-to-peer approach where each node acts as both server and client. Some distributed context-aware systems present specific requirements that may dictate the distribution approach. For example, such systems usually include mobile components (sensors, cellphones) that are subject to intermittent connections, which will certainly cause problems on centralized or partitioned architectures. Ironically, most of these systems are centralized, perhaps because the routing algorithm is simpler to implement. Some centralized systems provide an intermediate layer acting as a proxy cache (e.g., UbiqMuseum, GUIDE) that alleviates the problem without actually solving it.

Location-aware systems (e.g., BusinessFinder) also imply specific requirements with direct influence in the distribution strategy. A promising approach is to use partitioned architectures, with each server responsible for a geographical region. In fact, depending on the system's goal, any context dimension can be used to partition the client space reducing the probability of multi-hop communications.[6] Peer-to-peer systems are more resilient to intermittent connections and allow greater privacy control but its management is more complex. A possible solution is to combine ad-hoc spontaneous peer-to-peer networks between nearby devices with a partitioned or centralized approach (e.g., BikeNet), bringing to system the advantages of both worlds.

Finally, the *consume* step is directly related to immediacy requirements ("Do client applications need up-to-date context information or do they tolerate some delay?") as well as frequency of updates ("Do sensors continuously produce context events or only sporadically?"). If there are a lot of updates but the client application doesn't need to receive them right away, than a simple pull-based approach is sufficient. For other scenarios, a push-based approach is more efficient, specially if propagation is filtered based on client-defined criteria. For applications that do not require immediate propagation, combining a delayed approach with aggregation can result in a more efficient use of resources, as shown in [Alve 11b].

As we have shown, the development of distributed context-aware systems faces many challenges. Among them, we would like to stress two problems and a possible roadmap. These are, in our opinion, the most important problems to be solved in large-scale distributed context-aware systems: (1) efficient and scalable context propagation and (2) privacy control. None of the surveyed systems in this article solves effectively these two problems.

At first sight, they seem to be unrelated problems but a closer look detects an high correlation between them. For example, delegating the inference step into middleware components may increase the system scalability but creates privacy issues, as raw sensed data is no longer under the user's control. Interestingly, we believe there is a technique that may solve both problems: *aggregation*. The term *aggregation* appears twice in this article to mean slightly different things. First, it appears as a result of the inference step, which effectively aggregates raw sensed data into high-level context information. Second, it appears as an option in push-based systems with delayed updates - since there is a delay, accumulated context data can be aggregated before it is transmitted. Although in different times on the context life cycle, these two types of aggregation achieve the same two important results:

- **reduce the volume of information** - an efficient solution to propagation of large volumes of information. The more system components apply aggregation, the more scalable the system is;
- **transform information** - a practical solution to many privacy problems. By judiciously combining or transforming information, we can effectively anonymize personal data.

As an example, consider the aggregation of data from a GPS device continuously tracking our position. By aggregating all this data into high-level sparse context information such as "at work" or "at cinema" we greatly reduce consumption of system resources while, at the same time, protect the user's privacy.

[6] In partitioned systems, multi-hop communication occurs when a client in one partition has to communicate with a client in other partition.

When referring to privacy management techniques, the term aggregation is also used as a means to increase entropy of personal sensitive information, thereby assuring increased privacy protection to the end-users [Alve 11a]. For example, one of the most promising anonymization techniques called Spatial-temporal cloaking consists on transmitting someone's location only when there are enough users in the vicinity also transmitting their location. In that case, the location is aggregated into something like "In the last 5 minutes, there were nine people in Mosteiro dos Jernimos", which still carries important context information without revealing the identity of the people involved. We refer to [Alve 11a] for a more thorough analysis of privacy issues in distributed context-aware systems.

References

[Alve 11a] P. Alves. "Privacy in distributed context-aware systems". 2011.

[Alve 11b] P. Alves and P. Ferreira. "ReConMUC - Adaptable Consistency Requirements for Efficient Large-scale Multi-user Chat". In: *Proceedings of the 2011 ACM conference on Computer supported cooperative work*, 2011.

[Arau 04] F. Araujo and L. Rodrigues. "Geopeer: a location-aware peer-to-peer system". *Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings.*, pp. 39–46, 2004.

[Bald 07] M. Baldauf, S. Dustdar, and F. Rosenberg. "A survey on context-aware systems". *International Journal of Ad Hoc and Ubiquitous Computing*, Vol. 2, No. 4, p. 263, 2007.

