Chapter 2
Foundations of Context Management in Distributed and Dynamic Environments

Abstract Context information is data that describes the state of a certain entity at a specific moment. A context management system is a computational element responsible for binding context providers, which produce context information, and context consumers, typically represented by context-aware applications. The main task of a context management system is to match consumer's interests with probed context information. The complexity of context management in a distributed scenario is defined by the wideness of an interest, i.e. the number of context management systems that should be involved in an interest matching. If a distributed scenario is also open, heterogeneous and dynamic, than the wideness of an interest is variable, as a result of characteristics such as dynamic introduction of new sensors and evolution of context models. The support of context interest of variable wideness imposes challenging requirements for context management systems.

Keywords Distributed context management · Context-aware computing · Middleware · Open distributed systems

2.1 Introduction

In context-aware applications, adaptations are triggered by changes of certain context information. For example, smart applications designed to support meetings may automatically transfer a presentation to a projector as soon as the presenter enters the meeting room [5]. In this case, both the location of the presenter and his/her role in the meeting room are basic pieces of context information used to trigger the transfer of the presentation. Basically, the development of a context-aware application, as in this example, involves the description of the actions to be triggered according to a set of contextual conditions.

A same piece of context information may be used for different purposes. The location of the presenter, for example, may also be used by another application to disseminate his availability status for an instant communicator. Moreover, this context information may be provided by different sensors, such as a proximity sensor to identify if the user is inside the classroom and using a microphone connected to a voice recognition software to identify specific users in the classroom, as in [22]. This requirement of reuse calls for middleware systems to enable context-aware computing, instead of requiring that applications be developed from scratch.

The main goal of middleware in context-aware computing is to enable decoupled communication between sensors that provide context data and applications interested in context information. Most middleware systems have developed mechanisms that ease incorporation of sensors (e.g., ContextToolkit [6]), and enable high-level description of context conditions that applications are interested in, thus avoiding applications having to poll sensors. Typically, these middleware systems adopt asynchronous communication mechanisms, such as publish/subscribe [7] or tuple-space systems [23], as the basis of interactions among sensors and applications. These mechanisms allow applications to register interests in context information and to asynchronously receive notifications of events that match their interests. RCSM [24], Confab [13], PACE [10] and MoCA [19] are examples of middleware systems that adopt such communication paradigm. Even higher-level programming abstractions for context-aware computing, such as *profiles* [24] and *preferences* [9], require lower-level mechanisms based on asynchronous notifications. In fact, asynchronous communication is the most elementary mechanism of context-aware middleware systems, which is in charge of three main tasks: storage of context information, management of application's subscriptions, and dissemination of events that represent a situation of interest. Some systems delegate this management task to general-purpose asynchronous event systems, which constitutes the context management layer of most middleware systems, as proposed by Henricksen and Indulska [9]. However, general-purpose asynchronous systems do not satisfy adequately the requirements to enable context-aware computing in a distributed and dynamic scenario. In general, publish/subscribe systems focus only on efficient event dissemination and routing in a distributed scenario.

This chapter discusses this challeging problem using the following organization. Section 2.2 defines the foundational concepts of context management. Sections 2.3 and 2.4 present the conceptual layers of context interest management and discusses challenges of enabling context management in a distributed and dynamic environment, respectively. These challenges call for a new class of context interest called *interest of variable wideness*, as presented in Sect. 2.4.2.

2.2 General Concepts

In order to exemplify the general concepts of context-aware computing, consider the following running example:

UMessenger is a location-aware messaging application that enables communication of a user with a group of people (his buddies), integrating functionalities of a mobile phone and of an instantaneous communicator. By knowing the position of his/her buddies on a map, the user can initiate location-oriented interactions based on their location. The user can also define location-based notification conditions, e.g., "tell me when buddy x arrives at home". The location of the user and his/her buddies are obtained from GPS sensors on their devices. The map is obtained from a geolocation map service. *UMessenger* has also the ability to adapt the communication mechanism (e.g. voice, video, asynchronous and synchronous messages) to the current device's network connectivity.

2.2.1 Context, Entity, Types and Instances

In a context-aware application, any interaction is based on two elementary concepts: *entity* and *context information*, as defined below.

Entity	is any object that has a state and that can be represented in a computational environment, such as a physical object, a user, or computational resource.
Context Information	is an abstract information that describes the state of an entity.

In the *UMessenger* example, *location* and *network connectivity* are pieces of context information that characterizes the state of the entity *user device*. Hence, the device's state at a specific instant could be: *(location = home)* and *(network connectivity = using wired network)*. For the sake of simplicity, consider that the user's device location in fact represents the user's location. This definition of context information is consistent to the definition already proposed by Dey [6]. Context-aware systems implement context information through *context types* and *instances* of these types.

Context type	is a computational implementation of a context information which specifies, at least, its data structure.

