FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

COPAL

An Adaptive Approach to Context Provisioning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Sanjin Sehic
Matrikelnummer 0426689

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Fei Li

Wien, 18.07.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Sanjin Sehic
Goldschlagstraße 125/34, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

This thesis is dedicated to my parents

Mehmed and Jesenka Sehic

who taught me to never stop learning and to always remember to have fun.

Acknowledgements

First and foremost, I would like to express deepest gratitude to my advisers Schahram Dustdar and Fei Li. Without their unselfish sharing of knowledge, long discussions, patience with answering my questions and never-ending support, implementing COPAL and writing this thesis would not have been as fun as they were. They gave me a chance to prove myself and this meant a huge deal to me.

I am especially thankful to my sister, Suncica Sehic. She was there for me when I need someone the most. She always reminds me who I am and where I come from. Because of her, I will never forget where I should be going. And to her, I can only say "voli te buraz."

Last but not least, I want to thank my friends who were my family away from my family, my colleagues at Distributed Systems Group for good company and interesting conversations, all professors and assistants during my bachelor and master studies at Vienna University of Technology for making me push myself to new limits, and the SM4ALL project for providing us with the financial support to develop COPAL.

Abstract

Context-awareness is one of the cornerstones of mobile and ubiquitous computing. It refers to the idea that an application can understand its *context* to reason about its current situation and perform suitable operations based on this knowledge. Moreover, as the situation changes over time, the application should adapt its behavior according to new circumstances, which would increase its usability and effectiveness.

In context-aware systems, context information is gathered from numerous heterogeneous context sources. These sources are mostly low-level sensors that are unaware of application requirements and its information models. Furthermore, information from sensors is too fine-grained and low-level for applications to consume. Thus, the task of context-aware systems can be summarized as hiding the complexity of gathering context information from context sources, inferring new context information from presence or absence of source information, and providing applications with an easy interface to retrieve the context information and adapt accordingly. These diverse requirements create a great challenge for development of flexible and scalable context-aware systems.

This thesis introduces the COPAL (COntext Provisioning for ALl) middleware — an adaptive approach to context provisioning. The COPAL middleware is a flexible and scalable context-aware service platform that provides a new publish-process-listen programming model. Its loosely-coupled and modular implementation allows the system to be customized for different use-cases and deployed on different platforms. The COPAL programming model separates the task of context-awareness into three independent steps supported by three loosely-coupled components: publishers, processors, and listeners. This component design enables developers to progressively extend the system to support new types of context information and various context-aware applications.

Furthermore, the thesis presents a customizable processing mechanism that dynamically couples context information with its processing. This mechanism is the key concepts in the COPAL middleware by which a wide range of operations can be carried out. Most importantly, it can be used to infer new context information and to provide some context information at different levels of granularity.

An instance of the COPAL middleware is currently deployed in a working prototype smart house at Fondazione Santa Lucia in Rome, Italy, where it provides context-awareness for automated execution of tasks, goals, and processes. The prototype smart home was built as part of the SM4ALL (Smart hoMes for ALl) project — an international scientific research project research project funded by the European Community under Seventh Framework Programme (FP7).

Kurzfassung

Kontextbewusstsein ist einer der Grundpfeiler des mobilen und ubiquitären Computing. Es bezieht sich auf die Idee, dass eine Applikation ihren *Kontext* verstehen kann, um ihre aktuelle Situation zu überlegen und beruhend auf diesem Wissen geeignete Operationen auszuführen. Während sich die Gegebenheiten mit der Zeit ändern, sollte die Applikation ihr Verhalten an die neuen Situationen anpassen, um ihre Verwendbarkeit und Wirksamkeit zu verbessern.

In kontextbewussten Systemen werden Kontextinformationen aus zahlreichen, heterogenen Kontextquellen gesammelt. Diese Quellen sind meistens systemnahe Sensoren auf niedrigem Abstraktionsniveau, welche für sich genommen die Anforderungen der Applikationen und deren Informationsmodelle nicht kennen. Weiters sind die rohen Sensorinformationen zu feinkörnig und systemnah, um direkt von den Applikationen konsumiert zu werden. Um diesen Umstand zu bewerkstelligen, ist es die Aufgabe eines kontextbewussten Systems, die Komplexität des Aufbereitens von Informationen aus Kontextquellen zu verbergen, neue Kontextinformationen aus dem Vorhandensein oder Fehlen von ursprünglichen Informationen abzuleiten, und den Applikationen eine einfache Schnittstelle für Informationsabfrage und entsprechende Adaptierung bereitzustellen. Diese diversen Anforderungen stellen eine große Herausforderung dar, flexible und skalierbare kontextbewusste Systeme zu implementieren.

Diese Arbeit stellt die COPAL (COntext Provisioning for All) Middleware vor – einen Ansatz für adaptive Verarbeitung und Provisionierung von Kontextinformationen. Die COPAL Middleware ist eine flexible und skalierbare kontextbewusste Serviceplattform, die ein neues publish-process-listen Programmiermodell bereitstellt. Die lose gekoppelte und modulare Implementierung ermöglicht es, das System auf verschiedene Anwendungsfälle anzupassen und in verschiedenen Plattformen zu verwenden. Das COPAL Programmiermodell unterteilt die mit Kontextbewusstsein verbundenen Aufgaben in drei unabhängige Bausteine, die mit drei lose gekoppelten Komponenten unterstützt werden, nämlich Publishers, Processors, und Listeners. Dieses Komponentendesign erlaubt Entwicklern, das System stufenweise zu erweitern, um neue Typen von Kontextinformationen und diverse kontextbewusste Applikationen zu unterstützen.

Weiters präsentiert die Arbeit einen anpassbaren Mechanismus, der Kontextinformationen dynamisch mit ihren Verarbeitungselementen koppelt. Dieser Mechanismus ist das Schlüsselkonzept in der COPAL Middleware, mit dessen Hilfe diverse Operationen ausgeführt werden können. Als wichtigster Anwendungsfall kann sie dafür benutzt werden, neue Kontextinformationen abzuleiten und Kontextinformationen auf verschiedenen Granularitätsstufen bereitzustellen.

Eine Instanz der COPAL Middleware ist derzeit in einem funktionierenden Prototyp von Heimautomatisierung (Smart Home) in Fondaziona Santa Lucia (Rom, Italien) im Einsatz, wo

sie Kontextbewusstsein für automatisierte Ausführung der Aufgaben, Ziele, und Prozesse bereitstellt. Der Prototyp des Smart Home wurde im Zuge des SM4ALL (Smart hoMes for All) Projekts entwickelt – ein internationales Forschungsprojekt finanziert von der Europäischen Gemeinschaft unter dem Siebenten Rahmenprogramm (Seventh Framework Programme, FP7).

Publications

Parts of this thesis have appeared in the following publications:

- Fei LI, Sanjin Sehic, Schahram Dustdar, (2010). *COPAL: An Adaptive Approach to Context Provisioning*. The 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2010), 11-13 October 2010. Niagara Falls, Canada.

- Sanjin Sehic, Fei LI, Schahram Dustdar, (2011). *COPAL-ML: A Macro Language for Rapid Development of Context-Aware Applications in Wireless Sensor Networks*. The 2nd International Workshop on Software Engineering for Sensor Network Applications (SESENA 2011) at ICSE, 21-28 May 2011, Honolulu, Hawaii, USA.

Furthermore, the COPAL middleware was additionally used in the following publication as part of the overall solution:

- Katharina Rasch, Fei Li, Sanjin Sehic, Rassul Ayani, Schahram Dustdar (2011). *Context-driven Personalized Service Discovery in Pervasive Environments*. World Wide Web (WWW): Internet and Web Information Systems Journal, Volume 14, Issue 4, July 2011.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

COPAL COntext Provisioning for ALl

API Application Programming Interface

DHT Distributed Hash Table

DOM Document Object Model

EBNF Extended Backus–Naur Form

EPL Event Processing Language

GPS Global Positioning System

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JAR Java ARchive

JMS Java Message Service

JSON JavaScript Object Notation

OSGi Open Services Gateway initiative

P2P Peer-to-peer

QoS Quality of Service

REST Representational State Transfer

RFID Radio Frequency Identification

SGML Standard Generic Markup Language

SM4ALL Smart hoMes for ALL

SOA Service-Oriented Architecture

UML Unified Modeling Language

URI Uniform Resource Identifiers

URL Uniform Resource Locator

XML eXtensible Markup Language

JAX-RS Java API for RESTful Web Services

CHAPTER 1

Introduction

Mark Weiser started his paper [70] from 1991 with these seminal words: "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." He envisioned the 21st century world as place where computing will be *ubiquitous*. The essence of his vision was an environment, he called it the "embodied virtuality", where computers are drawn out of their hardware shells and become part of the physical world around us. His vision was simply too far ahead for his time as there was yet no technology required to support it.

After two decades of software and hardware development, we can see first glimpses of his vision coming true. Netbooks, tablets, smartphones, smart appliances, cloud computing, and pervasive environments just to mention a few. We can easily see the trend: computers are becoming ubiquitous in our daily lives and we are becoming more dependent on them to understand and interact with the world and people. To support this, computers are becoming increasingly more aware of their surrounding and the person that is using them. Advantage of increasing awareness about the environment in devices is that they allow us build applications that are *minimally intrusive* to the user [62]. These applications are aware of user's state and surroundings, and adapt their behavior based on this information. A user's environment and state can be quite rich in information, 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 forth. If we were to give such information to a person, he or she would be able to make decisions in a *proactive* fashion, anticipating the user's needs. Thus, we can expect from applications that use this information to be able to reason about current situation and can be built to be less intrusive and more proactive for the user.

Context-awareness is one of the cornerstones of mobile and ubiquitous computing [8,27,62]. The history of context-aware systems started with introduction of *Active Badge Location System* by Want et al. [68] in 1992. The system is considered to be one of the first context-aware applications. It used the infrared technology to determine location of its users and forward phone calls to a telephone closes to its user. The actual term "context-aware" was coined by Schilit and Theimer in their paper [64] from 1994. Today, context-awareness has been associated with some

other terms: adaptive [15], reactive [24], responsive [30], situated [46], context-sensitive [55], and environment-directed [34].

In general, context-awareness refers to the idea that applications can understand their *context* and adapt their behavior based on information gathered from the environment without an explicit user intervention. Thus, it aims to increase usability and effectiveness of applications. This feature is particularly important in mobile system, where it is desirable that applications and services react specifically to their location, time and other environmental attributes and modify their behavior according to changing circumstances. Most importantly, this dynamic adaptation aspect can provide context-aware applications with a degree of autonomy. Furthermore, it frees users from the current model of human-computer interaction, where they have to explicitly express all information that is relevant for application's task and to confirm all actions executed by the application.

1.1 Motivation

The following motivational scenario is meant to introduce the reader with requirements that are expected from a flexible context-aware system. Moreover, it should be considered as a prelude into challenges associated with designing and implementing a context-aware system that will be explained in next section.

In this scenario, we will consider an imaginary person named Alice that lives in a smart home. The smart home is equipped with different sensors and a context-aware service platform. The first requirement that we will consider in this imaginary scenario is that context-aware services should keep a comfortable ambiance for Alice. For the most basic use-case, these context-aware services should decide suitable lighting and settings for an air-conditioner (AC). That would require them to be able to access information about current luminance and temperature in the house.

Additional helpful reading would be to have presence of Alice in the house. This information could control if keeping comfortable ambiance is currently required at all. For example, it would be very wasteful to keep the lights on when Alice is not even in the house. On the other hand, the context-aware services should be implemented such that they do not depend on the presence information, as some houses do not have a sensor for it. They should instead fail gracefully to keep the comfortable ambiance when user manually demands them to. In our scenario, the smart home does contain a device that can determine exact coordinates of Alice in the house using a RFID (Radio Frequency Identification) tag. This information does not exactly provide the presence information, but it does provide enough information to deduce if Alice is in the house or not based on the presence or absence of the signal emitted from the RFID tag.

Furthermore, Alice is additionally involved in an experimental deployment of smart meters and smart grids [20] that implement a brand new pattern of power service and encourage residential energy saving. Smart meters are deployed in Alice's house to monitor and control appliances like lights in each room, AC, fridge, television, etc. They provide the current power consumption for each appliance and are deployed together with the current price indicator that receives real-time price information from the power market [67]. Using this information and the context-aware service platform, Alice wants to develop a context-aware service that will re-

duce her power consumption. This context-aware service will have three modes of operation depending on her total power consumption and total power cost: aggressive, moderate, and lenient. Based on the current mode of operation, it will control how much freedom the ambiance services have. For example, the tolerance level before AC is turned on would be higher in the aggressive mode than in the moderate mode, thus keeping AC turned off for longer periods of time. This service also shows the discrepancy between data that is required, total power consumption and cost, and data that is provided, power consumption per appliance and current price of power. Therefore, the context-aware service platform must be extensible enough to provide mechanism to compute this information.

This motivational scenario was meant to illustrate the requirements for a flexible context-aware service platform. First, it enumerates the major actors that play role in communicating with the platform: appliances, sensors, and context-aware services; and that all of them communicate differently with the platform and expect different requirements from it. Second, it shows that context-aware services must exploit multiple types of context information that are provided by many sources: luminance sensor, thermometer, RFID sensor, smart meters, and price indicator. Finally, it demonstrates that some additional implicit information will have to be deduced in the platform and to be provided to the services: user presence in the house, total power consumption, and total price of consumed power.