[Bard 04] J. E. Bardram and T. R. Hansen. "The AWARE Architecture: Supporting Context-Mediated Social Awareness in Mobile Cooperation". In: *Proc. Conf. on Computer-Supported Collaborative Work (CSCW)*, pp. 192–201, Chicago, 2004.

[Bard 06] J. Bardram, T. R. Hansen, and M. Soegaard. "AwareMedia: a shared interactive display supporting social, temporal, and spatial awareness in surgery". *Proceedings of the 2006*, pp. 109–118, 2006.

[Bjer 03] E. Bjerrum and S. Bø dker. "Learning and living in the 'new office'". In: *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, pp. 199–218, Helsinki, Finland, 2003.

[Bolc 07] C. Bolchini, C. a. Curino, E. Quintarelli, F. a. Schreiber, and L. Tanca. "A data-oriented survey of context models". *ACM SIGMOD Record*, Vol. 36, No. 4, p. 19, Dec. 2007.

[Brat 07] P. Bratskas, N. Paspallis, and G. Papadopoulos. "An Evaluation of the State of the Art in Context-aware Architectures". In: *Sixteenth International Conference on Information Systems Development (ISD 2007). Springer Verlag*, Citeseer, 2007.

[Brow 97] P. Brown, J. Bovey, and X. Chen. "Context-Aware Applications: From the Laboratory to the Marketplace". *IEEE Personal Communications*, No. 4(5), pp. 58–64, 1997.

[Cano 06] J.-C. Cano, P. Manzoni, and C.-K. Toh. "UbiqMuseum: A Bluetooth and Java Based Context-Aware System for Ubiquitous Computing". *Wireless Personal Communications*, Vol. 38, No. 2, pp. 187–202, March 2006.

[Carz 01] A. Carzaniga, D. Rosenblum, and A. Wolf. "Design and evaluation of a wide-area event notification service". *ACM Transactions on Computer Systems (TOCS)*, Vol. 19, No. 3, pp. 332–383, 2001.

[Chak 06] D. Chakraborty, K. Dasgupta, and A. Misra. "Efficient Querying and Resource Management Using Distributed Presence Information in Converged Networks". In: *7th International Conference om Mobile Data Management*, Nara, Japan, 2006.

[Chak 07] D. Chakraborty, K. Dasgupta, S. Mittal, A. Misra, and A. Gupta. "BusinessFinder: Harnessing Presence to enable Live Yellow Pages for Small, Medium and Micro Mobile Businesses". *IEEE Communications*, 2007.

[Chen 00] G. Chen and D. Kotz. "A Survey of Context-Aware Mobile Computing Research". *Dartmouth Computer Science Technical Report TR2000-381*, pp. 1–16, 2000.

[Chen 02] G. Chen and D. Kotz. "Context aggregation and dissemination in ubiquitous computing systems". *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*, No. June, pp. 105–114, 2002.

[Chev 01] K. Cheverst. "The role of shared context in supporting cooperation between city visitors". *Computers & Graphics*, Vol. 25, No. 4, pp. 555–562, Aug. 2001.

[Chev 02] K. Cheverst, K. Mitchell, and N. Davies. "Exploring Context-aware Information Push". *Personal and Ubiquitous Computing*, Vol. 6, No. 4, pp. 276–281, Sep. 2002.

[Chev 99] K. Cheverst. "Design of an object model for a context sensitive tourist GUIDE". *Computers & Graphics*, Vol. 23, No. 6, pp. 883–891, Dec. 1999.

[Cowz 09] N. Cowzer and A. Quigley. "GeoIGM: A Location-Aware IGM Platform". In: *2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pp. 105–110, IEEE, 2009.

[Deme 88] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. "Epidemic algorithms for replicated database maintenance". *ACM SIGOPS Operating Systems Review*, Vol. 22, No. 1, pp. 8–32, Jan. 1988.

[Dey 00] A. Dey and G. Abowd. "Towards a better understanding of context and context-awareness". In: *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, pp. 1–6, 2000.

[Dey 02] A. Dey, J. Mankoff, G. Abowd, and S. Carter. "Distributed mediation of ambiguous context in aware environments". *Proceedings of the 15th annual ACM symposium on User interface software and technology - UIST '02*, p. 121, 2002.