For example, to represent the data provided by the GPS sensor, the *UMessenger* may implement a `GPSLocation` type composed of three float numbers: *latitude*, *longitude* and *elevation*.

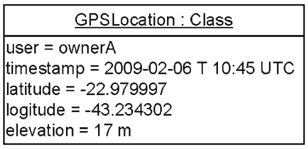

Fig. 2.1 Example of a Context Instance

A middleware may adopt various types to represent an abstract context information. For example, location may have various representations [12], such as symbolic location [17] (e.g., *RoomA*, *BuildingFPLF*) and proximity-based location [18]. As a result, each representation could be modeled in a particular context type. However, an application may be only prepared to deal with some of these types. For example, if the *UMessenger* is prepared only to display the location on a map based on geo coordinates, then a location sensor that provides symbolic location will not be useful for this application.

This book uses the term *context instance* to describe a piece of data that contains context information, as defined below.

Context instance is a value or an aggregate of values that describes the state of an entity at a specific instant of time and which conforms to a certain *context type*. A context instance $\boldsymbol{i}$ is an object of context type $\boldsymbol{T}$ defined by the tuple $C_i^T = (e, t, V_T)$, where

- e: the entity.
- t: a timestamp.
- V_T: a set of values for each attribute defined in type T.

A `GPSLocation` instance could be described by the tuple shown in Figure 2.1. A context instance is a snapshot of the state of an entity, at a specific instant of time. The relationship between a context type and an instance is similar to the relationship between a class and an object in the object-oriented programming paradigm. Although the concept of *context information* is an abstraction of *context instance*, for an implementation of a context-aware system, these two terms can be used interchangeably.

2.2.2 Context Model and Modeling Approach

Context Model A context model determines the set of all context types and entities, and relationships among them.

The definition of a context model is a part of the implementation of a context-aware system. A context model defines relevant concepts to the application domain, which the middleware is prepared to deal with. For example, the CoBrA middleware [5] models entities such as *Agent*, *Person*, *Meeting*, *Event* and *Schedule*, which are the basis of the implementation of smart meeting applications. In the case of

UMessenger, since the application basically deals with location and resources of a device, the application should adopt a model that, at least, describes context types to represent *location* and *resources*, as well as an entity type to represent *devices*.

The expressiveness and complexity of a context model depends on the *modeling approach* adopted in the system, which defines how the concepts and their relationships are described. We define context modeling approach as follow.

Context Modeling Approach is the schema used to describe concepts and their relationships in a context model.

A context modeling approach also defines the kinds of relationships a model may support and the meaning of each relationship. An example of a simple context modeling approach is the pair key-value schema, which uses tuples of pairs $(key, value)$ to describe context information, as adopted in [20]. Using this modeling approach, an instance of `GPSLocation` would be described by the following set of pairs: $((latitude = -22.979997), (longitude = -43.234302), (elevation = 17))$.

Other examples of context modeling approaches [2, 21] are markup schema, graphical, object-oriented, logic based, ontology based and hybrid approaches (e.g. [11]), as discussed in [1].

Some modeling approaches support the formal description of how a context information is inferred from previous existing information. For example, ontology-based approaches use first-order logic to describe how a concept may be inferred from another concept.

2.2.3 Context Providers and Consumers

A context-aware scenario is composed of interations between elements that produce context, called *context providers*, and elements that consume context, called *context consumers*, as defined below.

Context provider is a computational element that populates the context-aware system with context instances of a particular type.

Context consumer is a computational element that consumes context instances to achieve some application-specific purpose.

A *context provider* translates raw data probes obtained from a low-level sensor (e.g., accelerometer, GPS sensor) into context instances on a context model. A provider is a proxy of a sensor in the context-aware system, translating raw data to information that can be used in the system. In the case of *UMessenger*, an application module may be responsible for collecting data from the GPS sensor and for creating the corresponding instances of `GPSLocation` type. The GPS sensor does not need to know the application's context model, and thus another computational element - the provider - generates the useful data for the application.

Typically, a context consumer is a context-aware application, such as the *UMessenger*, which consumes location information. A computational element may act both as a context consumer and producer, generating a new context information from another lower-level context. For example, some location positioning systems (e.g., [15, 17]) infer a location of a device from triangulation of radio frequency signal strengths from reference points (e.g., 802.11 access points). If such positioning systems model both signal strengths and location as context types, then they infer a context type from another one. This inference is called context reasoning. The external element that produces this reasoning is an inference agent.

Inference Agent	is a computational element that consumes context instances to deduce a new context of a different type. The inference agent publishes the resulting context in the system, thus also acting as a context provider.

Hence, an inference agent acts both as a context provider and a consumer.

2.2.4 Contextual Event and Context Interest

A context consumer specifies the situation it is interested in terms of *contextual events* and *context interests*.

Contextual Event	is a change in the state of one or more entities that is relevant for some consumer.