1.2 Challenges

A key challenge in context-aware systems is how to obtain information needed for an application to function in a proactive and non-intrusive manner [62]. In a context-aware system, context has to be gathered through automated means. This removes burden for users to explicitly provide all necessary information. Thus, context has to be acquired from many heterogeneous information sources with each one having its own way of accessing and storing data. Some sources will require continuous pooling of data for changes, and others will notify when information changes. Some will require the system to access data over a network, and others will store it locally. Moreover, data format can be different from one device to another. This creates an additional challenge, because if we were to expect from context-aware applications to understand all these formats, the complexity of developing the applications would increase exponentially with each addition of new context source. Hence, accessing, storing, and understanding data from different context sources has to be solved in the context-aware system, using some dedicated *context model* [13, 66], for it to be usable by different types of context-aware applications and to be able to scale linearly with each new type of context information.

Other major challenge in supporting context-aware applications is how to bridge the vast information gap between context sources and context-aware applications [49]. Context information is generated by heterogeneous devices, which are unaware of the application requirements and information models. This information is mostly too fine-grained and low-level for context-aware applications to consume. For example, a GPS (Global Positioning System) can provide you with a three-dimensional position using longitude, latitude, and altitude. This information might be is important for some types of critical systems like airplane navigation, but many user applications do not require such fine-grained information. They might only require current

street name or name of the city for it to function correctly. This discrepancy between granularity level of information that is provided by the context sources and level that is required by context-aware applications can create a challenge for a context-aware system, as well as its programming model, as it will need to provide support to bridge this gap.

The last challenge in a context-aware system is that it needs to allow context-aware applications to adapt their behavior. In the core of this challenge lies situation-reasoning. Context-aware applications have to be able to define situations in the environment that are of interest to them. For example, the aforementioned service that keeps comfortable temperature in Alice's home will define temperature range when it will turn on the AC and temperature ranges for which the AC will be off. Change of temperature from one temperature range to another is a situation in the environment, which is of interest to the service, as it will need to either turn on or turn off the AC. Thus, a context-aware system should provide a flexible mechanism, so a context-aware application can specify under which circumstances it should be notified.

Finally, we can now define the objective of this thesis:

> *To develop a context-aware service platform which will provide, by gathering and abstracting, sufficient information about the environment for context-aware applications to be able to reason about the current situation, make proactive decisions and adapt themselves to the changing context.*

1.3 Approach

Our approach in designing a context-aware system extends the *context provisioning* [48] idea that bases itself on the *publish/subscribe model* [12]. The three aforementioned challenges are solved as three independent steps in the system, namely *publishing*, *processing*, and *listening*. The publishing step uses set of publishers to acquire information about the environment from context sources. Each publisher is associated with one context source and specifically developed for that source to hide the complexity of accessing and communicating with it. Furthermore, they enforce a common representation model for context and publish their observations as events that represent a single change in context. The processing step allows construction of complex context provisioning schemes by invoking multiple processors before events reach context-aware services. The idea of processing context is borrowed from the model for event-processing networks [65] and is adapted in our solution to work as part of context provisioning. Processors define abstract relationships between input and output events. They can be used to modify events, filter malformed or low-quality information, translate events from one representation to another, and aggregate/differentiate events into one or more events. Most importantly, processors can provide same information in different levels of granularity and to infer new information from occurrence and/or absence of events. Lastly, the listening phase provides a complex selection mechanism to continuously query published and processed events. Continuous queries provide the mechanism to specify under which circumstances a context-aware service will be notified. Hence, services can react to changes in context and adapt their behavior accordingly.

1.4 Contribution

The COPAL[1] (COntext Provisioning for ALl) middleware is the implementation of the aforementioned approach for building a context-aware system. It is situated between communication layer that allows access to context sources and application layer that is interested in context information. Its main advantage is that it hides complexity of accessing context sources from context-aware applications and provides developers with simple API to create and deploy context-aware applications. Merits of the COPAL middleware are as follows:

- It is a flexible context-aware service platform that provides a new programming model for development of context-aware systems using publisher-processor-listener abstraction. The programming model allows to progressively extend the system to support new types of environmental information and to add new context-aware services.
- It provides a scalable and distributed architecture that can handle many heterogeneous context sources and can be used for varying context-aware services. Instances of the COPAL middleware that are part of same global context can be distributed over multiple devices. Communication and synchronization of context between the instances is transparent to all context-aware services.
- It is separated into multiple modules to allow easy extension with new modules or even to be reimplemented using new ideas. The loosely-coupled and modular architecture allows the system be customized for different use-cases and deployed on different platforms.
- It provides a flexible solution to build adaptive and customizable processes using common processing patterns. Processing of context is one of the key concepts in this thesis by which a wide range of operations can be carried out. This thesis proposes five processing patterns, namely filter, aggregation, differentiation, enrichment, and peeling, which can be used to design and compose complex processing schemes.

Finally, an instance of the COPAL middleware is currently deployed in a working prototype smart house at Fondazione Santa Lucia[2] in Rome, Italy, where it provides context-awareness for automated execution of tasks, goals, and processes. The prototype smart home was built as part of the SM4ALL[3] (Smart hoMes for ALl) project, which investigates middleware platform for inter-working of smart embedded services in environments like private houses and home-care assistance. The SM4ALL project is an international scientific research project founded by the European Community under Seventh Framework Programme (FP7).

1.5 Organization

The remaining chapters of this thesis are organized as follows: An in-depth look at context and context-awareness is presented in Chapter 2. Chapter 3 provides a short survey of other

[1] http://www.infosys.tuwien.ac.at/m2projects/sm4all/copal/
[2] http://www.hsantalucia.it/
[3] http://www.sm4all-project.eu/

implementations of context-aware systems and examines their characteristics compared to the COPAL middleware. The in-depth design of the COPAL middleware is explained in Chapter 4 together with description of processing patterns and distribution of COPAL instances. Chapter 5 describes the technologies behind the implementation of the COPAL middleware and how they were used to support publishing, processing, querying, and distribution of COPAL instances. Chapter 6 evaluates the performance of the COPAL middleware and demonstrates the usage of the COPAL middleware in the SM4ALL platform. Finally, Chapter 7 concludes the thesis with a short summary and plans to improve the COPAL middleware in future efforts.

CHAPTER 2

Context-Awareness

Context and context-aware systems have been investigated for a decade. Many definitions and concepts about context and context-awareness have emerged in this period, which wanted to clarify the usefulness of these basic ideas in computing. In Section 2.1, we will first review numerous attempts to define, categorize, and model the context. Then, Section 2.2 will examine, based on by then increased understanding of what context is, several efforts to define the "context-aware" term more formally. Finally, we will summarize the history of investigating the concepts of context and context-awareness in Section 2.3

2.1 Context

Definitions

At first, we can try to understand context by looking at how it used in written and spoken text. The meaning of context in natural language can be seen as an implied knowledge that allows us to fully comprehend the point of some statement. For example, if someone says, "he will be joining us there," the pronoun "he" and the adverb "there" are implied knowledge and can be inferred from previous spoken context. Listener has to share this knowledge with the speaker to be able to fully understand whom the speaker is referring to and which place he is talking about.

Context and context-awareness in computing emerged as part of the ongoing research in ubiquitous/pervasive environments. In a ubiquitous system, computing capability is an intrinsic part of any object in the human environment. Thus, a ubiquitous system can be seen as a system of "machines that fit the human environment instead of forcing humans to enter theirs." [70] These objects provide us with rich information about the current environment. Furthermore, applications can use this information to improve their understanding of the environment, modify their behavior based on current situation in the environment, and execute suitable operations depending on the situation. Thus, context information becomes an inherent part of application's state.

First definitions of context were based on enumerating examples of information that should be part of context. Schilit and Theimer [64] defined context as location, identities of nearby people and objects, and changes to those objects. Brown et al. [18] defined it as location, identities of the people around the user, the time of day, season, temperature, etc. Ryan et al. [59] proposed definition of context as the user's location, environment, identity and time. Finally, Dey [26] enumerated context as the user's emotional state, focus of attention, location and orientation, date and time, objects, and people in the user's environment. These definitions were given as part of specialized context-aware applications like context-aware tour guide, which uses user's location to provide relevant information about sites around him. Problem with them became apparent when general context-aware systems emerged. When we want to determine whether a type of information that is not listed in the definitions is in context or not, these definitions are not helpful as it is not clear how we can use any of them to solve the dilemma. Thus, they result in a limited scope of understanding context.

Better way to define context is with usage of synonyms. Brown [16] defined context to be the elements of the user's environment that the user's computer knows about. Franklin & Flaschbart [38] saw it as the situation of the user. Ward et al. [69] viewed context as the state of the application's surroundings and Rodden et al. [56] defined it to be the application's setting. Hull et al. [46] included the entire environment by defining context to be aspects of the current situation. Finally, Dey and Abowd [27] proposed a definition for context that is currently most widely used:

> *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.*

The advantage of this definition is that it allows developers of context-aware service to define which context information is relevant for a given scenario. If a piece of information can be used to describe a situation of a participant in an interaction with the service, then that information is part of the context relevant for the service. For example, let us consider a service that controls lighting in a house. If user executes the command "turn on lights", the actual "lights" have to be implied from the user's location in the house. Thus, location information is in the context for this service because it can be used to characterize the situation of an entity.

Categories

Prekop and Burnett [54] and Gustavsen [40] separated context based on the way it was acquired into *external* and *internal*. External context consisted of measurements made by hardware sensors like location, temperature, lighting, and so forth. Whereas, internal context consisted of information that was inferred from user's interactions with the context-aware system like goals, tasks, emotional state, and so forth. Most commonly, context-aware systems use only external context as it provides enough information about the environment and can be easily acquired from sensing technology that is already available at the market. Notable exceptions from this

rule were made by Watson project [19] and the IntelliZap project [37] that used logical context gathered from information read out of open web pages, documents, etc.

Hofer et al. [44] differentiated context based on its origin into *local* and *remote*. In a distributed network of context-aware devices, the device's own sensors generate local context. Whereas, remote context is generated by nearby devices and can be shared with other devices. The remote context can be used together with local context to approximate values like current temperature at some location. Furthermore, the remote context can be used to provide additional information about the environment that local context does not poses.

Dey and Abowd [28] partitioned context based on the type of information into *primary* and *secondary*. The primary context consisted of location, identity, time and activity. This information was chosen as it can fully answer "who's, where's, when's, and what's of an entity and could be used to determine why a situation is occurring." Furthermore, this information can also be used as indices into secondary context of same entity as well as primary context for other related entities. For example, using person's identity, we can find his phone number, email address, birthday, and so forth, which correspond to his secondary context. Additionally, using the same identity we can find identities of his parents, relatives, coworkers, and so forth, which correspond to primary context of other related entities.

Models

Context models define and store context data in a machine readable and processable format. A well-designed model is a key to the success of any context-aware system. Strang and Linnhoff-Popien in their survey [66] summarized the most relevant context modeling approaches. In this section, context models presented in the survey will be enumerated and described briefly.

Key-value models are the most simple model representations. They use key-value pairs to provide the value for some specific type of context information [63]. For example, user's location can be represented as the pair $(\text{location}, \text{kitchen})$. The key-value model provides an easy way to manage context information, but lacks any sophisticated means to structure the information in a more efficient manner for context retrieval algorithms. This context model is frequently used in distributed service frameworks, where services are most commonly described with a list of attributes in a key-value representation. Then, service discovery operates by exact matching of these attributes.

Markup scheme models use a hierarchical data structure consisting of markup tags with attributes and content. In particular, the context of the markup tags is usually defined by other markup tags. Typical representatives of this kind of context modeling approach are profiles, which are usually based on a serialization of a derivative of Standard Generic Markup Language (SGML) [1]. Typical examples of such profiles are Composite Capabilities/Preference Profile (CC/PP) [47], User Agent Profile (UAProf) [3], and Pervasive Profile Description Language (PPDL) [23].

Graphical models use graphical components to model the context. This type of models are particularly applicable for generating ER-models (Entity-Relationship models) [22] that can be used as structuring tools for a relational database in information systems. Typically, this context model extend any of graphical modeling languages, such as Unified Modeling Language (UML) [4] or Object-Role Modeling (ORM) [43].

Object-oriented models use various object to represent different types of context information. Access to contextual information is only allowed through a well-defined interface. Details of context processing is encapsulated on an object level and hence hidden to other components. Common to all object-oriented context modeling approaches is the intention to employ the main benefits the object-oriented technique, namely encapsulation and reusability, to solve some parts of the problems arising from the dynamics of context in a context-aware system.

Logic-based models use facts, expressions, and rules. A logic-based system allows conditions to be defined on which a concluding expression or fact is derived from a set of other expressions or facts. This process is known as reasoning or inference. In a logic-based model, context information is most commonly inserted, updated, or removed in terms of facts. Furthermore, additional context information is inferred by applying expressions and rules from the model. Common to all logic-based models is a high degree of formality.

Ontology-based models provide a standardized vocabulary to define entities in a context-aware system, their properties and relations between them. Formally, ontology is a standardized representation of knowledge as set of concepts within a domain and relationships between those concepts. It can be used to effectively reason about entities within that domain. Due to their high level of expressiveness, they are becoming a very promising instrument for modeling context information.