[Dour 92] P. Dourish and V. Bellotti. "Awareness and coordination in shared workspaces". *in Proc ofCSCW'92, Toronto, ACM Press*, Vol. pp, pp. 107–114, 1992.

[Eise 07] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. a. Peterson, G.-S. Ahn, and a. T. Campbell. "The BikeNet mobile sensing system for cyclist experience mapping". *Proceedings of the 5th international conference on Embedded networked sensor systems - SenSys '07*, p. 87, 2007.

[Fitz 99] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, and T. Phelps. "Augmenting the workaday world with Elvin". *ECSCW 1999*, No. September, pp. 12–16, 1999.

[Foga 04] J. Fogarty. "Presence versus availability: the design and evaluation of a context-aware communication client". *International Journal of Human-Computer Studies*, Vol. 61, No. 3, pp. 299–317, Sep. 2004.

[Foga 07] J. Fogarty and S. E. Hudson. "Toolkit support for developing and deploying sensor-based statistical models of human situations". *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07*, p. 135, 2007.

[Gutw 05] C. Gutwin, K. Schneider, D. Paquette, and R. Penner. "Supporting Group Awareness in Distributed Software Development". In: *Engineering Human Computer Interaction and Interactive Systems*, pp. 383–397, Springer Berlin / Heidelberg, 2005.

[Henr 05] K. Henricksen, J. Indulska, and T. McFadden. "Middleware for distributed context-aware systems". *Internet Systems 2005*, pp. 846–863, 2005.

[High 01] J. Hightower and G. Borriello. "A Survey and Taxonomy of Location Systems for Ubiquitous Computing". *Computing*, pp. 1–29, 2001.

[Hofe 02] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, and W. Retschitzegger. "Context-awareness on mobile devices - the hydrogen approach". *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, Vol. 43, No. 7236, pp. 292–301, 2002.

[Indu 03] J. Indulska and P. Sutton. "Location management in pervasive systems". In: *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003-Volume 21*, p. 151, Australian Computer Society, Inc., 2003.

[Jova 01] M. Jovanovic. *Modeling large-scale peer-to-peer networks and a case study of Gnutella.* PhD thesis, University of Cincinnati, 2001.

[Kotz 04] D. Kotz. "Design and implementation of a large-scale context fusion network". *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.*, pp. 246–255, 2004.

[Lane 10] N. Lane, E. Miluzzo, H. Lu, D. Peebles, and T. "A Survey of Mobile Phone Sensing". *IEEE Communications Magazine*, No. September, pp. 140–150, 2010.

[Lu 09] H. Lu, W. Pan, N. Lane, T. Choudhury, and A. Campbell. "SoundSense: scalable sound sensing for people-centric applications on mobile phones". In: *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pp. 165–178, ACM, 2009.

[Milu 08] E. Miluzzo, N. Lane, K. Fodor, R. Peterson, and H. "Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application". In: *6th ACM conference on Embedded network sensor systems*, p. 337, ACM Press, New York, New York, USA, 2008.

[Oula 05] A. Oulasvirta, M. Raento, and S. Tiitta. "ContextContacts". *Proceedings of the 7th international conference on Human computer interaction with mobile devices & services - MobileHCI '05*, p. 167, 2005.

[Pasc 98] J. Pascoe, N. Ryan, and D. Morse. "Human-Computer-Giraffe Interaction : HCI in the Field". *Technology*, 1998.

[Plal 03] B. Plale and Y. Liu. "Survey of Publish Subscribe Event Systems". *Technical Report TR574, Indiana University*, 2003.

[Prek 03] P. Prekop and M. Burnett. "Activities , Context and Ubiquitous Computing". *Computer Communications*, pp. 1–14, 2003.

[Prin 99] W. Prinz. "NESSIE: an awareness environment for cooperative settings". *ECSCW'99*, pp. 391–410, 1999.

[Priy 00] N. Priyantha, A. Chakraborty, and H. "The cricket location-support system". *Proceedings of the 6th*, Vol. 2000, No. August, pp. 32–43, 2000.

[Quin 86] J. Quinlan. "Induction of decision trees". *Machine learning*, Vol. 1, No. 1, pp. 81–106, Oct. 1986.

[Raen 05] M. Raento, a. Oulasvirta, R. Petit, and H. Toivonen. "ContextPhone: A Prototyping Platform for Context-Aware Mobile Applications". *IEEE Pervasive Computing*, Vol. 4, No. 2, pp. 51–59, Apr. 2005.