For example, the contextual event *phone is offline* could be triggered when the connection with a cellular network is no more available. Upon this event, the *UMessenger* may disable the sending of SMS messages.

Context Interest	is a representation of a class of contextual events that a consumer is interested in. A context interest $\boldsymbol{n}$ is defined as a tuple $I_n = (E, T, \varepsilon(V_T))$, where • E is a set of entities. • T is the context type • $\varepsilon(V_T)$ is a boolean function that contains a logic expression based on the values of the attributes V_T. It defines the constraint on context instances that satisfies the interest.

The complexity of $\varepsilon(V_T)$ evaluation depends directly on the context modeling approach adopted. For example, in a middleware that adopts a pair key-value modeling approach, a constraint is a composition of logic expressions based on the values of each attributes (i.e. key).

2.2.5 Context Selection and Matching

One of the core responsibilities of a context-aware system is to decide, in a set of context instances, which ones satisfy interests of each consumer, as specified by its *context matching function*. This responsibility still envolves two tasks: *context selection* and *notification composition*.

Context Matching Function is a boolean function $\boldsymbol{Match(n, i)}$ that determines if an context instance $\boldsymbol{i}$ satisfies an interest $\boldsymbol{n}$.

A context matching function is executed against context instances to check if an instance change must produce a notification for the consumer. Every return $true$ results in a notification to a consumer. Basically, a matching function is a translation of interest's $\varepsilon(V_T)$ to the context of the computational element responsible for an interest matching. The complexity of a matching depends directly on the modeling approach adopted. For example, for a pair *key-value* approach, the matching is a comparison of the values of the keys that appear in the interest expression. For an ontology-based approach, the matching is based on the execution of inference on ontology models. In general, the more flexible the matching is, the more complex the implementation of the matching function becomes.

The matching function must be executed when there is a change in the state of an instance, which may correspond to a contextual event.

Context Selection is the task of selecting a subset of context instances to which an interest applies.

A context selection function determines the context instances that should be applied to an interest match, according to the interest specification. In theory of event-based systems, context selection is corresponding to a task called *event filtering* [16]. The complexity of context selection depends on the modeling approach adopted in the system and, in general, defines the class of context the consumer is interested in. Context selection typically depends on the implementation of the underlying asynchronous communication mechanism that a middleware adopts. For example, in a middleware that adopts context management based on a topic-based [7] publish/subscribe system, context selection is based on the topic used in subscriptions. In general, context selection is based on the entity and additional properties of a context interest. In the aforementioned example, the additional property is the topic name.

Notification Composition is the task of choosing the more appropriate notification resulting from an interest match, when more than one context instance satisfies an interest.

Multiple matches may occur when more than one context provider produces the same context information. The resulting information may be complementary or inconsistent, so a consumer must use only one of the notifications to trigger their adaptation. In general, an application may use meta-attributes or quality-of-context information, to select the notification that is more appropriate for an application. For example, an application may specify that it is only interested in the most trustable notification.

Some middleware systems have developed mechanisms to deal with these notification conflicts, such as PACE [10] and CARISMA [3]. When the middleware is responsible for the notification composition, the interest description must support such meta-attributes. Notification composition may be implemented as a part of the middleware or as an external element, as another middleware component or the application.

2.2.6 Context Management System

A *context management system* (CMS) is an independent computational infrastructure that enables interactions among context providers and consumers, as define below.

Context Management System	A context management system (CMS) is an architectural component in context-aware computing responsible for storing context information, published by context providers, and matching previously registered interest to context instances.

A CMS must both store context instances published by providers, as register context interest of consumers, and check them against context instances, thus executing context selection and matching. A CMS is also responsible for managing the context model, validating the consistency of interest and instances according to the model. The underlying modeling approach plays an important role in defining the complexity of implementing the CMS. Depending on the context model approach, management of a model may be resource-intensive. For example, ontology-based models require constant execution of inference rules, which usually degrades the performance of the CMS.

Four main elements characterize a CMS, as illustrated in Fig. 2.2: *primitives*, the *context model*, *context interests*, and *context instances*. Each CMS is responsible for a particular set of context interest and instances, as result of context providers and consumers interacting with it. The CMS's primitives correspond to the interaction paradigm, context modeling approach and the underlying communication middleware on which the CMS is based. An application that interacts with a CMS, is capable for interacting with any other CMS that adopts the same primitives.

Some middleware systems, such as Nexus [8], support heterogeneity among CMS's context models, i.e. each CMS can adopt a particular context model.

Table 2.1 shows the primitives of interaction with a CMS, considering only the asynchronous mode of operation, which is the focus of the work presented in this book.

