Chapter 2
Taxonomy

Abstract This chapter introduces some basic definitions related to context-aware systems and presents a taxonomy for such systems. Both are useful in the following sections. The taxonomy is well suited for guiding the architectural decisions of application developers; it is built around the four main layers found in context-aware systems considering context data from the moment it is acquired by sensors in raw format to the moment it is consumed by the end-user application: capture, infer, distribution, and consume. In this chapter we address each one of such layers.

2.1 Basic Definitions

This section introduces some basic definitions: context, context-aware systems, and distributed context-aware systems.

2.1.1 Context

The word "context" is subject to multiple interpretations and has been researched in various fields like psychology, philosophy, and computer science [12]. In the last field, specifically in the area of computer-supported collaborative work (CSCW), context was initially perceived as user location [14, 85] 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 [81].

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 [1] 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 [15] 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 [83] 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 [22], 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 [1]. 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, e-mail 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.

2.1.2   Context-Aware Systems

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

Schilit [85] 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 [64]. In fact, there is a classical debate in this area between

Fig. 2.1 Typical
context-aware architecture

| end-user application |
| middleware (optional) |
| sensor |
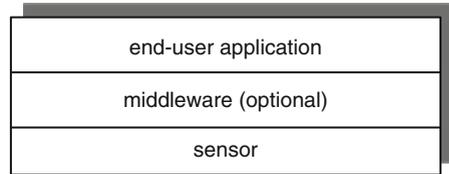
"use context" advocates and "adapt to context" advocates. Although research has
been more active on systems that adapt to context, even an application that simply
displays the context of user's environment can be considered context-aware, even
though it is not modifying its behavior.

Dey [1] 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
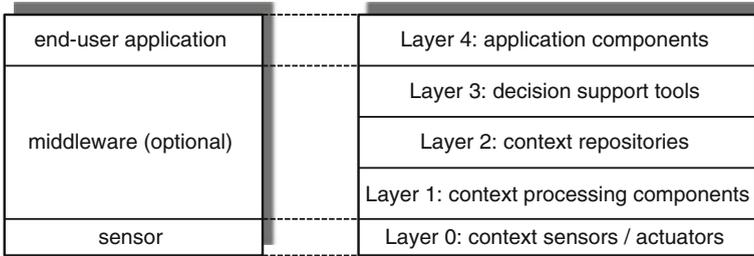to the user, where relevancy depends on the user's task.*

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 provide 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, and 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 others to see).

It is noteworthy that a context-aware application does not 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.

2.1.3 Distributed Context-Aware Systems

Distributed 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 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 [45] (see Fig. 2.1).

| end-user application	Layer 4: application components
middleware (optional)	Layer 3: decision support tools
	Layer 2: context repositories
	Layer 1: context processing components
sensor	Layer 0: context sensors / actuators

Fig. 2.2 Layers of a context-aware application according to Henricksen

In simple context-aware systems, the end-user application communicates directly with the sensor (e.g., a mobile phone 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 distributed context-aware applications, improve maintainability, and promote reuse. Henricksen [45] enumerates the following five layers, as shown in Fig. 2.2 (relation to previous figure layers also indicated):

- 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);
- 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 Fig. 2.3). Local systems are systems in which sensors and applications are tightly coupled (usually through a direct physical connection). For example, a mobile phone application that sets the mode to silent, while its owner is jogging, is a local system, since the accelerometer that provides the "running" context is directly attached to the mobile phone as well as the application that uses that context to activate the silent mode.

On the other hand, distributed context-aware 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 mobile phone
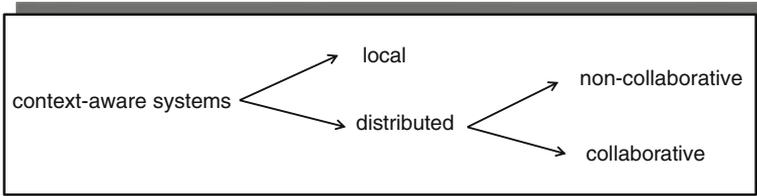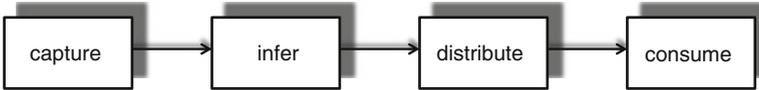
Fig. 2.3 Types of context-aware systems

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 context-aware 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 [40] provides real-time availability information within a group of colleagues, using speech detection, location, computer activity, and calendar entries. In this case, the common goal 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 [15] 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.