2.2 Context-Aware

The first category of definitions for "context-aware" term defines it as an ability to "adapt to context." Schilit and Theimer [64] define context-aware system as a system that "adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time." [64] Ryan [58] defined context-aware applications as applications that monitor input from environmental sensors and allow users to select from a range of physical and logical contexts according to their current interests or activities. This definition is more restrictive than the previous one, because it additionally specified the method, the user selection, in which applications act upon context. Brown [17] saw context-aware applications as applications that automatically provide information and/or take actions according to the user's present context as detected by sensors. He also took a narrow view of context-awareness by specifying that these actions can take the form of presenting information to the user, running a program according to context, or configuring a screen of the user's mobile device. All these definitions require that an application modifies its behavior for it to be considered context-aware.

The second category of definitions tries to be more general and not to exclude existing context-aware applications. Thus, they define "context-aware" simply as an ability to "use context." Hull et al. [46] and Pascoe [52] defined context-awareness to be the ability of computing devices to detect and sense, interpret and respond to aspects of a user's local environment and the computing devices themselves. Dey [26] limited context-awareness to the human-computer interface, as opposed to the underlying application. Afterwards, he began to introduce the notion of adaptation by defining context-awareness to be the work leading to the automation of a software system based on knowledge of the user's context [29]. Salber et al. [60] defined

context-aware to be the ability to provide maximum flexibility of a computational service based on real-time sensing of context.

Finally, Dey and Abowd [27] provided their own definition for context-awareness that is most liberal:

> *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.*

This definition only requires a context-aware system to respond to context, unlike Hull et al. [46] and Pascoe [52] definition that requires a context-aware system to detect, interpret and respond to context. This allows the detection and interpretation of context to be performed in some other computing entity like e.g. context-aware middleware. Moreover, it differs from the other "use context" definitions given above by not limiting awareness to just the human-computer interface as in [26], not requiring applications to perform services automatically as in [29], and not requiring real-time acquisition of context as in [60].

2.3 Summary

The review of definitions demonstrated the gradual development of the understanding of context, and the evolution of the field. First context-aware systems were specialized applications for certain use cases. Examples of these were context-aware tour guides [5, 16] and office awareness systems [53, 68]. Accordingly, definitions of context and context-awareness were influenced by their respective use-case. Context was defined by examples of which information the particular system required and context-awareness was defined by how the system adapted to context information. Afterwards, came general context-aware toolkits and frameworks [28, 61] that provided support for implementing context-aware applications. These frameworks were built on the concept that a context-aware system should enable applications to retrieve the context they require without them having to worry about how the context was acquired. Thus, definitions for context and context-awareness evolved to be more general. Instead of using examples and adaptation techniques, they defined context using synonyms and context-awareness simply as usage of context. Dey and Abowd's definition of context [27] is of great importance here because it provided an abstraction for context. They defined the context as "information used to characterize the situation of an entity." This enabled developers of context-aware systems to stop thinking about use-cases that they should support in their system, and focus on context-awareness in terms of entities that interact with the system and programming abstractions for the system to represent these entities. Hence, many context-aware systems [21, 33, 48, 57] emerged to easily build various context-aware applications that support many different use-cases.

CHAPTER 3

Related Work

Baldauf et al. describe in their survey [8] many different approaches to design a context-aware system. In this chapter, five distinct implementations will be mentioned: Context Toolkit (Section 3.1), CASS (Section 3.2), Gaia (Section 3.3), Solar (Section 3.4), and C-CAST (Section 3.5). These implementations are chosen and explained because each one represents a distinct approach to implement a context-aware system. Finally, comparison and summary of solutions, including the COPAL middleware, will be presented in Section 3.6.

3.1 Context Toolkit

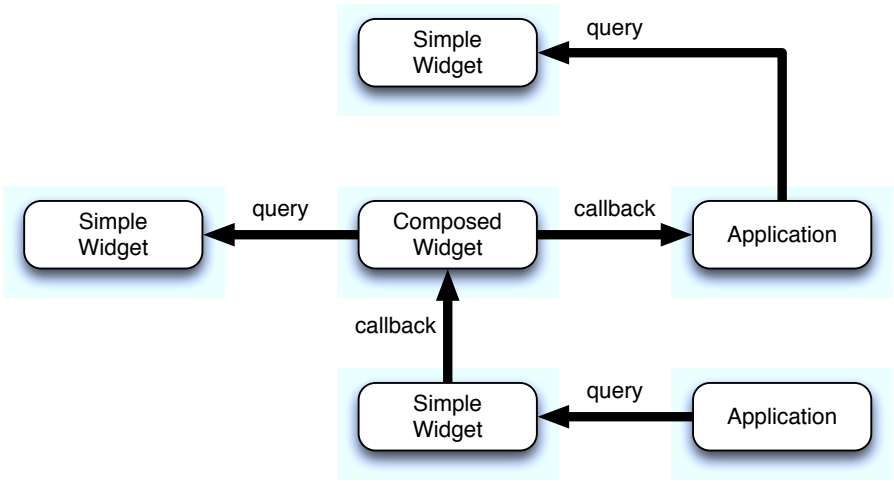


Figure 3.1: The Context Toolkit

The Context Toolkit [28, 61] (Figure 3.1) is one of the first developed context-aware frameworks. The central idea of the system is borrowed from GUI (Graphical User Interface) toolkits

and widget libraries that create reusable building blocks and hide specifics of physical system. *Context widgets* equivalently try to hide the complexity of actual sensors and provide reusable and customizable building blocks for context-aware applications. Thus, a context widget is a software component that provides applications with access to context information from their operating environment. In the same way GUI widgets insulate applications from some presentation concerns, context widgets insulate applications from context acquisition concerns. Hence, context widgets provide developers with following benefits:

- They hide the complexity of the actual sensors used from the application. For example, whether the presence of people is sensed using an active badge, floor sensors, video image processing or a combination of these should not impact applications that require this information.
- They abstract context information to suit the expected needs of applications. For example, a context widget that tracks the location of a user within a building notifies applications only when the user moves from one room to another and doesn't report less significant moves to the applications.
- They provide reusable and customizable building blocks of context sensing. For example, a context widget that tracks the location of a user can be used by a variety of applications, from tour guides to office awareness systems
- They can be tailored and combined to create more complex widgets. For example, a presence widget senses the presence of people in a room. A meeting widget may rely on the presence widget and assume a meeting is beginning when two or more people are present in same room.

Applications can access context information from context widgets using two methods. First, a widget provides a set of attributes that can be queried by applications. For example, an identity presence widget has attributes for its location, the last time a presence was detected, and the identity of the last user detected. Second, applications can register to be notified of context changes detected by the widget. The widget triggers callbacks to the application when changes in the environment are detected. The identity presence widget for instance, provides callbacks to notify the application when a new person arrives, or when a person leaves.

The problem with the context toolkit is that applications have to know beforehand which context widgets they require and where they are located. This creates a problem when we want to use a new sensor. Although, the context widget for this sensor will hide the complexity of accessing the sensor, the application still has to be modified to use the new context widget. Furthermore, the new context widget might provide context information in different format, which will require further modifications to the application.

3.2 CASS

CASS (Context-Awareness Sub-Structure) [33] (Figure 3.2) is *server-based middleware* intended to support context-aware applications on handheld and other small mobile devices. It

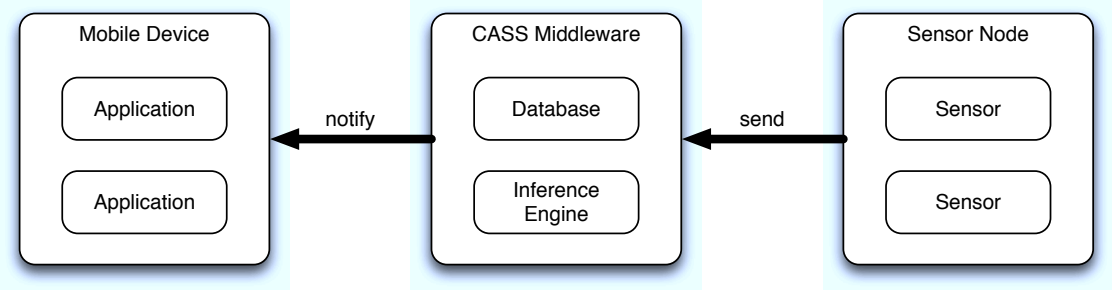


Figure 3.2: CASS

enables developers to overcome the memory and processor constraints of small mobile devices while supporting a large number of low-level sensor and other context sources. A key feature of CASS is its support for high-level context data abstraction and the separation of context based inferences and behaviors from application code. This separation opens the way to making context-aware applications that are configurable by users.

The system is separated into three components: *sensor nodes*, a *server* hosting the CASS middleware, and *context-aware applications*. Sensor nodes are computers that communicate directly with one or more sensors that are attached locally to it. Sensor node can be mobile or static. All sensory data from sensor nodes is transmitted to the CASS middleware that is running on the server. Context-aware applications run on mobile devices that are connected to the server over wireless networks. They do not need to communicate with each individual source of context directly but only with the CASS middleware and therefore do not need to store low-level details of context sources. The CASS middleware provides them with a mechanism to listen for notification of context change events. *Database* storage and *inference engine* are part of the CASS middleware, which removes burden from sensor nodes and mobile devices to process and store data locally. The database offers a persistent data storage for context-awareness. Besides storing context information, the database can be used to store domain knowledge represented as rules and behavior relevant to specific context-aware application. This allows the context reasoning and behaviors of context-aware applications to be changed in a dynamic way. The inference engine works in conjunction with the database and uses rules and goals that are stored in the database to solve problems. It provides the CASS middleware with the ability to infer high-level context information using rules to create an abstraction of low-level context information. Then, it uses this high-level information to find matching goals when a change in context is detected. Goals allow the CASS middleware to change or activate some specific context-aware behavior of context-aware applications.

The biggest drawback of this architecture is its centralized server. Although, performance, memory, and battery limitations create a strong point for having processing and storing context information remotely, this does not constitute a need to have it centralized on one server. If we start to increase number of sensors and applications, shear amount of data stored in the system that needs to be processed will create a bottleneck and the system would not be able to scale.

3.3 Gaia

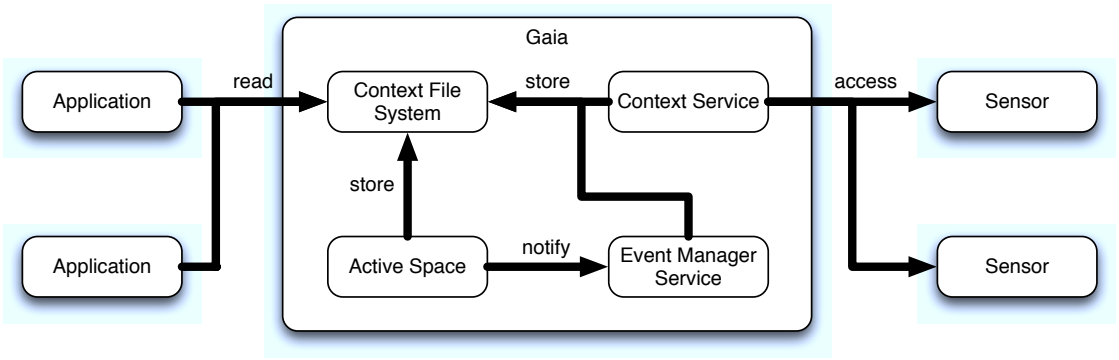


Figure 3.3: Gaia

Gaia [57] (Figure 3.3) is a *meta-operating system* built as a distributed middleware infrastructure that coordinates software entities and heterogeneous networked devices contained in a physical space. It is designed to support the development and execution of portable applications for *active spaces*. Active spaces are programmable ubiquitous computing environments in which users interact with several devices and services simultaneously. Gaia extends the concepts of traditional operating systems to ubiquitous computing spaces, which enables it to simplify space management and application development. It exports services to query, access, and use existing resources and context, and provides a framework to develop user-centric, resource-aware, multi-device, context-sensitive, and mobile applications.

A physical space is some geographically defined region with well-defined and physical boundaries. It contains physical objects, numerous network-connected devices, and users that perform activities. An active space is a physical space that is additionally controlled and coordinated by a context-based software infrastructure that allows mobile users to interact and configure physical and digital seamlessly. Thus, active spaces support development and execution of user-centric mobile applications. In active spaces, *sessions* store user data and associate applications with users. This allows users to move around an active space and have their data and applications always available.

Because Gaia is a general solution for providing fully ubiquitous environment for its users, we will only focus on its components that support the context-awareness part of the solution, namely its *event manager*, *context service*, and *context file manager*. The event manager service allows active spaces to expose and propagate events that signify changes in their current state to other interested parties in the space. It is implemented as a decoupled communication model based on producers, consumers, and channels. A channel forwards events from producers to consumers. A default set of event channels in Gaia notify their consumers about new services, applications, people, errors, and component heartbeats. The context service lets applications query and register for some specific context information. It consists of context providers that offer sensory information about the current environment and of components that can infer cer-

tain high-level information from sensory data. Its context model is based on first-order logic and Boolean algebra. Context information is described using predicates. Logic-based model allows the context service to create rules for inferring higher-level context information using quantifications, implications, conjunctions, disjunctions and negations. The context file system is used to make user's data automatically available to applications, organizing the data to facilitate locating relevant material, and retrieve the data in a format based on user preferences or device characteristics through dynamic data types. It constructs a virtual directory hierarchy based on the types of context associated with particular files. It combines the environment's current context information with user-specific information to create correct data for an application.