A CMS may be implemented as a set of distributed infrastructural components, such as proposed in [4]. However, to adhere to the proposed definition, the CMS

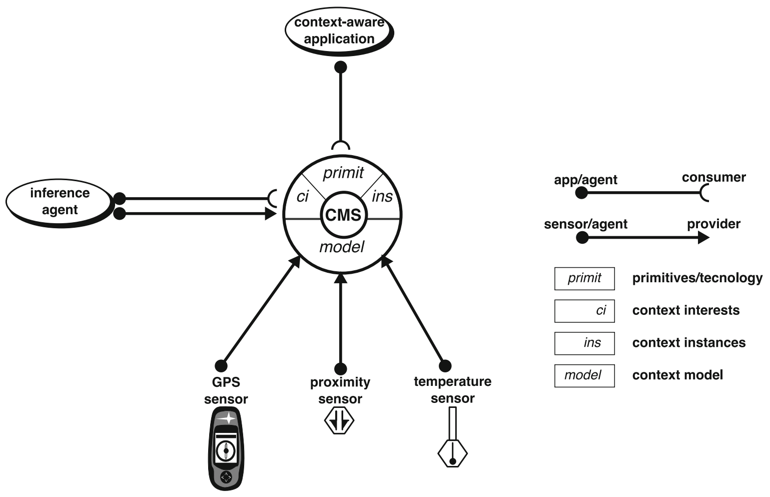

Fig. 2.2 Diagram of a CMS structure and its interaction with providers and consumers

Table 2.1 Main Asynchronous Primitives of Context Management Systems

Operation	Direction	Meaning
`publish`	Provider → CMS	publishes a context information
`registerInterest`	Consumer → CMS	registers an interest
`notifyMatching`	CMS → Consumer	notifies a consumer that a previously registered interest has matched to a context information
`unregisterInterest`	Consumer → CMS	unregisters a context interest

distribution must be totally transparent for providers and consumers, that address the CMS through the same addressing abstraction. Thus, in the perspective of consumers and providers, there is no difference between accessing distributed CMSs and accessing a unique CMS. Section 2.3 discusses a scenario of distributed CMSs and the role of middleware to confer transparency to such distribution.

2.3 Conceptual Layers of Context Interest Management

The support of context interest in distributed environments, i.e. distributed CMSs, brings up challenges in terms of context management infrastructures and programming abstractions, besides the traditional problems of scalability and distribution[1]. The goal of a middleware for open and evolutionary scenarios is to support context

[1] E.g., event notification routing and mobility management.

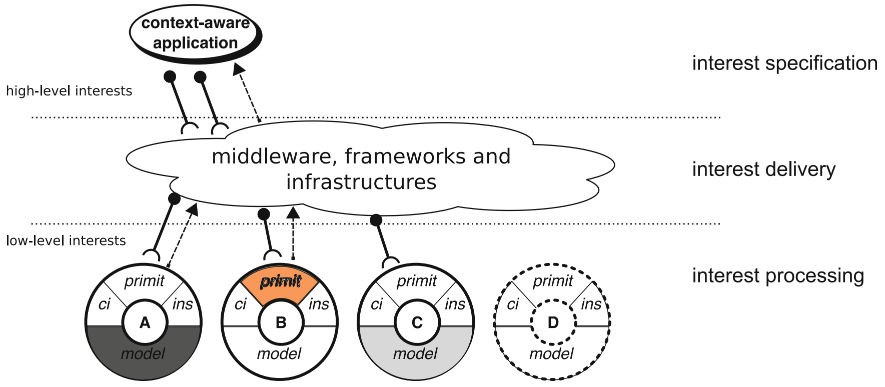

Fig. 2.3 Layers of interest implementation on context-aware ecosystems

interest without increasing the application's complexity. Middleware systems should make transparent the diversity and distribution of a CMS.

Figure 2.3 shows distributed context management systems organized in the conceptual layers. These layers range from the application to CMSs responsible for managing and storing context information. In the figure, the different shading of each CMS's component represents heterogeneity in such aspect among the CMSs. For example, CMSs *A*, *B* and *C* adopt a particular context model, whereas *A* and *C* adopt a same primitive which is different from *B*'s primitives. In addition, the different style of CMS *C* represents that the CMS is not included in the processing of the application's interest.

The *interest specification layer* comprises applications, frameworks and middleware systems that specify and register context interests and that receives notifications when a context data matches an interest. The *interest processing layer* comprises CMSs responsible for storing context information and matching context interests. The *interest delivery layer* comprises middleware infrastructures that route context interests to the corresponding CMS and deliver notifications to the corresponding clients. It also may implement transparent access to distributed CMSs and translate a higher-level interest to lower-level interest. The set of computational elements of all interest layers is called *context-aware ecosystem*.

Several context providers may provide the same context type regarding a specific entity, and those may change dynamically. Thus, an application may need to register its interest on several CMSs that are responsible for that interest, or require that the interest delivery layer translates its high-level interest to the corresponding lower-level interests. The interest translation needs to conform the context model of each CMS, if they are heterogeneous.

The interest delivery layer may also need to implement a distributed notification composition if more than one CMS disseminates notifications for the same interest match. This composition may either be done on the specification layer (application's side) or in the delivery layer. The drawback of the first case is that it increases application complexity.