2.2 Taxonomy Rationale

As explained in Sect. 1.4, existing 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;

Fig. 2.4 Layers of a context-aware system

- 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 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 Fig. 2.4.

  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 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.

2.3 Capture

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 [7]. Sensors have been traditionally classified in two dimensions that, although differently named, are very similar. Prekop [66] calls these dimensions *external* and *internal* and Hofer [47] refers to *physical* and *logical*. We adopt the names proposed by Indulska [49], *physical* and *virtual* sensors, as depicted in Fig. 2.5.

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 [46] and Indulska [49]. The latter 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

Fig. 2.5 Capture layer
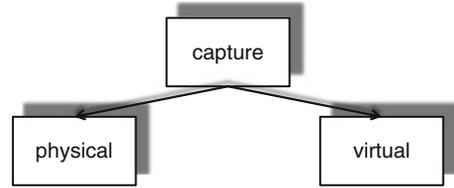
sensors provide the location within a region (e.g., mobile phone cells). Note that both modes have different accuracy levels, ranging from centimeters to tens of meters.

- **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 [22] 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 [68] 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 [68]) allow greater end-user control over publishing their location in the system.

Virtual sensors acquire context data from software applications, operating systems, and networks [49]. Detecting new appointments on an electronic calendar and watching the file system for changes are examples of virtual sensors. These sensors can also be used to infer *location*. Indulska [49] gives some examples such as using a travel-booking system or the IP address of the active device[1] 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 be tracked easily 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.

---

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

Table 2.1 Types of context and corresponding physical and virtual sensor

	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*: smart card 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

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 (e.g., databases) in order to produce higher-level context data [7, 49]. 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 [49] 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 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 will not be any sensing conflicts.

Different sensors can provide different types of context. According to Dey [1], 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 2.1 we show examples of physical and virtual sensors grouped by the type of context they are able to capture.

In Table 2.2, we show some context-aware applications and their corresponding sensors. Table 2.2 reflects a general trend in context-aware applications: while early applications were predominantly based on location sensors, researchers have been recently using other types of sensors to infer richer context information such as user activities (walking, running) and even mood (sad, happy) [82].

2.4 Infer

The *Context-Inference* layer, also known as the Preprocessing layer [7], 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,

Table 2.2 Some context-aware applications and corresponding sensors

	Description	Sensors used
GUIDE [25]	Information for city visitors	WiFi (location)
Welbourne [97]	Mode of transit: walking, running, riding a vehicle	Clock, GSM/WiFi (location), accelerometer
ContextContacts [62]	Enhanced phone's contact list	GSM, phone activity, Bluetooth (nearby environment)
UbiqMuseum [15]	Assists museum visitors	Bluetooth (location)
AwareMedia [9]	Support coordination at an operation ward	Bluetooth (location), video camera, shared calendar
Stiefmeier [91]	Help workers perform critical and complex assembly tasks in a car production environment	Body-worn, car-mounted, and tool-mounted accelerometers
BikeNet [36]	Cyclist experience mapping	Magnetometer, inclinometer, speedometer, microphone, GPS, GSR stress monitor, CO^2 meter
CenceMe [59]	Social activities (dancing, lunching)	Microphone, GPS, camera, Bluetooth (nearby environment), accelerometer
SoundSense [57]	Music events' sharing	Microphone, camera, GPS
Upcase [82]	Daily activities (working, driving, sleeping, resting, walking outside)	Luminosity, microphone, temperature, accelerometer

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 [86] 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, wavelength, etc.). Further inference can be made using these features. Zander [100] uses the term "context provider" as something that converts any kind of input data (either sensorial or web-based content) to an RDF-based context description.