The only drawback of this solution is the context model. Logic-based context model does provide a very nice way to create rules for inferring new context information that are based on logical expressions, but this model is not comprehensive enough to define rules that are based on even simplest algebraic calculations. For example, we cannot use this model of inference to define a rule that calculates power consumption in a house by summing the power consumption of each appliance in the house.

3.4 Solar

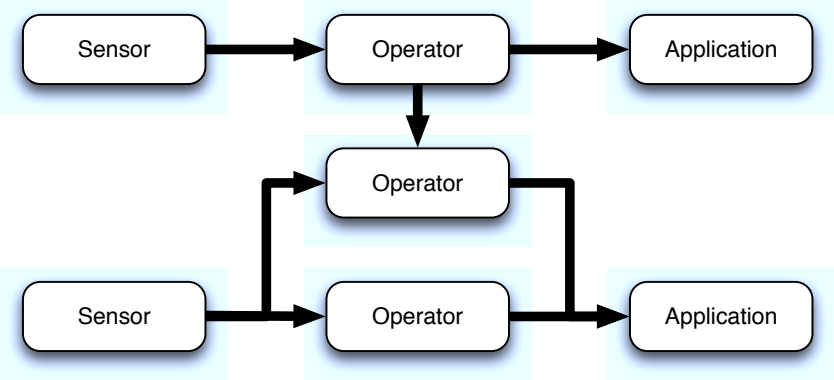


Figure 3.4: Solar

Solar [21] (Figure 3.4) is a *peer-to-peer* (P2P) *network* based on the application layer multicasting and Distributed Hash Table (DHT) [7]. It distinguishes that applications typically need high-level context rather than raw sensor data and that high-level context can be derived by aggregating data from one or more sensors. Solar aims to make it possible to offload this computation from the end-user application device into the middleware, running on one or more servers that host the Solar software. From the application's viewpoint, Solar encourages a modular structure and reduces programming time through code-based reuse. From the system's viewpoint, Solar minimizes redundant computation and network traffic and increases scalability through instance-based reuse.

In the system, applications compose data flows, rather than interacting directly with mobile and embedded sensors and devices. They instruct Solar on which sensor to use and how the

sensor data should be aggregated into desired context. Solar uses the *filter-and-pipe* software architectural pattern for data-stream oriented processing, which supports reuse and composition naturally. In a filter-and-pipe style, each component (filter) has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. A connector (pipe) serves as conduits for the streams, transmitting outputs of one filter to inputs of another. A data flow starts from a source, through a sequence of pipes and filters, and reaches a sink. In Solar terminology, a filter is called an *operator* and a pipe a *channel*. A channel is directional and has two ends; at the input end is attached a source, and at the output end is attached a sink. A *sensor* is a source and an application is a sink; an operator is both a source and a sink. An operator is a self-contained data-processing component, which takes one or more data sources as input and acts as another data source. A channel connects an upstream operator to a downstream operator, and the direction of a channel indicates the direction of data flow. This simple model allows developers to easily connect the sensors, operators, and applications with channels to form an acyclic graph.

Solar takes a fully-distributed approach and consists of a set of functionally equivalent hosts named *planets*. Planets provide some key services: (1) operator hosting and execution, (2) sensor/operator registration and discovery, (3) and data dissemination through operator graphs. The more Planets are deployed, the more capacity Solar has to serve sensors and applications. Since planets are functionally equivalent, Solar interconnects them using an application-level P2P protocol. The advantage of using a P2P-based service overlay is its capability of automatic handling as planets join and depart. Furthermore, Solar uses a DHT where each planet is assigned a unique numeric identification. The DHT interface allows components to send a message to a numeric key and the message will be delivered to a planet with the numerically closest identification.

The biggest drawback in Solar architecture is its processing design using a filter-and-pipe architecture. It creates a very rigid software architecture as in does not provide ways to employ processing on event level. We can easily see that either all data in a channel is processed or none is. Furthermore, the decision on how to compose sensors and operators lays in applications. This increases complexity of applications because different sensors that provide same type of context information might need different operators to create same high-level context. For example, retrieving information about a presence of a user in a room will require different combination of operations based on if we use raw data from active badges or from a security camera feed. From this example, we can see that applications will have to know which sensor is available in the system and create an operator graph accordingly.

3.5 C-CAST

C-CAST [48] (Figure 3.5) first proposed a *broker-based context-provisioning system* that is supported by a *publish/subscribe model* [12]. It was designed to support mobile context based services over any network. Thus, all communication in the system is based by exchange of HTTP (Hypertext Transfer Protocol) [11, 36] messages using a REST (Representational State Transfer) [35] interface, which allows components to be ported onto almost any networked device. Furthermore, the context publish-subscribe model, together with the context broker model,

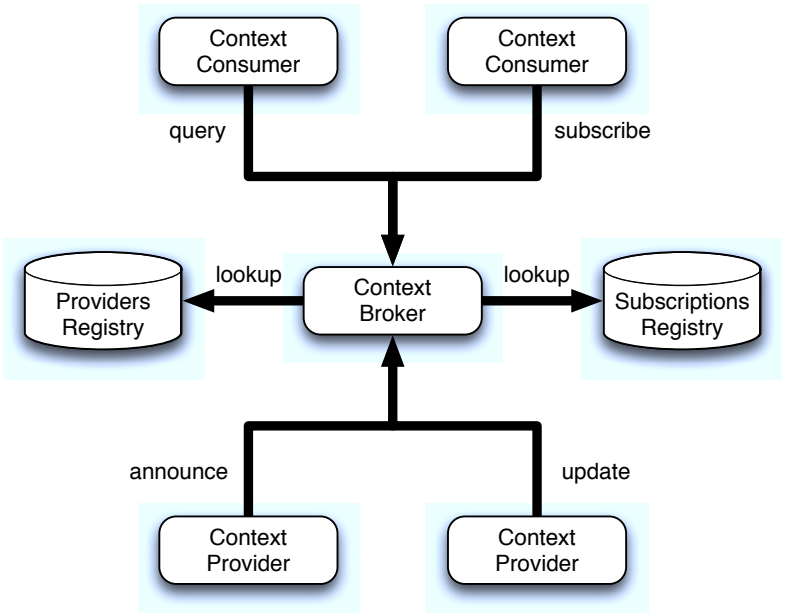


Figure 3.5: C-CAST

allows the framework to scale in terms of physical distribution of components and in terms of context reasoning complexity.

In the system, components take the role of either *context provider* or *context consumer*. The *context broker* holds registrations of all context providers and offers a directory and lookup service so context consumers can find and access the context providers. Registration and lookup of context providers is based on subjects of interests, i.e. types related to the contextual information. Context providers are responsible for accessing and gathering context information from any kind of sensor. A context provider will register with the context broker using an advertisement message in which the context provider announces its capabilities and type of context information that it can provide. The advertisement message additionally contains information on how to access the context provider, its query model, and the way context can be accessed. Context consumers may be any kind of application or actuator utilizing the context information. They ask the context broker for a list of context providers that can provide a specific context type, and, if available, can directly query a context provider for its context. This is a *synchronous* method of communication. Another method of context query by context consumer is available which is *asynchronous* in nature. First, a context consumer subscribes for context information with the context broker. Next, the context broker forwards subscriptions from the context consumer to all registered context providers. When a context provider can match the subscription with produced context information, it will inform the context broker of the available match and from then on will update the context broker on context changes. Finally, the context broker will forward any context changes from the context provider to the subscribed context consumer.

The major limitation of this architecture is the centralized context broker, which creates a single point of failure in the system. Although, the context producers and consumers can be

distributed on many machines, the system is inherently centralized, as there is one single context broker, which mediates almost all communication between context consumers and producers. This can create a huge bottleneck in the system as number of context consumers and producers increases and will not allow the system to scale. Furthermore, the system does not provide any simple way to process context information before it reaches context consumers. Filtering low-quality data, composing multiple types of context to create a high-level information about the environment, and transforming one type of context into another are all responsibility of context consumers and the system does not envision a way to share this information between them.

3.6 Summary

Table 3.1 compares described approaches to designing context-aware systems together with the COPAL middleware. By examining the requirements to develop context-aware systems, the following six criteria for the evaluation have been chosen:

- **C1**: Decoupling of acquisition
- **C2**: Processing mechanism
- **C3**: Complex notification mechanism
- **C4**: Decentralized
- **C5**: Extensible
- **C6**: Programming model

First three criteria (C1–C3) are based on the context-aware challenges that were described in Section 1.2. They are directly related to task of supporting context-awareness in applications. First challenge of context-aware system is to hide acquisition of context from rest of the system (C1). This enables a context-aware system to more easily support many heterogeneous context sources and allows the system to standardize the context model and the interface for accessing the context information. Second challenge is to provide a mechanism to process context information before it is consumed by context-aware applications (C2). Processing of context can be used to infer new context information from presence or absence of other information and to transform one type of information into another type that is more suitable for consumption. Last challenge is to allow context-aware applications to specify under which circumstances they should be notified. Context-aware applications are required to use context information to change their behavior. Most of the time, they only do this if only some specific situation in context occurs. Complex notification mechanism (C3) supports applications by allowing them to more easily define these situations.

Last three (C4–C6) are additional criteria that respectively measure system's support for scalability, flexibility, and ease-of-development. They do not inherently help with the task of context-awareness, but they can lead to solutions that are more versatile and can be deployed to support context-awareness in wider range of use-cases. Decentralized context-aware system

(C4) allow a graceful handling of growing number of context sources and applications by increasing the number of machines that the system is deployed on. A context-aware system that can be dynamically and progressively extended (C5) helps developers to adapt the system as new requirement arise and new use-cases are envisioned. Finally, programming models (C6) enable the developers to more easily reason about behavior of context-aware applications and to prove the correctness of their source code.

	C1	C2	C3	C4	C5	C6
The Context Toolkit	yes	yes	no	yes	no	yes
CASS	yes	yes	yes	no	yes	no
Gaia	yes	partially[1]	no	yes	yes	yes
Solar	yes	yes	no	yes	yes	yes
C-CAST	yes	no	no	no	yes	yes
COPAL	yes	yes	yes	yes	yes	yes

Table 3.1: Comparison of Context-Aware Systems

Although acquisition of context varies between the approaches, all of them separate the acquisition mechanism from the rest of the system (C1). Processing of context information (C2) is also very well supported; except for C-CAST which does not have a dedicated processing mechanism. Interestingly, the complex notification mechanism (C3) is only supported in two context-aware systems: CASS and the COPAL middleware. The COPAL middleware puts big emphasis on this criterion, because it can ease the development of context-aware applications. In context-aware systems that do not support the complex notification mechanism, the system will notify context-aware applications on every change in context. Thus, applications have to reason by themselves if a change in context is relevant or not. If the relevancy changes over time as, for example, user preferences change, it can lead to complex ad-hoc solutions that are hard to maintain and error-prone. Only two approaches, CASS and C-CAST used a centralized context-aware system (C4). They did create a context-aware system that is well suited for their particular use-case involving constrained mobile devices, but this lead to architecture that is not particularly useful for other use-cases. The Context Toolkit is only context-aware system that is not easily extensible (C5), because it does not provide any kind of mediator that decouples components that acquire context from context-aware applications, which can be progressively extended to support new context sources without impeding applications that are already using it. Finally, only CASS does not provide any programming model for building the context-aware system (C6). It does not create any high-level abstraction for context-awareness and applications directly use technologies like database and rule engine to communicate with the system.

[1] Gaia can only do logical inference.

CHAPTER 4

Design

The design of the COPAL middleware is presented in this chapter. First, Section 4.1 will describe context model used in the COPAL middleware and explain gathering, reacting, and processing changes in context. Next, processing patterns, which are supported in the COPAL middleware, will be introduced in Section 4.2. Finally, Section 4.3 will explain the distributed hierarchy for organizing instances of the COPAL middleware and distinguish two ways to share environmental information between the instances.

4.1 Context

Model

In Chapter 2, context has been formally defined as "any information that can be used to characterize the situation of an entity." Informally, context can be acquired or inferred from the surrounding and can help to explain current situation that user or an application is in. For example, weather, time, location, or even CPU utilization can all be part of the same context as they describe different aspects of the same surrounding. We can immediately see that there are many distinct types of environmental information and that each has different domain and has to be represented differently. Even one type of information can have different representations depending on what is required by context-aware applications. For example, current time can be represented as exact value in time continuum (e.g. 1st May, 2011 16:00), or it can be represented as time in a day with possible values being morning, midday, afternoon, evening, or night. Thus, we will have to divide context into many distinct context types — like time, location, temperature, etc. — that each specifies different information about the environment. Each type of information in the environment is associated with exactly one **context type**, which describes which properties is part of this environmental information. For example, a context type describing temperature may consist of properties that specify when a measurement was made, where it was made, how many degrees and in which unit the measurement is.