Consider the case of *UMessenger*. To obtain the updated location of each buddy, the application needs to register in any CMS that stores the location of a buddy. Peer-to-peer messaging applications tipically obtain references to peers to connect and their current connectivity states from a centralized server. This architecture could be suitable for *UMessenger* if location is limited to a GPS embedded sensor. However, a distributed scenario for context management suggests the implementation of a more flexible application, which we call *UMessenger 2.0*, as described below.

UMessenger 2.0 is an extension of *UMessenger* that can work with flexible semantics of location information and different location providers. In the default mode of operation, *UMessenger 2.0* obtains a map from a centralized map provider, as in the previous application version. In this mode, *UMessenger 2.0* describes interests for geo locations of all user's buddies, specifying a preference to obtain location from the most precise provider, which is typically a GPS provider. If `GPSLocation` is not available, e.g., the user is not using a GPS-enabled device or the user is in an indoor environment, the application shows the location using other alternative providers (e.g., E911 and Active Bats). If the location provider is based on an indoor position system, as in Active Bats, the application provides an option to the user to switch the map view to the view directly associated to the provider (e.g., a building map for an indoor positioning system). Then, the application starts showing the buddy's location according to this new map view, so a buddy who is not present in the area covered by the map will not be shown. *UMessenger 2.0* still enables the corresponding location-based notifications, using the *place* semantic of the new map view: instead of geo locations, semantic locations, such as *Room510* and *5thFloor*. When required, the user can switch to the previous or another map view. *UMessenger 2.0* still maintains the ability to adapt the communication mechanism (e.g., voice, video, async/synchronous messages) to the current device's network connectivity.

In this scenario, an application may need to specify broader or narrower interests, in terms of the CMS involved in the resolution and the context types that satisfy the interest. The complexity of managing an application depends on how broad are its context interests. An interest is more abstract if it involves context managed in more CMSs and if its type is implemented by specific means in more than one CMS. For example, an interest in the `Location` of a `Person` p is more abstract than an interest in the `GPSLocation` of a `Buddy` p of user u, although both contexts may describe locations of the same person p. Whenever interests are more abstract, applications may need to specify more context interests at different CMSs to describe the condition that triggers the intended adaptation. Such concepts may need to be translated to context model of each CMS. Consequently, the notification composition involves more interest matches.

2.4 Context Interest Management in a Dynamic Context-Aware Ecosystem

In a dynamic context-aware ecosystem, the components of each interest layer may change, as the result of the evolution of the whole ecosystem. Such changes may compromise the consistency of interests and cause disruptions in a context-aware interaction. The main issue regarding this scenario is how to support a context interest that involves more that one CMS. Composing isolated CMSs do not enable to deal with the challenges of this scenario with efficiency and achieving scalability. Furthermore, by supporting a dynamic ecosystem, instead of just isolated CMSs, applications may describe more complex interests. As the ecosystem grows in size, the complexity of dealing with context interests increases, since CMSs may be dynamically introduced or changed.

Five characteristics make challeging the implementation of distributed and dynamic context-aware ecosystems:

- Dynamic deployment of new context providers,
- Dynamic deployment of new context types,
- Scoping of context models,
- Lack of in-advance knowledge of CMS, and
- Dynamic deployment of new CMSs.

The following paragraphs discuss each of these characteristics.

Dynamic Deployment of New Context Providers

New sensors may be constantly introduced in the ecosystem, as a result of the development of new devices, more precise sensors, new sensing mechanisms or new inference mechanisms. If a new sensor provides context involved in an interest, then it must be included in the interest matching. From the point of view of an application, perceiving a new sensor means to have the provided context included in the interest selection and matching. On the matching of interests, running applications with alive interests should be able to recognize the new provider, without requiring them to be restarted, recompiled or redeveloped.

The need to perceive a sensor may be the result of client mobility. For example, if a device enters in a physical environment that provides location sensors, than any context consumer interested in the device's location must have its interests also registered in the corresponding CMS.

Table 2.2 Example of context location providers (sensors) and the placement of their CMS

Provider	CMS Location	Applicability
GPS	Locally placed	Outdoor
E911	Cellular network	Indoor/Outdoor
ActiveBadges	Building infrastructure	Indoor

Dynamic Deployment of Context Types

As the result of the introduction of new sensors, the context model may also need to conform to particularities of the new provided context information. For example, location sensors that provide geographic coordinates and relative location must have different representations in the context model. Although both describe a location, the structure of the provided context is completely different. If a consumer can deal with the information of the new provider/type, then his/her active interests for an already existing type must include the new type in the interest matching.

Scoping of Context Models

Applications should be prepared to describe context interests based on various context models, and some of them may be restricted or relevant only within their administrative domains. The heterogeneity of CMS's context models is also desirable, since it promotes efficiency and security.