[Rand 00] C. Randell and H. Muller. "Context awareness by analysing accelerometer data". *The Fourth International Symposium on Wearable*, pp. 175–176, 2000.

[Ratn 02] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. "Ght: A Geographical Hash Table for Data-Centric Storage". *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications - WSNA '02*, p. 78, 2002.

[Redd 07] S. Reddy, A. Parker, J. Hyman, J. Burke, D. Estrin, and M. Hansen. "Image browsing, processing, and clustering for participatory sensing: Lessons from a dietsense prototype". In: *Proceedings of the 4th workshop on Embedded networked sensors*, pp. 13–17, ACM, 2007.

[Rich 03] S. Riché and G. Brebner. "Storing and accessing user context". *Mobile Data Management*, pp. 1–12, 2003.

[Sain 05] P. Saint-andre and R. Meijer. "Streaming XML with Jabber/XMPP". *IEEE Internet Computing*, Vol. 9, No. 5, pp. 82–89, Sep. 2005.

[Sait 02] Y. Saito and M. Shapiro. "Replication: Optimistic approaches". *Hewlett-Packard Labs Technical Report HPL-2002*, 2002.

[Salb 99] D. Salber, A. Dey, and G. Abowd. "The context toolkit: Aiding the development of context-enabled applications". In: *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, p. 441, ACM, New York, New York, USA, 1999.

[Sant 09] A. Santos, L. Tarrataca, J. Cardoso, D. Ferreira, P. Diniz, and P. Chainho. "Context inference for mobile applications in the upcase project". *MobileWireless Middleware, Operating Systems, and Applications*, pp. 352–365, 2009.

[Saty 01] M. Satyanarayanan. "Pervasive computing: vision and challenges". *IEEE Personal Communications*, Vol. 8, No. 4, pp. 10–17, 2001.

[Sawy 98] S. Sawyer and P. J. Guinan. "Software development:processes and performance". *IBM Systems Journal*, Vol. 37, No. 4, 1998.

[Schi 94] B. Schilit, N. Adams, and R. Want. "Context-aware computing applications". In: *Mobile Computing Systems and Applications, 1994. WMCSA'08. First Workshop on*, pp. 85–90, IEEE Comput. Soc. Press, 1994.

[Schm 99] a. Schmidt, M. Beigl, and H. Gellersen. "There is more to context than location". *Computers & Graphics*, Vol. 23, No. 6, pp. 893–901, Dec. 1999.

[Sega 00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. "Content based routing with elvin4". In: *Proc. AUUG00*, 2000.

[Sega 97] B. Segall and D. Arnold. "Elvin has left the building: A publish/subscribe notification service with quenching". *in Proceedings AVVG 1997, Brisbane, September*, 1997.

[Stie 06] T. Stiefmeier, C. Lombriser, D. Roggen, H. Junker, G. Ogris, and G. Tröster. "Event-based activity tracking in work environments". In: *Proceedings of the 3rd International Forum on Applied Wearable Computing (IFAWC)*, Citeseer, 2006.

[Van 00] K. Van Laerhoven and O. Cakmakci. "What shall we teach our pants?". *Digest of Papers. Fourth International Symposium on Wearable Computers*, No. c, pp. 77–83, 2000.

[Van 01] K. Van Laerhoven. "Combining the self-organizing map and k-means clustering for on-line classification of sensor data". *Biological Cybernetics*, No. c, pp. 464–469, 2001.

[Wang 07] F.-Y. Wang, K. M. Carley, D. Zeng, and W. Mao. "Social Computing: From Social Informatics to Social Intelligence". *IEEE Intelligent Systems*, Vol. 22, No. 2, pp. 79–83, March 2007.

[Welb 05] E. Welbourne, J. Lester, A. Lamarca, and G. Borriello. *Mobile Context Inference Using Low-Cost Sensors*, pp. 254–263. 2005.

[Widr 94] B. Widrow, D. Rumelhart, and M. Lehr. "Neural networks: applications in industry, business and science". *Communications of the ACM*, Vol. 37, No. 3, pp. 93–105, Nov. 1994.

[Zand 10] S. Zander and B. Schandl. "A Framework for Context-driven RDF Data Replication on Mobile Devices". *Language*, 2010.