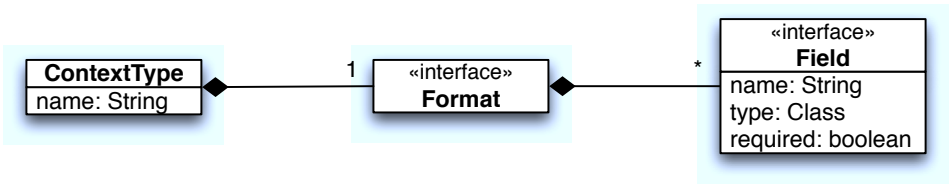


Figure 4.1: Context Type

Figure 4.1 shows a UML class diagram that defines a context type. First, each context type has a unique `name`. Names of context types differentiate types from each other and allow referencing them in the COPAL middleware based on just their name. Second, each context type is associated with a `Format` that describes how and where data is stored. Format of a context type allows the COPAL middleware to understand, for example, where to find the value of temperature. It does this by storing an array of field definitions that describe which fields are allowed for this specific type of environmental information. Each field definition consists of three properties: `name`, `type` and a Boolean value specifying if field is `required`. For example, the temperature context type would have a field with name `degrees` of integral type that is required.

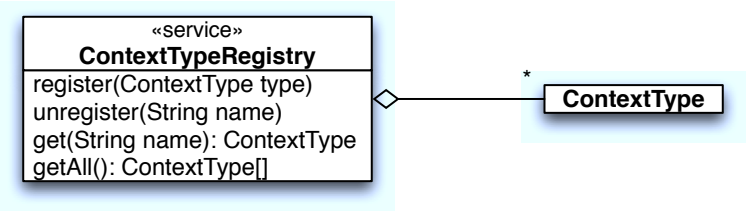


Figure 4.2: Context Type Registry

Furthermore, the COPAL middleware requires all context types to be registered with a **context type registry** service, which provides a centralized place for other components to retrieve context types (Figure 4.2). First, the `register` and `unregister` methods allow to respectively add and remove context types from the system. Second, the context type registry provides a way to retrieve a context type using its name with the `get` method. Finally, the `getAll` method returns all currently registered context types in the system.

In the process of developing context-aware applications, services that applications provide are dominantly interested in just a small subset of environmental information and want to be notified when this information changes. The most common case is that the application will react to some change in the environment and provide a different service to the user or execute some specific task. For example, let us consider a context-aware application that shows a list of favorite people that user can call. In normal situation, this list is static and defined by the user, but in an emergency situation this list is replaced with a new list to call hospital, police, or fire department. Therefore, we can summarize that the task of a context-aware service platform is to provide applications with a way to specify which changes in the environment the application is

interested in and to allow the application to react when one of these changes happens. Therefore, the COPAL middleware uses **events** that represent a change in a context, as means to communicate between the components in the system. An event carries new information about the context and is associated with one context type. An example of an event for the aforementioned temperature context type is a measurement where temperature in the living room on 1st May, 2011 at 16:00 was 20°C.

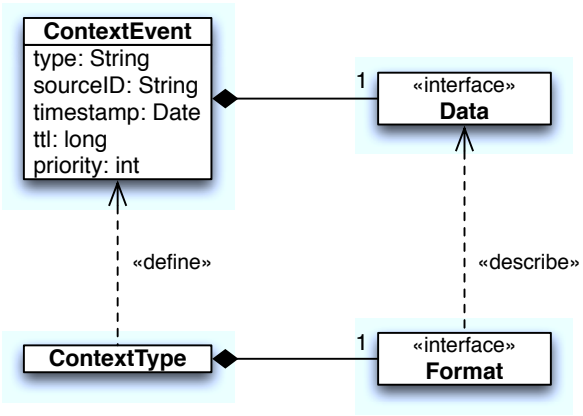


Figure 4.3: Context Event

Figure 4.3 shows a UML class diagram that defines a context event. First, each event is associated with a single context type using the `type` property to associate it with the name of context type. Second, it provides the name of its origin and time when change in context happened with the `sourceID` and `timestamp` properties respectively. The `ttl` property defines the time-to-live period of an event in milliseconds and together with `timestamp` can determine if event is still valid or not. For example, a time event that specifies change in minutes of an hour should only be valid for one minute. On other hand, if event should be valid until next event of same type and origin happens, we specify the `ttl` property to be 0. The `priority` property specify the importance of an event using an integer between 1 and 10 inclusive where 1 is the lowest possible priority and 10 is the highest one. Finally, a context event carries information about the environmental change using an instance of the `Data` class. The information must adhere to format that is specified in event's context type so the COPAL middleware can understand the event and process it.

Finally, it is important to emphasize that all context events in the COPAL middleware are *self-contained*. This requires them to serialize all the information necessary to be correctly distinguished and processed. Thus, event's `type`, `sourceID`, `timeStamp`, `ttl`, and `priority` properties are all persisted together with its `Data` when the event is serialized. This requirement easies the development of distributed COPAL, as we will see in Section 4.3, because an event can be easily transmitted between two instances of COPAL and will be understood by the receiving instance immediately without having to query the sending instance for additional information.

Gathering

In this section, we will focus on gathering environmental information from physical, virtual and logical sensors. We define *publishing* as process of gathering information about the environment and publishing it in a format understandable by other components in the COPAL middleware. We can distinguish two problems that have to be solved in this layer:

- *Reading environmental information from sensors.* This problem arises from heterogeneity of sensors and lack of specific standard for gathering environmental information from sensors. First, we can distinguish different ways of receive information from a sensor like pooling, broadcasting and multicasting changes from the environment. Second, there is an additional difference arising with respect to different transportation protocols that can be used in communicating with the sensor like TCP/IP, UDP, HTTP, etc.

- *Transforming the sensory information into context events.* This problem arises from heterogeneity of data formats used by the sensors to provide their sensory information. For example, sensors can format the sensory outputs using a custom binary format, XML (eXtensible Markup Language) [14], JSON (JavaScript Object Notation) [25], and so forth.

Publisher is a component that is responsible with encapsulation of one single solution for the problem of heterogeneity in communicating with sensors. Thus, each publisher is specifically developed to be used with one single sensor and can be considered as a mediator between the sensor and the COPAL middleware. Most important benefit of such a loosely-coupled design is that publishers may be added and removed from the system at any time without affecting functionality of other components in the COPAL middleware. Next, we distinguish and explain three main responsibilities of publishers:

- *Sensor data acquisition*. Each publisher is responsible for acquiring the raw information and providing it as the environmental information for the COPAL middleware. Sources typically comprise of physical sensors (e.g. temperature sensor), virtual sensors (e.g. calendar files) and logical sensors (e.g. databases). In addition, sensors may be attached locally or remotely via wired or wireless communication means and using various communication protocols. Hence, publishers are responsible to completely hide the acquisition complexity from the rest of the system.

- *Event creation*. A publisher is responsible to transform raw sensor information into context events that represent changes in the environment. Context events must use data format defined in their respective context types, which might be different from the format provided by the sensor. Therefore, the publisher is only component that is required to understand the data format provided by the sensor and has to be able to create context events from it. Thus, publishers are able to hide the complexity of heterogeneous sensor formats from the rest of the system.

- *Publishing strategy*. Publishers are responsible to decide when to publish context events. We can distinguish two distinct solutions: *time-based* and *change-based*. In time-based

publishing, context events are published in specific time intervals. This approach is suitable when change in the environment is continuous and happening all the time. For example, time is one example of environmental information that is changing continuously, therefore we have to publish time events in some predefined interval like every minute. In change-based publishing, context events are published if and only if the change from previously published context event of same type and from same source is greater than some predefined threshold. For example, we could publish a temperature event only when change from previously published temperature from same location is greeter than 1°C.

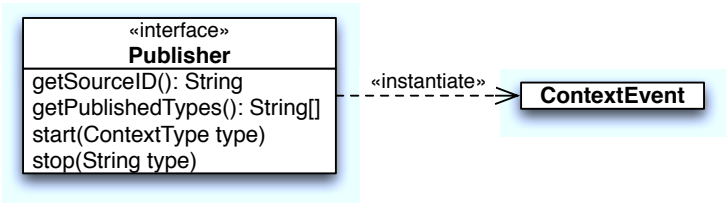


Figure 4.4: Publisher

Figure 4.4 shows a UML class diagram of the interface that all publishers in the COPAL middleware have to implement. First, a publisher is associated with a source ID, returned by the `getSourceID` method, that is used to define the origin of all context events published by the publisher. The source ID is unique between each publisher and is used in every context event publish by a publisher to specify its `sourceID` property. Furthermore, a publisher has to return names of all event types it will publish using the `getPublishedTypes` method. As context types can be registered and unregistered from the COPAL middleware at any point in time during the execution (see Figure 4.2), the COPAL middleware has to know which event types a publisher might publish so it can notify the publisher that it should start or stop publishing of events of those types. The mechanism to start and stop publishing in a publisher is implemented by the publisher, with the `start` and `stop` methods, as the mechanism is different between publishers with respect to how they communicate with sensors. Publishers have a guarantee from the COPAL middleware that it will invoke these methods as context types are registered and unregistered from the system.

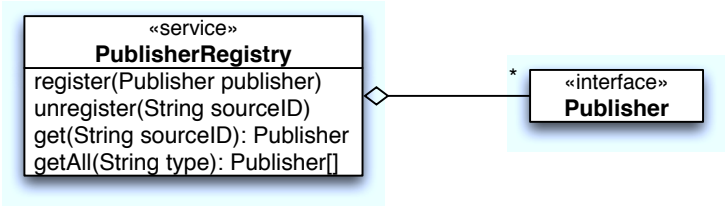


Figure 4.5: Publisher Registry

Furthermore, all publishers in the COPAL middleware are registered with the **publisher registry**, which offers a lookup service so other components can find and access the publishers

(Figure 4.5). First, the `register` and `unregister` methods allow to respectively add and remove publishers from the system. Second, the publisher registry provides two ways to find publishers: by a source ID using the `get` method and by a published type using the `getAll` method. The source ID lookup, as name says, uses the source ID to find a publisher. The lookup based on the published type finds all publishers that can publish context events of specified context type. Finally, the publisher registry is the responsible party for activating and deactivating publishers using their `start` and `stop` methods respectively. When a context type is registered with the COPAL middleware, the publisher registry will notify all publishers that they are allowed to publish context events of registered context type. Analogously, when a context type is unregistered, the publisher registry will notify all publishers that they must stop publishing context events of unregistered context type.

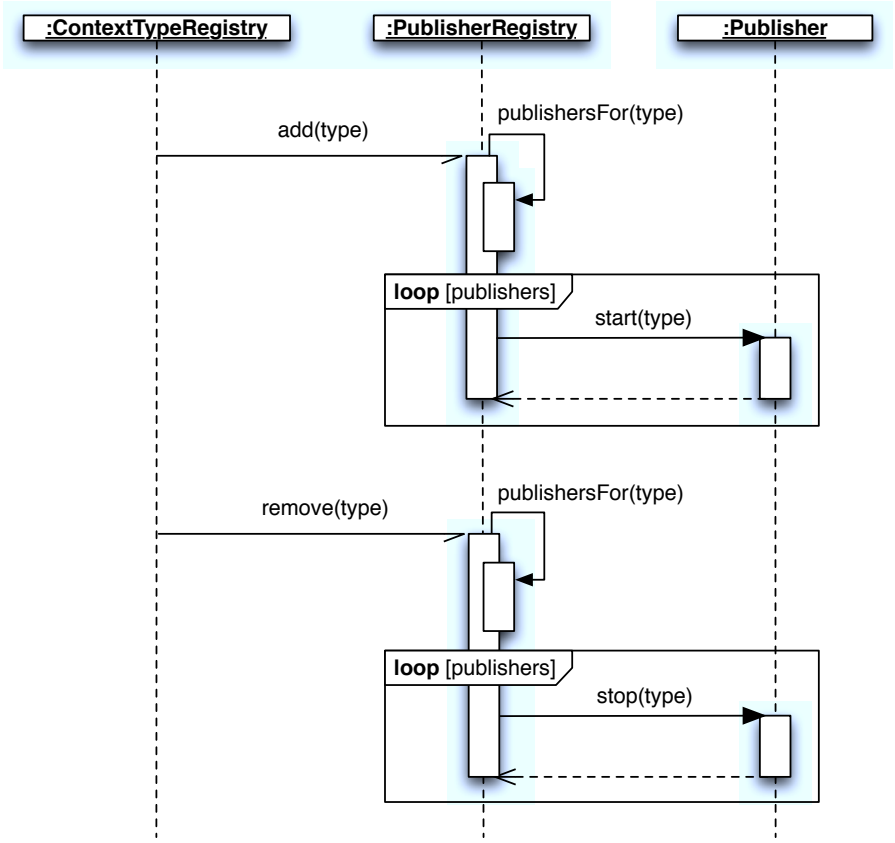


Figure 4.6: Starting and Stopping Publisher

Figure 4.6 shows a UML sequence diagram between context type registry, publisher registry and publisher as new context type is registered and unregistered. First, during registration of new context type, the publisher registry will receive an asynchronous notification that the context type has been registered. Then, the publisher registry will find all publishers that can publish this context type and will invoke their respective `start` methods with the context type as argu-

ment. Analogously, during unregistration of a context type, the publisher registry will find all publishers that can publish this context type and will invoke the publisher's `stop` method.