Lack of In-Advance Knowledge of CMS

For some interests, the CMSs responsible for managing the context can be statically discovered. For example, both a GPS sensor and an accelerometer are internal device sensors, so applications may be statically prepared to collect and deal with context information they provide. For other providers, however, a consumer does not know previously the CMSs that contain the desired context and, thus, where its interests must be registered. For example, there may be many CMSs, as Table 2.2 exemplifies, each one from a different administrative domain, that provide a particular location information for some users. If the context-aware ecosystem requires that all consumers make an explicit addressing of CMSs where their interest has be registered, then applications should be developed to have a previous knowledge of existing CMSs. In a highly distributed ecosystem, dealing with all CMSs may introduce a heavy burden to applications, which have to deal with an amount of interest registry and the composition of the resulting notification matches.

The interest delivery layer plays an important role to identify the CMSs where a specific consumer interest must be registered.

Dynamic Deployment of CMSs

As a result of the incremental introduction of context-aware computing, new CMSs may be dynamically deployed and start participating in the ecosystem. This fact introduces a more challenging scenario in terms of addressing and delivering interests to CMS, discussed in the previous characteristic.

These five characteristics produce a dynamic and unpredictable behavior on the interest processing layer. A middleware for context-aware computing should deal with such dynamics, keeping it transparent to the upper level interest description layer. In fact, the challenges for the implementation of dynamic context-aware ecosystems, relates both in the interest delivery layer and in the interest description layer. The following sections discuss the resulting middleware requirements for the layer of interest delivery and interest description.

2.4.1 Requirements of the Interest Delivery Layer

A middleware that supports dynamic context-aware ecosystems must satisfy five requirements: (i) support for seamless evolution of context management systems, (ii) dynamic context discovery, (iii) domains of context perception, (iv) uniform representation of context interests, and (v) distributed context management.

The interest delivery layer directly handles changes of the elements of the interest processing layer, as result of the dynamic deployment of CMSs, sensors, and types (context model). Hence, the interest delivery layer is responsible for accommodating such changes for the currently active context interests, without the need to restart or invalidate registered context interests, i.e. supporting seamlessly the evolution of context management systems.

This dynamics within an ecosystem also requires the dynamic discovery of CMSs. For example, in *UMessenger 2.0*, to track the location of a specific buddy, the interest delivery layer must register the application's interest at the corresponding CMSs that maintain location information for the selected buddy. If the buddy moves to another environment and, as a consequence, sensors connected to other CMSs start publishing the location of the buddy, then the application's interest must be registered at those CMSs. Registering an interest at all available CMSs is not an acceptable approach because the middleware will not scale with the number of application's interests. Ideally, the middleware should dynamically discover which CMSs may contains context involved in a context interest. This book uses the term *dynamic context discovery* to express this requirement.

A middleware for context management must support *domains of context perception*, i.e. must allow each CMS to adopt a particular context model. In addition, the middleware must be aware of this model heterogeneity among CMSs, and then register an interest at the CMSs for which it applies. Also, in this case, the middleware would not scale if the whole ecosystem is based on a single context model.

Table 2.3 Mapping between requirement for context management middleware and Kindberg and Fox's principle for ubiquitous computing

Requirement	Related Principles
Support for seamless evolution of context-aware management systems	Volatility
Dynamic context discovery	Volatility & System boundary
Domains of context perception	System Boundary
Uniform Interest Description	Spontaneous Interoperation
Distributed Context Management	Scalability & System boundary

The dynamics of the ecosystem calls for heterogeneous CMSs, specially in terms of models and managed sensors. The middleware must adopt a primitive to describe context interests that can be the interpreted and registered at any CMS, in spite of their heterogeneity. This requirement is called *uniform representation of context interests*.

Finally, since an ecosystem is inherently composed of distributed CMSs, the middleware also must support distributed context management.

These five requirements are aligned with the following three principles of Kindberg and Fox [14] for system software for ubiquitous computing:

- *Volatility*: the set of participating users, hardware, and software is highly dynamic and unpredictable.
- *System boundary*: an ecosystem is divided into environments with boundaries that demarcate their content, creating the notion of environment's scope.
- *Spontaneous interoperation*: software components may spontaneously enter in the ecosystem and start interactions with each other.

In fact, the goal of proposed approach for context management is to promote ubiquity in a dynamic context-aware ecosystem. In addition to Kindberg and Fox's principle, a context-aware ecosystem also demands for *scalability* as an orthogonal principle. Table 2.3 shows the relationship between each discussed requirement for a context management middleware and the corresponding principle proposed by Kindberg and Fox.

2.4.2 Requirements for Interest Description Layer

The characteristics of a distributed and dynamic context-aware ecosystem may impact on the complexity of a context interest. For example, an application may need to describe the specific CMS to which an interest applies. In this case, the CMS assumes the meaning of the interest's scope.