Also, this layer is normally associated with classification techniques, mostly borrowed from Artificial Intelligence (AI) algorithms, such as Kohonen self-organizing maps (KSOMs) [54], k-nearest neighbor [94], and neural networks [74]. These techniques have been successfully applied to some context domains such as inferring the user activity (walking, running) from an accelerometer [97].

Fig. 2.6 Infer layer
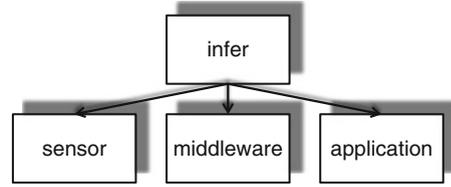
Context-inference techniques is an active topic of research which we do not detail in this book. Since we are analyzing *distributed* context-aware systems, we classify these systems based on *where* the inference occurs.

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 [97], while another may be more interested in detecting screw tightening [91]. 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 mobile phones. To avoid this problem, some systems use a middleware component (e.g., Solar [23], PACE [45]), 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 [81] 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 Fig. 2.6, 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 wide area network, 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 [99] or decision trees [71] 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 (e.g., 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 as 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 does not 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 [81].
- **Personalization**—Somehow opposed to reusability is the ability to personalize the inference engine to better suit individual needs. For example, CenceMe [59] 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 [33]. Sometimes, context inference includes dealing with ambiguous data and explicit user mediation is needed (the application prompts the user to resolve ambiguity) [39], again adapting the inference mechanism to suit individual needs.

Figure 2.7 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 a hybrid approach where the inference engine resides in multiple places. Miluzzo [59] proposes a split-level classification for its CenceMe system, pushing some classification to the phone and other to the back-end 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 [23], in the Solar system, proposes an interesting solution to achieve personalization without sacrificing reusability. 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
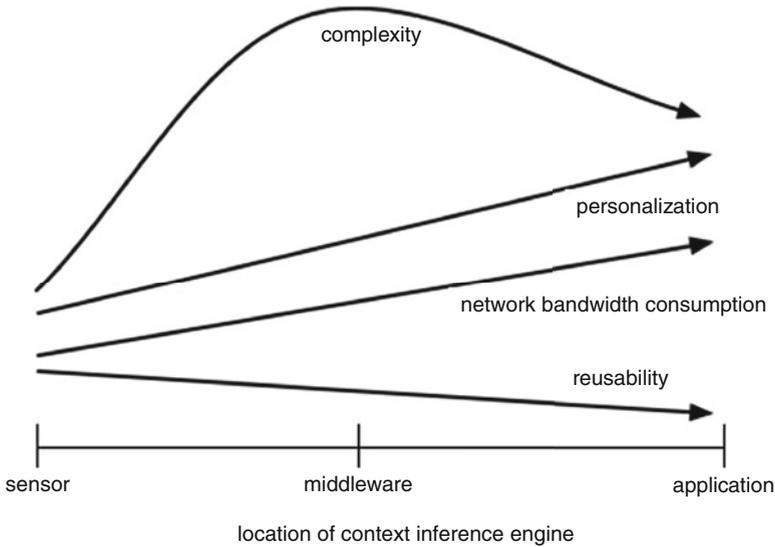
Fig. 2.7 How context-aware systems properties are affected by the location of the context-inference engine

applications, as shown in Fig. 2.8. 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.
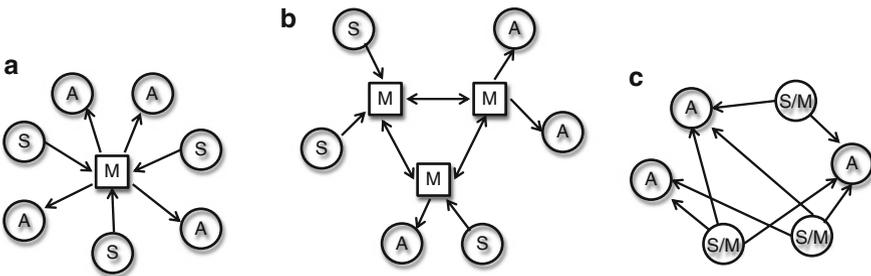
2.5 Distribution

Independently of how context is captured or inferred, it has to be distributed among the system components (sensors, middleware, application). Some articles refer 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. In this section, we focus solely on how context is distributed in the system.