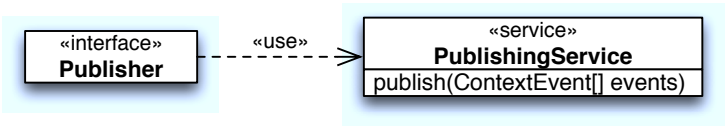


Figure 4.7: Publishing Service

Finally, it is worth mentioning that the actual publishing of context events is done using a **publishing service** that is provided by the COPAL middleware (Figure 4.7). This service provides a unified interface for publishers to use for publishing context events. The reason for providing the publishing service in the COPAL middleware is that it hides the actual underlying technology (see Section 5.1) used to transmit context events between components in the system. The publishing service implements only one method, the `publish` method, that allows atomic publishing of an array of context events where atomicity means that either all context events are published or none of them is. If there is any problem with publishing of any one of specified context events, the method will fail and throw a `FailedPublishignException` with a failure reason.

Reacting

On the other side of the COPAL middleware are context-aware services and applications. They utilize the COPAL middleware by subscribing to be notified about changes in the environment and to be able to react when they happen. Their main goal is to provide relevant information and/or actions to the user with help of the environmental information. For example, a service that controls air-conditioning (AC) in a house can ask user to specify temperature (t_e) and maximum deviation (Δt) from the temperature. The task of this service is to turn on the AC and set it to the specified temperature t_e whenever the current temperature in the house is above or below the maximum deviation ($t_e \pm \Delta t$). This will ensure that the temperature in the house is always comfortable for the user.

Next, we will explain two important features provided by the COPAL middleware, which allows it to support context-awareness in applications:

- *Callback invocation*. When a change in the environment occurs, the COPAL middleware is responsible with *asynchronous* invocation of all callback methods that are interested in this specific change in the environment. The programmer's task while developing context-aware applications is to implement all application's reactions to changes in the environment using the callback methods that do some specific actions based on the context event they receive from the COPAL middleware.

- *Continuous querying*. Context-aware application should be in control of which types of context events are delivered to which of their callback methods. This allows fine-grained control of the reactions to changes in the environment by defining that only certain callback methods are invoked when some specific change in the environment occurs. This

mechanism is supported in the COPAL middleware with creation of a query that defines a context event selection mechanism. Furthermore, all queries in the COPAL middleware run continuously and deliver all received events that pass their respective selection mechanism to callback methods that are associated with them. The continuous delivery of context events will run continuously until query is discarded by the context-aware applications that created it.

Listener defines a callback method in a context-aware application that the COPAL middleware can invoke on some context changes. Idea behind listeners is that they should be highly specialized tasks that are defined by the context-aware application and that should be invoked on some environmental changes. If the context-aware application provides multiple actions to the user, then it should implement each action as separate callback method. For example, the previously defined service that controls AC has two different actions: (1) to turn on AC when current temperature is outside the interval defined by the maximally allowed temperature deviation, and (2) to turn off AC when current temperature is inside the interval. Hence, we will require two separate callback methods: one that turns on AC and one that turns it off. Furthermore, callback methods do not define on which context changes they should be invoked. We expect that during runtime context-aware applications might change criteria when to invoke their callback methods as, for example, user preferences change. It would be very unreasonable to expect developers to implement separate callback method for each possible configuration choice. From out previous example, the AC control service has two configuration parameters, the expected temperature t_e and maximum deviation Δt, but the actions to turn on AC and to turn it off remain fundamentally same regardless of change in the parameters.

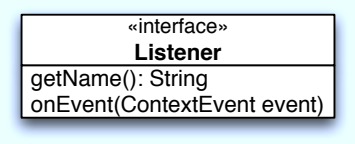


Figure 4.8: Listener

Figure 4.8 shows a UML class diagram of the interface that all listeners in the COPAL middleware have to implement. First, each listener is associated with a name, returned by the `getName` method, which separates it from the rest of listeners. This name should be a combination of name of the context-aware application and name of the callback method inside the application that this listener implements (e.g. `ACController.TurnACOn`). Second, the interface defines the `onEvent` method that expects a context event as argument that represents our aforementioned callback method. The COPAL middleware will invoke this method with context events using continuous query mechanism that is explained next.

Query is a component that is provided by the COPAL middleware to support continuous querying of context for changes. It supports two mechanisms to select events from the context: (1) name of event type (e.g. temperature), and (2) optional criteria based on event's fields (e.g. value > 10°C). Furthermore, queries are also responsible for asynchronous delivery of context events to all registered listeners. This mechanism assures that context-aware applications do

not have to directly pool the COPAL middleware for new context events, but instead, they create queries and register their callback methods with them. A query will start receiving context events from the COPAL middleware as soon as it is created and will continue to receive them until it is destroyed. When a context event is received, it will invoke all currently registered listeners and will pass them the received context event as argument for their `onEvent` method. Multiple listeners can be registered with a query during runtime and each registered listener will receive context events from the query until it is unregistered from the query or the query is destroyed.

	Operation	Name	Returns
Logical	not a	*negation*	true iff a is false
	a and b	*conjunction*	true iff a and b are both true
	a or b	*disjunction*	true iff at least one of a and b is true
Comparison	$a = b$	*equality*	true iff a is equal to b
	a != b	*inequality*	true iff a is not equal to b
	$a < b$	*less*	true iff a is less than b
	a <= b	*less-or-equal*	true iff a is less than or equal to b
	$a > b$	*greater*	true iff a is greater than b
	a >= b	*greater-or-equal*	true iff a is greater or equal to b
	a is null	*presence*	true iff a is not defined in an event
	a is not null	*absence*	true iff a is defined in an event
Arithmetic	$a + b$	*addition*	sum of a and b
	$a - b$	*subtraction*	difference of a and b
	$a * b$	*multiplication*	product of a and b
	a / b	*division*	integral quotient of a and b
	a % b	*modulo*	remainder of dividing a with b
String	a || b	*concatenation*	string ab

Table 4.1: Operations in Query's Criteria

As previously mentioned, criteria in a query is an optional Boolean statement that allows queries to further filter out context events based on event's fields. Table 4.1 lists all allowed operations in query's criteria and Appendix B shows the complete EBNF (Extended Backus–Naur Form) [2] for the criteria.

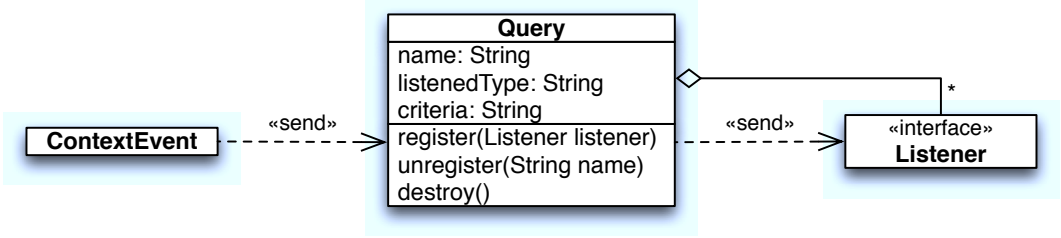


Figure 4.9: Query

Figure 4.9 shows a UML class diagram of the query class in the COPAL middleware. First, each query has a unique `name` property that distinguishes it from other queries and allows context-aware applications to reference and retrieve queries from the COPAL middleware. Second, it has a `listenedType` property and an optional `criteria` property that will be used to filter out interesting events. Furthermore, the `register` and `unregister` methods allow to respectively add and remove a listener from receiving context events that pass the query's selection mechanism. Finally, the `destroy` method will unregister all currently registered listeners and notify the COPAL middleware to discontinue delivery of any future context events to the query.

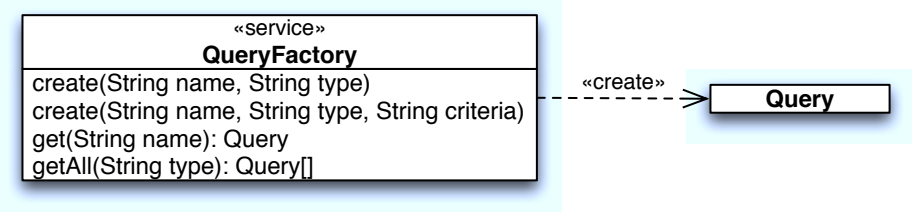


Figure 4.10: Query Factory

The COPAL middleware provides a **query factory** service that is used to create queries (Figure 4.10). The query factory implements two overloaded `create` methods to create a query with or without criteria. These methods will create an instance of query class that will use underlying libraries (see Section 5.1) to continuously deliver context events to its listeners. Moreover, the query factory assures that if a query with same name, listened type and criteria was previously created by some other service, then no new query instance will be created, and instead the previously created instance will be returned. Conversely, if we try to create a query with same name but different listened type or criteria, the query factory will throw a `RedefinitionOfQueryException` signaling the reason of failure. Finally, we can retrieve an already created query using its name with the `get` method or all created queries that listen on some specific event type using the `getAll` method.

Processing

We define *processing* of a context event as using the event as input to create zero or more output context events. In the COPAL middleware, processing phase of a context event comes after the event has been published and can consists of zero or more steps before the event is consumed by context-aware applications. Output in each step of processing can consist of any number of new context events. Optionally, a modified version of input event can be a result of processing step and the modified version of the input event will be used as the input for subsequent processing steps. Conversely, if the input event is not in the list of output context events, the unmodified version will be used as the input in subsequent processing steps.

We define a processing step in a context event by adding an **action** into the event. Therefore, processing phase of a context event is defined with zero or more actions. Each action (Figure 4.11) contains two properties: `name` and a Boolean value specifying if action is `required`.

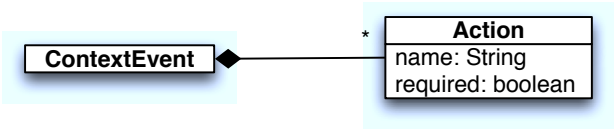


Figure 4.11: Action

Action's name together with type of the context event uniquely defines which action needs to be done on the event. For example, Celsius temperature event might contain an action `ToFahrenheit` that creates a new Fahrenheit temperature event. This action is different from the `ToFahrenthit` action in a Kelvin temperature event because context types are different and we will require different equations to calculate Fahrenheit from Kelvin and Fahrenheit from Celsius. Requirement of an action specifies if the action is mandatory or not. If we cannot process some mandatory action in a context event, the event will be removed from the system and will never reach context-aware applications. Conversely, if we cannot process an optional action in a context event, the action will be skipped and processing will continue with any subsequent action.

Processor is component in the COPAL middleware that can process actions on some input context events. Each processor defines a set of pairs $(\textit{action name}, \textit{context type})$ that define which actions on which context type the processor can handle. For example, let say that a context event of type t requires an action a. If a processor specifies that it can do the action a on the context type t, this processor will be invoked by the COPAL middleware to handle the action. Furthermore, each processor action has to specify which types of context events can be output of invoking the action. For example, if a processor action defines t_1 and t_2 as types of output events, result of invoking the action can be any of these cases: (1) no context events, or (2) any number of context events of type t_1 and t_2.

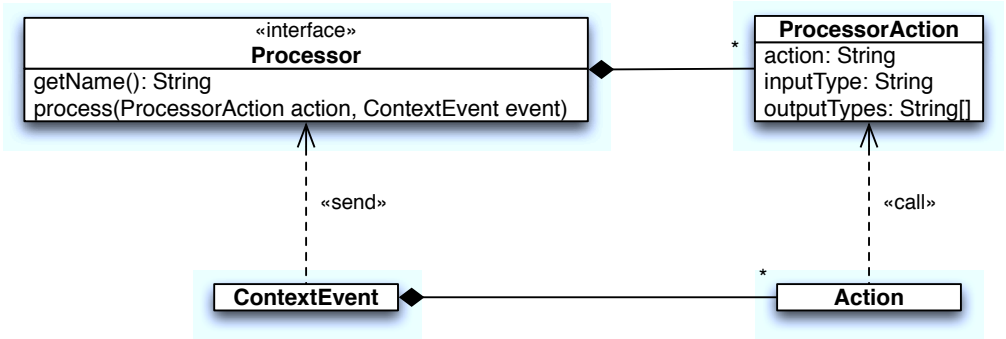


Figure 4.12: Processor

Figure 4.12 shows a UML class diagram of the interface that all processor in the COPAL middleware have to implement. First, a processor has a unique name, returned by the `getName` method, which is used to specify which processor was used to handle an action in a context event. Second, each processor specifies which actions it can do with the array of `ProcessorAction`

instances. Each processor action has three properties: name of action, name of input context type, and array of names of output context types. Finally, processor must implement the `process` method that is invoked by the COPAL middleware with a processor action and a context event as arguments for the method. This method returns an array of context events that represent the result of invoking specified action on specified input context event. The types of output context events must be in the array of output context types for specified action.