In contrast, if an interest is not explicitly restricted to a particular CMS then the set of CMSs responsible for its processing cannot be solved *in interest description time*. In a context-aware ecosystem, each CMS may have a particular context model,

Table 2.4 Classification of Context Interest Expressions

Aspect	Type	Description
Domain of CMS	Closed domain (D_C)	Expression applies to a specific (and well-known) CMS. **Ex**: *Application adapts its mode of communication according to device resources (local domain).*
	Relative domain (D_R)	Expression only applies to the current CMS to which an application or the contextualized entity is associated. **Ex**: *Obtain the map of the current domain.*
	Open domain (D_O)	Expression must be applied to a broader domain space of CMS. **Ex**: *UMessenger 2.0uses location obtained from any provider to track the user's buddy location.*
Type Coverage	Specific Type (T_S)	Expression applies to a very specific and previously known context type. **Ex**: *UMessenger 2.0uses wireless bandwidth to adapt its communication mechanisms/protocol.*
	Abstract Type (T_A)	Interest expression applies to a general and abstract context type, which may be specialized by different and specific context types, that in turn is provided by different context sources. **Ex**: *UMessenger 2.0requests abstract location to locate a user in a map, which may be either a coordinate-based location or a symbolic location.*

as a result of the particularities of the corresponding environment. Consequently, a type involved in a specific interest may be recognized only in a subset of ecosystem's CMSs. Thus, the CMSs that process an interest may range from a particular one to the whole ecosystem's CMSs. We call *wideness* of an interest such behavior of being more broad or more narrow. The set of CMSs that recognizes a certain context type must be evaluated at runtime, so we say that a context-aware ecosystem enables *context interests of variable wideness*, as defined below.

Context interest of variable wideness	is a context interest in a context-aware ecosystem that either involves an undefined number of CMSs in its matching or undefined actual context types.

Table 2.4 classifies context interest expressions according to two orthogonal aspects: *domains of CMS*, i.e. which set of CMS may be involved in a context interest resolution, and *type coverage*, i.e. how specific is the context type which the application is interested in. Table 2.4 also shows some examples of these interests, using *UMessenger 2.0* as a reference.

The *domain of CMS* specifies which of the CMSs will participate in the processing of a context interest. Since more than one CMS may contain providers for the same type of context information, a context interest may require the registration of lower-level interests at several CMSs. Then the interest must be disseminated to each CMS and the corresponding notifications must be delivered back to the application. If an interest specifies exactly one CMS responsible for its processing, then we say that it is a closed domain (D_C) interest.

However, in a distributed and open scenario, where new CMSs and context providers may be added and removed at anytime, an application may not be able to identify the set of CMSs that provide a specific context. In this case, if a change occurs at runtime, it may cause inconsistencies or disruptions in interest match notifications. When an interest must be applied to an undefined set of CMSs, we call this expression of an open domain (D_O) interest. A relative domain expression (D_R) is a particular case of interest where it must be applied only to the closest scope of CMS to which the application or the contextualized entity is associated.

The other aspect of interest expression is the scope of the context type requested in an interest expression. In this case, we assume that the system supports hierarchical context models with the notion of super and subtyping among context types. An interest expression associated to an abstract type (T_A) may be refined to interests of any subtype, increasing the number of notifications and the complexity of result interpretation. An expression for a specific type (T_S), defines precisely the actual type that must be involved in an interest match. Distributed and open CMSs increase the complexity of implementing abstract type expression, since they allow each domain to have its own context model.

The goal of a middleware for such an open and evolutionary scenario is to enable those interest expressions without increasing the application's complexity. Middleware systems should make transparent the diversity and distribution of CMS, in terms of context models and available context sources. Furthermore, the middleware's programming abstractions should allow applications to specify in just one context interest expression, D_O interests, leaving for the middleware to solve the inconsistencies among interest match notifications, according to the application requirements.

Chapter 5 presents a case study of an implementation of the *UMessenger 2.0*, discussing in more detail the different types of interest expressions that can be used by the application.

2.5 Summary

This chapter has shown the fundamental concepts for context-aware computing and the main components of an architecture for context-aware computing. This chapter used as a running example a hypothetical application called *UMessenger*. In special, two fundamental concepts were used: context interest and context management system (CMS).

Section 2.3 introduced the term context-aware ecosystem to specify all elements that interact among each other in an architecture of distributed context-aware computing. The management of context interest in an ecosystem is composed of three conceptual layers: interest description, interest delivery, and interest processing layers.

Section 2.4 discussed how the dynamics of a context-aware ecosystem challenges the implementation of management layers of context interests. As a result of this discussion, Sect. 2.4.1 enumerated five requirements for a middleware for distributed context management in respect to interest delivery layer whereas Sect. 2.4.2 argued that in a context-aware ecosystem, applications demand for interest description approach that enable the description of interest with variable domain of CMS and coverage of types. This interest is called *context interest of variable wideness*.