The most common approach for distributed context-aware systems uses a centralized middleware as a broker between dispersed sensors and client applications (see Fig. 2.9a). For example, CenceMe [59] shares personal context obtained from mobile phones through 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 is a single point of failure and thereby lacks robustness. Even if the server does not fail, its scalability is limited—as more devices use the
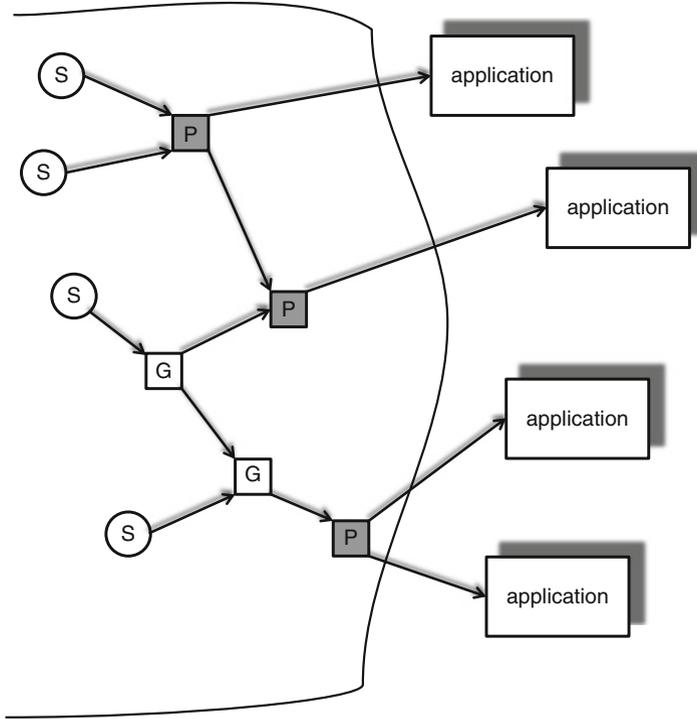
Fig. 2.8 The Solar system pushes application-specific processing into network proxies (represented by *dark squares*) that can be combined and reused by other applications (S=sensors; *white squares*=generic context processors)

Fig. 2.9 The three types of distributed organization for context-aware systems (M, middleware; S, sensor; A, application): (**a**) centralized, (**b**) partitioned, and (**c**) peer-to-peer

system, its performance will start degrading. Also, it is not well fitted to scenarios where network connection is intermittent as it happens with mobile phones. Unless the application uses local caching mechanisms, it may stop working when the connection with the server drops.

Some systems try to solve some of the above mentioned problems by using multiple servers [16, 38]. 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 Fig. 2.9b. The servers communicate directly with each other, propagating information from devices in one partition to other partitions. This distributes the load among the servers, increasing the system scalability. For example, the ContextContacts application [62], which enhances the traditional mobile phone 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 identifier is like an e-mail 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 with the centralized approach, 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 [20] addresses 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 [50], where each device communicates directly to the other devices (see Fig. 2.9c). 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 [47] 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 (e.g., 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 [75] and IGM [31]. The advantage of these systems over traditional peer-to-peer architectures is that nodes know their location and the location of nearby nodes, which allows efficient geographical routing (e.g., a notification can be sent to all nodes that are within a predefined geographic region). 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 these results to the user in New York [31].

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 mobile phones). 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 [9], an awareness tool to help hospital staff coordination. AwareMedia is developed using SIENA [16], which features a hybrid approach that mixes the partitioned and the peer-to-peer models. In SIENA, devices are distributed within partitions, with each partition having an associated 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 [36] 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 then transmitted to the central server.