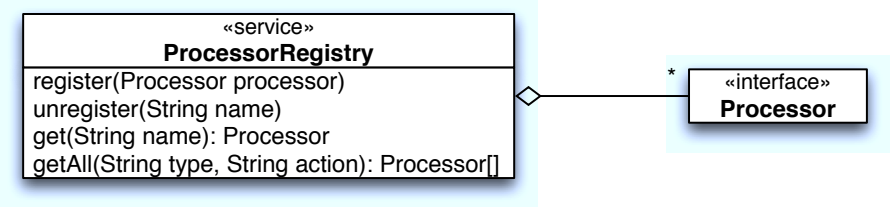


Figure 4.13: Processor Registry

The COPAL middleware requires all processors to be registered so the system can know which actions can be handled and which cannot. The **processor registry** (Figure 4.13) is a service that is provided by the COPAL middleware that allows context-aware applications to add and remove processor from the system using the `register` and `unregister` methods. Furthermore, the processor registry provides two ways to find processors: by a name using the `get` method and by a (*action name*, *context type*) pair using the `getAll` method. The name lookup, as name says, uses the name to find a processor. The lookup based on the (*action name*, *context type*) pair finds all processors that can handle specified action on specified input context type.

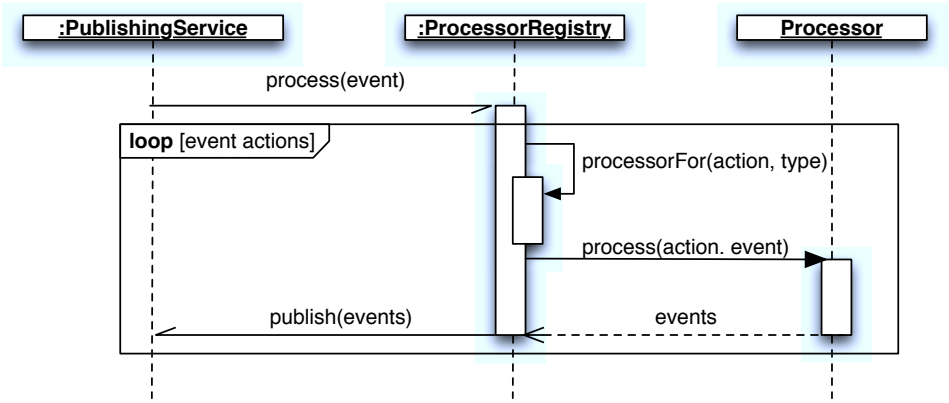


Figure 4.14: Process Event

Figure 4.14 shows a UML sequence diagram between publishing service, processor registry and processor as a context event is published that requires an action to be processed. First, if during publishing of context events there is a context event that needs processing, the event will be delivered to the processor registry for processing. The processor registry will find a suitable

processor for specific context type and action and will invoke the processor with action and the context event as arguments for the `process` method. Finally, the output context events, which are returned by the `process` method, will be further passed to the publishing service to be published as new events in the context.

Summary

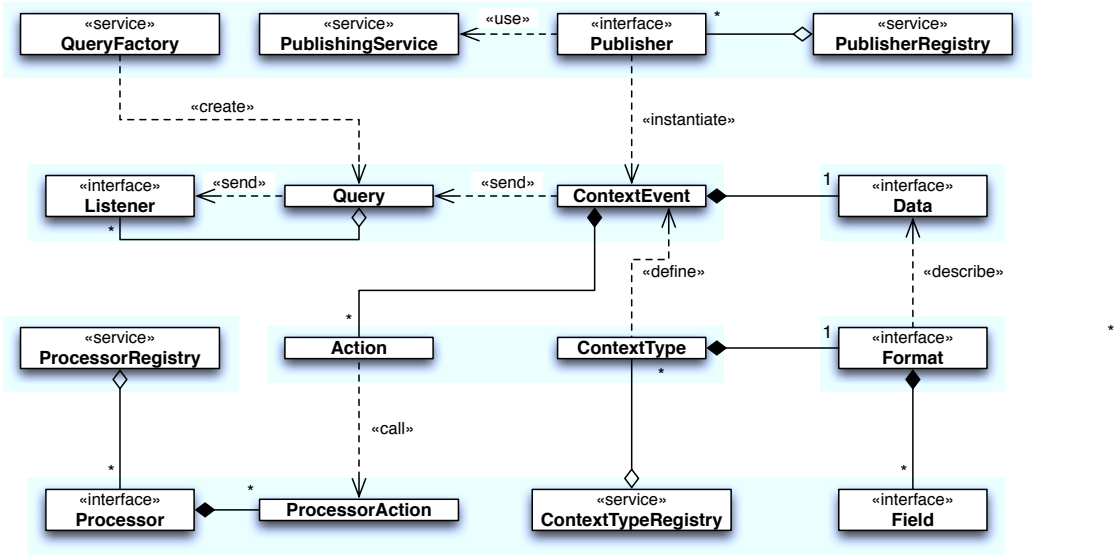


Figure 4.15: COPAL Components

This section explained many details about how context is modeled in the COPAL middleware, how environmental information is gather and processed using context events, and how context-aware applications can continuously query and listen to context events. Figure 4.15 shows the summary of all COPAL components that have been explained in this section and details how the components interact and come together to create the COPAL domain model.

4.2 Processing Patterns

Understanding and utilizing processors can help with designing and constructing complex context provisioning schemes. Hence, we differentiate five primitive patterns that the COPAL middleware supports and can be used as building blocks when constructing a context-provisioning schema: *filter*, *enrichment*, *peeling*, *abstraction*, and *differentiation*. These patterns are inspired by the work in complex event-processing [50] and event-processing networks [65], and are adapted specifically for context provisioning.

Figure 4.16: Filter

Filter

Filter processing pattern (Figure 4.16) excludes context events from being further processed and the excluded events will never reach context-aware applications. The basic idea is to exclude context events based on some QoS (Quality of Service) criteria. For example, a RFID location sensor may occasionally provide some false-positive readings. A location filter would be responsible with assigning a confidence level with each location event based on the sensor's signal strength and pruning those events that are under some specific threshold.

Enrichment

Figure 4.17: Enrichment

Enrichment processing pattern (Figure 4.17) adds additional information into context events. The additional information can be derived from the information that is already in the context events or it can be queried from some external source (e.g. file system, database, web service, etc.). For example, we could add the name of person into a location event based on the RFID tag number. For this, we would could create a database with associations between RFID tag numbers and names, and query it whenever a location event is published.

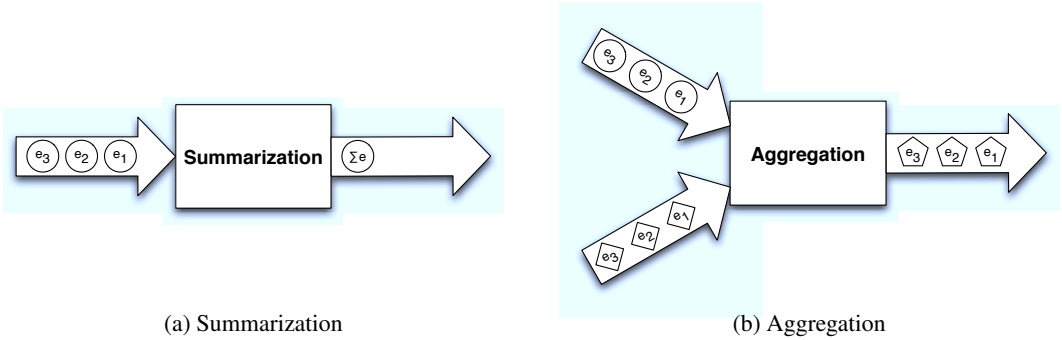
Peeling

Figure 4.18: Peeling

Peeling processing pattern (Figure 4.18) removes information from context events. Some information after processing may not be required anymore or it would be a security risk if context-

aware applications could access it, and hence, it should be removed from context events. For example, after adding the name of person into a location event, we can safely remove the RFID tag number because no service should depend on it.

Abstraction



(a) Summarization (b) Aggregation

Figure 4.19: Abstraction

Abstraction processing pattern (Figure 4.19) creates new context events to indicate new information about the environment, which can be inferred from occurrence or absence of one or more other context events. The abstraction can be furthered separated into two distinct processing patterns: *summarization* and *aggregation*. *Summarization* processing pattern (Figure 4.19a) gathers context events of same type that have occurred based on some criteria like time span or number of events, and publishes a summarizing context event. For example, the power usage in a whole house can be a summary of power usages of all appliances in the house. Finally, *Aggregation* processing pattern (Figure 4.19b) gathers context events of different types and infers a context event of new type. For example, we can infer that a person is watching a television by observing that the person is in a same room where the television is and that the television is turned on.

Differentiation

Differentiation processing pattern (Figure 4.20) uses a context event of some type to infer new context events of different types. Using differentiation, same information can be forked and fed to different processors and listeners at different granularity or abstraction levels. For example, using an absolute coordinates of a person in a house, we can infer and publish new event, beside the old absolute location, specifying which room the person is in. This new context event can then be used to determine settings for lightning in the house based on which room the person is in.

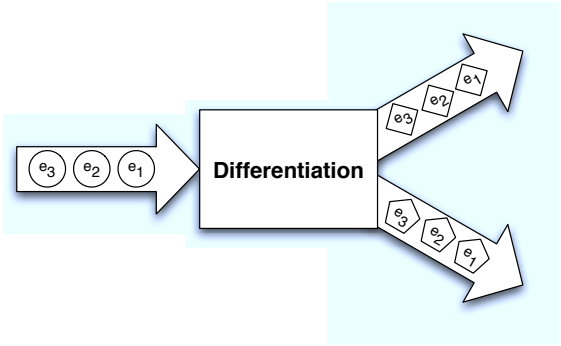


Figure 4.20: Differentiation

4.3 Distribution

Distributed COPAL is an extension to the COPAL middleware. It supports creation of distributed network of COPAL instances that share environmental information between each other. Instances can be on separate machines and are connected with each other through a transport layer. The approach on how to design a functioning network of distributed COPAL instances requires three important questions to be answered:

1. What is the organization of COPAL instances in a distributed network?
2. Which information needs to be shared between COPAL instances?
3. How is information shared between COPAL instances?

The COPAL middleware uses a hierarchical approach with a tree structure to layout nodes in a distributed network of COPAL instances. Each node in the network can have many downstream nodes, but can only have one upstream node. The information about the environment flows from the downstream nodes towards upstream nodes. Therefore, the more we go upstream, the more information about the environment we have. This topology inherently creates a tree-like structure from the nodes in the system where topmost upstream node will contain all the information about the environment. An example of this tree-like topology is shown in Figure 4.21.

This approach in organizing nodes in distributed COPAL was taken because in our primary use-case, that involves smart homes, we have a clear separation of entities in topology: buildings consist of floors that consist of apartments that consist of rooms. We can see that this use-case also creates a tree-like structure between each entity with part-of associations between them that easily fits the COPAL organization of distributed network. Therefore, in this scenario each entity would use a separate COPAL instance where room instances will use the apartment instance as the upstream node, the apartment instances will use the floor instance, and the floor instances would use the building instance. This topology allows us to create context aware services that only receive environmental information on the granularity level that it is interested in. For example, a context-aware service that is only interested in temperature in one room of an apartment would only have to connect to the COPAL instance associated with this room.

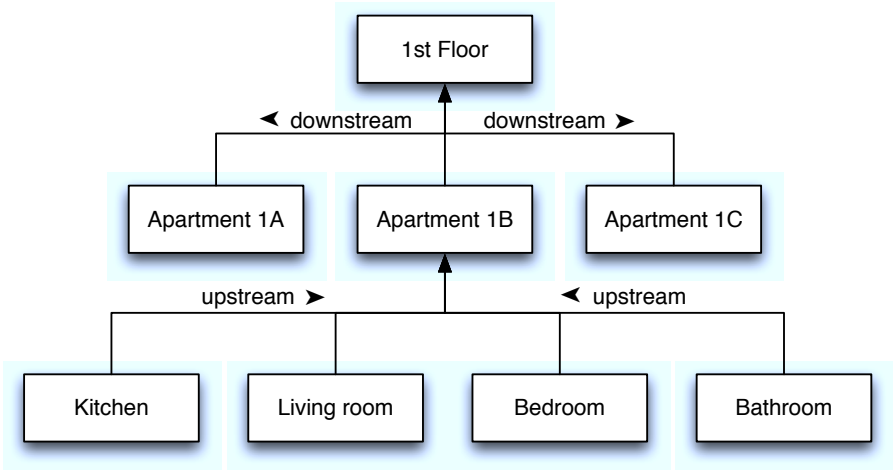


Figure 4.21: Distributed COPAL Nodes

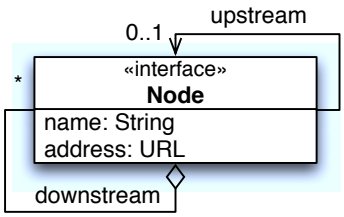


Figure 4.22: Distributed COPAL Node

Figure 4.22 shows a UML class diagram of the interface that distributed COPAL nodes have to implement. Each node has a unique `name` property to distinguish it from all other nodes and an URL (Uniform Resource Locator) [9] `address` property where it can be reached. Moreover, the node is associated with many downstream nodes and at most one upstream node. The actual implementation of the nodes will depend on the approach on how information is distributed between COPAL instances, as we will see later in this section.