The next chapter presents distributed architectures for context management that integrates and composes distributed CMSs, to build a context-aware ecosystem, and discusses how they support context interests of variable wideness.

References

1. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive Mob. Comput. **6**(2), 161–180 (2010). doi:10.1016/j.pmcj.2009.06.002
2. Bolchini, C., Curino, C.A., Quintarelli, E., Schreiber, F.A., Tanca, L.: A data-oriented survey of context models. SIGMOD Rec. **36**(4), 19–26 (2007). doi:10.1145/1361348.1361353
3. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: context-aware reflective middleware system for mobile applications. IEEE Transact. Softw. Eng. **29**(10), 929–945 (2003). doi:10.1109/TSE.2003.1237173
4. Chen, G., Li, M., Kotz, D.: Design and implementation of a large-scale context fusion network. In: The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, Mobiquitous 2004, pp. 246–255 (2004). doi:10.1109/MOBIQ.2004.1331731
5. Chen, H.: An intelligent broker architecture for pervasive context-aware systems. Ph.D. thesis, University of Maryland, Baltimore County (2004)
6. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum. Comput. Interact. **16**(2–4), 97–166 (2001). doi:10.1207/S15327051HCI16234_02
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003). doi:10.1145/857076.857078
8. Grossmann, M., Bauer, M., Honle, N., Kappeler, U.P., Nicklas, D., Schwarz, T.: Efficiently managing context information for large-scale scenarios. In: Third IEEE International Conference on Pervasive Computing and Communications, PerCom 2005, pp. 331–340 (2005). doi:10.1109/PERCOM.2005.17
9. Henricksen, K., Indulska, J.: Developing context-aware pervasive computing applications: models and approach. Pervasive and Mobile Computing **2**(1), 37–64 (2006). doi:10.1016/j.pmcj.2005.07.003
10. Henricksen, K., Indulska, J., McFadden, T., Balasubramaniam, S.: Middleware for distributed context-aware systems. Lect. Notes Comput. Sci. **3760**, 846–863 (2005)
11. Henricksen, K., Livingstone, S., Indulska, J.: Towards a hybrid approach to context modelling, reasoning and interoperation. In: 1st International Workshop on Advanced Context Modelling, Reasoning and Management, pp. 54–61. Orlando, Florida (2004)
12. Hightower, J., Borriello, G.: Location systems for ubiquitous computing. Computer **34**(8), 57–66 (2001). doi:10.1109/2.940014

13. Hong, J.I., Landay, J.A.: An architecture for privacy-sensitive ubiquitous computing. In: Proceedings of the 2nd international conference on Mobile systems, applications, and services, MobiSys '04, pp. 177–189. ACM Press, New York (2004). doi:10.1145/990064.990087
14. Kindberg, T., Fox, A.: System software for ubiquitous computing. IEEE Pervasive Comput. **1**(1), 70–81 (2002). doi:10.1109/MPRV.2002.993146
15. LaMarca, A., Chawathe, Y., Consolvo, S., Hightower, J., Smith, I., Scott, J., Sohn, T., Howard, J., Hughes, J., Potter, F., Tabert, J., Powledge, P., Borriello, G., Schilit, B.: Place Lab: device positioning using radio beacons in the wild. In: 3rd International Conference on Pervasive Computing. Munich, Germany (2005)
16. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer, New York (2006)
17. Nascimento, F.N.D.C.: A service for location inference of mobile devices based on IEEE 802.11. Master's thesis, Departamento de Informática, PUC-Rio (2006)
18. Orr, R.J., Abowd, G.D.: The smart floor: a mechanism for natural user identification and tracking. In: CHI '00 extended abstracts on Human factors in computing systems, pp. 275–276. ACM, New York, (2000). doi:10.1145/633292.633453
19. Sacramento, V., Endler, M., Rubinsztejn, H.K., Lima, L.S., Goncalves, K., do Nascimento, F.N.: MoCA: a middleware for developing collaborative applications for mobile users. IEEE Distrib. Syst. Online **5**(10) (2004)
20. Strang, T., Linnhoff-Popien, C.: Service interoperability on context level in ubiquitous computing environments. In: Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet. L'Aquila, Italy. (2003)
21. Strang, T., Linnhoff-Popien, C.: A context modeling survey. In: First International Workshop on Advanced Context Modelling, Reasoning And Management. Nottingham, England (2004)
22. Undercoffer, J., Perich, F., Cedilnik, A., Kagal, L., Joshi, A.: A secure infrastructure for service discovery and access in pervasive computing. Mob. Netw. Appl. **8**(2), 113–125 (2003). doi:10.1023/A:1022224912300
23. Wyckoff, P., McLaughry, S., Lehman, T., Ford, D.: TSpaces. IBM Syst. J. **37**(3) (1998)
24. Yau, S.S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S.: Reconfigurable context-sensitive middleware for pervasive computing. IEEE Pervasive Comput. **1**(3), 33–40 (2002)