2.6 Consume

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 2.10 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 that has the information
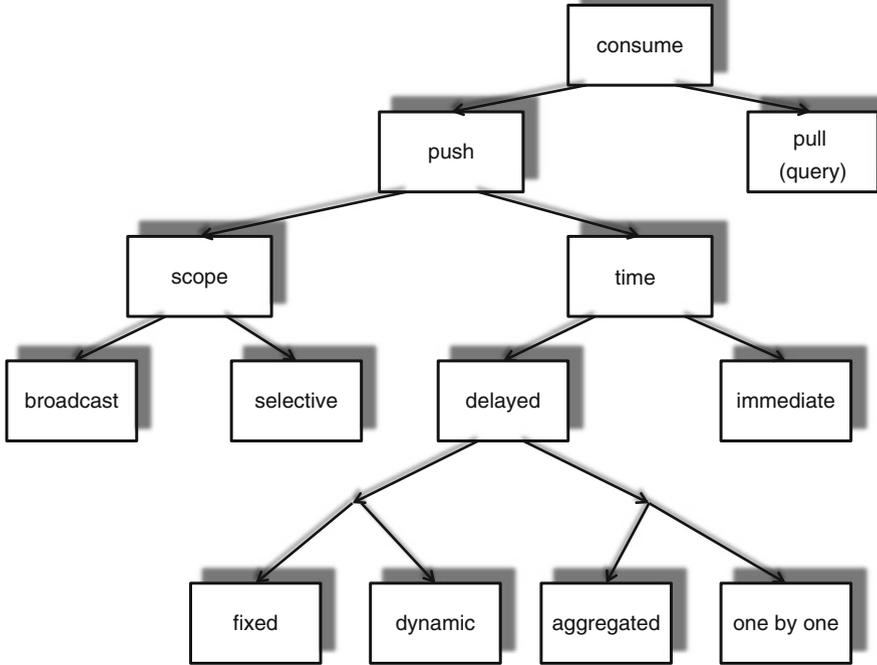
Fig. 2.10 Consume layer

(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 min). 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 to the user. Also, this periodicity (polling interval) can change according to the application current needs. For example, an application running on a mobile phone 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 min, there will be cases where the application receives context information from 5 min 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 is no new context information available. Since the application does not

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 than pull-based mechanisms. Since the component that 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 [27] argues that applications that use push-based mechanisms may surprise the user, disrupting his 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 [26] that was originally designed following the pull model, whereby the onus was on the users to request the presentation of context-aware information.[2] 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 showed that visitors were comfortable with the push-based version, because it required less effort to learn and use than the pull-based version.

2.6.1   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 [8] (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 B in their contact list are not notified. AwarePhone is built on top of the AWARE framework [8], 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 mobile phones), and HTTP. HTTP, due to its stateless request-response nature, is the only one that does not support push-based context propagation. ContextPhone [72] uses a similar mechanism on top of the XMPP protocol [79].

---

[2]For example, by tapping on the information button.

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 Fig. 2.10).

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 [88] 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.[3]

2.6.2 Time

Another way to improve push-based systems scalability is to assume that certain context information does not 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 is not 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) loses 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 [80]. 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 masters and end-user applications acting as slaves. Since sensors are the only components that *write* context information, these systems do not suffer from conflicting updates, which is 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, Riché [77] proposes a system for managing user context

---

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

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.

Delayed context propagation can be further categorized based on the *interval between updates* and the *amount of information transmitted on each update* (see Fig. 2.10):

- **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 [6] 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 from each other almost immediately while context information from people in the vicinity, but not in the building, would be received with a certain delay.[4]

- **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 was not any delay. However, it can also be extremely inefficient.

  Consider the case of a moving person 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 min (i.e., a maximum delay of 1 min 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 bandwidth 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 [6].

  Note that, in context-aware related literature, the term "aggregation" is often associated with the combination of information from different sensors, along

---

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

with conflict resolution. Such *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.

2.7   Summary

We started this chapter with some basic definitions regarding context-aware systems. In particular, we say that a system is context-aware if it uses context to provide relevant information and/or services to the user. Then, we presented a taxonomy for distributed context-aware systems which is based on the following main layers: capture, infer, distribute, and consume. 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. For each one of these layers we detailed the several techniques that can be used.

Springer