With respect to sharing information between distributed COPAL nodes, first and foremost, we need to share information about the environment between nodes in a distributed network. This requires context events to flow from downstream nodes towards upstream nodes and also their associated context types to be registered with all upstream nodes. Therefore, if we register a context type in a downstream node, additionally all upstream nodes have to receive this context type as they would need to understand context events of this type that are published in the downstream node. Next, publishers and listeners can use any node in the system and they are only allowed to communicate with the nodes that they are registered to. Hence, they do not require to be distributed between the COPAL instances, as they are the ones that are in control of which instances they require and use. On other hand, processors do need to be distributed, as a node might need a processor for a context event that is not available locally. The sharing of processors between the nodes in the system goes in reverse from sharing events — a processor registered

in an upstream node can be only used in downstream nodes. To see why complete sharing of processors between all nodes in the system is not reasonable, let us consider a case with a processor that can resolve a user name from a user identification number and two COPAL instances associated with two apartments in a building. The aforementioned processor is registered with second instance and provides resolution only for residents in the second apartment. If a context event that requires this processor was published in first apartment, it would be unreasonable to use the processor from second apartment, as residents are different. Therefore, our topology should only allow sharing of processors registered in upstream nodes, as for example we could have a processor that resolves user names on the level of building which could be used for any context event that requires this processor in any apartment in the building.

The only remaining component that might require sharing between nodes in a distributed system is a query. This component requires further understanding of how the distribution of context events is handled in a distributed system before the question if sharing queries is required can be answered. We can distinguish two approaches to send context events in the distributed system, namely eager and lazy, which will be explained next.

Eager Approach

In the eager approach, all context events that are published locally or received from downstream nodes are sent to the upstream node. In this approach, queries only have to be created locally in a node, as we will receive context events nevertheless if there is a query that catches them in the node. This approach has a drawback that bandwidth in a distributed network is wasted if none of the upstream nodes is interested in a particular context event, but it has advantage that is much simpler to implement and that we can be sure that when a query is created it will immediately start to receive context events from downstream nodes.

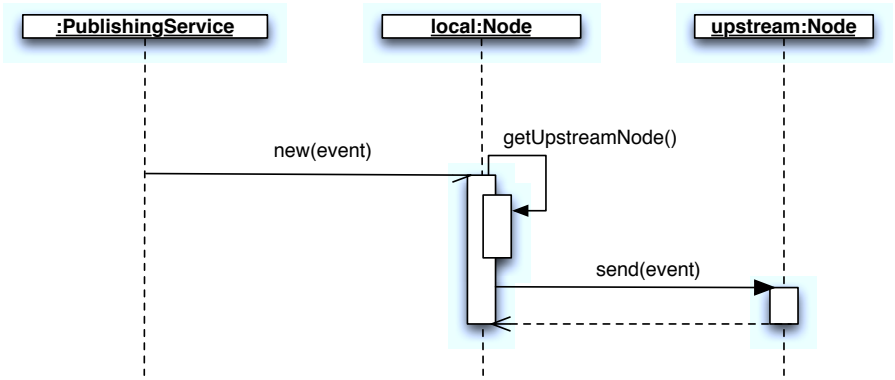


Figure 4.23: Eager Distribution

Figure 4.23 shows a UML sequence diagram between publishing service, local node and upstream node as new context event is published. We can see that publishing service will invoke the local node whenever a new context event is published. This will in turn send this context event to the upstream node so the event can also be published there.

Lazy Approach

In the lazy approach, only context events that upstream node is interested in are sent to it. The interest in a context event is tightly coupled with queries and listeners. If we have a query that has listeners registered to it in some upstream node, we have to announce the interest in context events that this query can catch to all downstream nodes. Therefore, the sharing of queries between the nodes is required in this approach and it uses same principle as sharing processors — queries created in an upstream node have to be created in all downstream nodes and the upstream node has to be registered as listener with the downstream nodes. In this case, only context events that are published locally should be transmitted to upstream nodes to avoid duplicate receipt of events in the upstream nodes. This approach does require either less or at most same amount of bandwidth as the eager approach, but it is also much more complex to implement and it does not guarantee that all context events from downstream nodes are received after a query has been created as there might be delay between when an event has been published and query has been created in a downstream node.

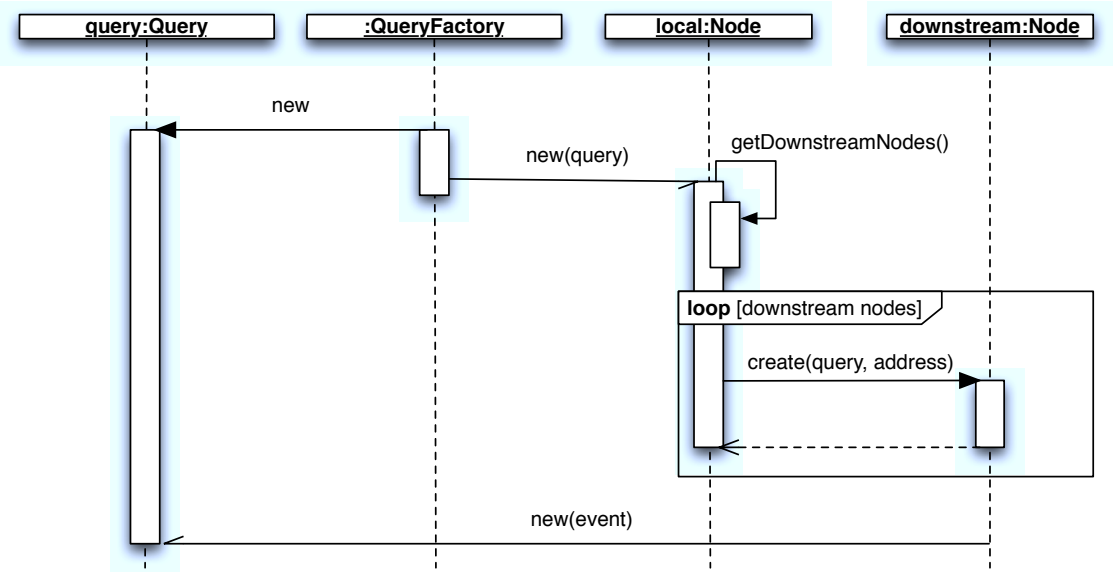


Figure 4.24: Lazy Distribution

Figure 4.24 shows a UML sequence diagram between query factory, query, local node and a downstream node as new query is created and a context event is published. First, the query factory will notify the local node as new query is created and this will in turn invoke `create` query method on all downstream nodes and pass the `address` property of the local node as additional argument so the local node can be immediately registered as listener in all newly created downstream queries. Finally, when a context event is published in a downstream node, it will directly invoke the query in the local node passing the event as argument.

CHAPTER 5

Implementation

In this chapter, the implementation of the COPAL middleware is presented. Section 5.1 will provide a short introduction into technologies used in the development of the COPAL middleware. The underlying mechanism for publishing, processing, and querying events in the COPAL middleware will be explained in Section 5.2. Section 5.3 will present two tools that are implemented in the COPAL middleware to support the communication between distributed COPAL instances. Finally, the separation of the COPAL middleware into modules and deployment options will be described in Section 5.4 and Section 5.5 respectively.

5.1 Technologies

This section will provide a short introduction into technologies used in the development of the COPAL middleware.

Esper

Esper[1] [32] is a complex event processing component that is designed for high-volume event correlation where millions of events coming in would make it impossible to store them all for later querying using the classical database architecture. It is a lightweight kernel written in Java which is fully embeddable into any Java process, Java EE application server, or Java-based Enterprise Service Bus, and enables rapid development of applications that process large volumes of incoming messages or events. A tailored Event Processing Language (EPL) allows expressing rich event conditions, correlation, and spanning time windows, thus minimizing the development effort required to set up a system that can react to complex situations.

Relational databases or message-based systems such as Java Message Service (JMS) make it really hard to deal with temporal data and real-time queries. For example, databases require explicit querying to return meaningful data and are not suited to push data as it changes. JMS

[1] http://esper.codehaus.org/

systems are stateless and require the developer to implement the temporal and aggregation logic himself. By contrast, the Esper engine provides a higher abstraction and can be thought of as a database turned upside-down. Instead of storing the data and running queries against stored data, Esper allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match user defined queries. The execution model is thus continuous rather than just momentary when a query is submitted.

Esper defines an *event* as an immutable record of a past occurrence of an action or state change, and event properties capture some useful information in the event. An event can be represented by any of the following types: (1) Java Beans, (2) Maps, and (3) XML documents. EPL is a SQL-like language with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY` clauses. *Event streams* replace tables as the source of data with events replacing rows as the basic unit of data. An event stream is a time-ordered, unbounded sequence of events in time. Thus, events can only be append into a stream and cannot be removed. EPL *statements* are used to derive and aggregate information from one or more streams of events, and to join or merge event streams. *Subscribers* of statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, views, filters and output rate. Lastly, a *sender* sends events into the Esper engine for some specific event type.

REST

In the concept of Service-Oriented Architecture (SOA) [31], services are used as fundamental software elements for developing applications. It defines services as "self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications" [51]. One possible and very popular way to implement SOA applications is by means of web services. The W3C defines a web service as "a software system designed to support interoperable machine-to-machine interaction over a network" [41].

The REST is a software architectural style of implementing web services. It recognizes that the web is composed of resources that are accessed using URIs (Uniform Resource Identifiers) [10]. The RESTful architectures consist of clients and servers where clients initiate requests to servers, and servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can essentially be any coherent and meaningful concept that is addressable, but most-often resources are arranged same as an underlying data model in a web service.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- *Resource identification using URI*. A RESTful web service exposes a set of resources that identify access points of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.

- *Uniform interface*. Resources are manipulated using a fixed set of four create, read, update, delete HTTP operations — namely PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then removed by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.

- *Self-descriptive messages*. Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as XML, JSON, plain text, and others.

- *Stateful interactions through hyperlinks*. Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Furthermore, information can be embedded in response messages to point to valid future states of the interaction.

Apache CXF[2] is an open source services framework that helps with building and developing services using frontend programming APIs like JAX-RS (Java API for RESTful Web Services [42]). JAX-RS is a Java programming language API that provides support in creating web services according to the REST architectural style and is an official part of Java EE 6. It uses annotations to simplify the development and deployment of web services and their respective endpoints. Developers decorate Java programming language classes with JAX-RS annotations to define resources and the actions that can be performed on those resources.

OSGi

The OSGi[3] (Open Services Gateway initiative) [6] framework is a module system and service platform for the Java programming language that provides a general purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications known as *bundles*. An OSGi-compliant container manages the installation and update of bundles in a dynamic and scalable fashion. To achieve this, it manages the dependencies between bundles in detail. Furthermore, it provides developers with the resources necessary to take advantage of Java platform independence and dynamic code-loading capability in order to easily develop services that can be deployed on in a larger system.

In the OSGi framework, each bundle is a tightly-coupled, dynamically loadable collection of interfaces, classes, and configuration files that explicitly declare their external dependencies. They are the only entities for deploying Java-based applications in the OSGi framework and are packaged as a Java ARchive (JAR) with a well-defined manifest file that specifies information that the OSGi framework needs to install correctly and activate the bundle. Furthermore, the OSGi framework defines six states in *bundle's life-cycle* shown in a UML state diagram (Figure 5.1) and explained in Table 5.1.

If a bundle defines a *bundle activator* in its manifest file, the OSGi framework will start the bundle by invoking the activator. The `BundleActivator` interface (Listing 5.1) defines methods that the OSGi framework invokes when it starts and stops the bundle. The `start` method can be used to register services or to allocate any resources that the bundle needs. If this method throws an exception, the bundle will be marked as stopped. Furthermore, the OSGi framework creates a `BundleContext` instance and provides this object as an argument to the `start` method. The `BundleContext` interface defines methods to retrieve information about bundles installed in the OSGi framework. Finally, The `stop` method is called when the bundle

[2] http://cxf.apache.org/
[3] http://www.osgi.org/

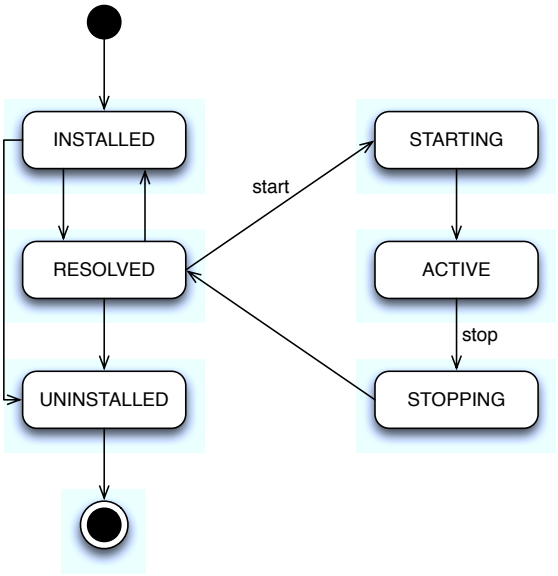


Figure 5.1: OSGi Bundle Life-Cycle

is stopped so the OSGi framework can perform the bundle-specific activities necessary to stop the bundle. In general, this method should undo the work that the `start` method has done.

Listing 5.1: Bundle Activator Interface

```
1 public interface BundleActivator {
2
3     void start(BundleContext context) throws Exception;
4
5     void stop(BundleContext context) throws Exception;
6 }
```

Lastly, the OSGi frameworks provide a mechanism to connect bundles in a dynamic way by offering a publish-find-bind model for defining and retrieving *services*. A service is defined semantically by its service interface and implemented as a service object. The service object is owned by, and runs within, a bundle. The bundle must register the service object with the OSGi framework service registry so that the service's functionality is available to other bundles. The OSGi framework transparently manages dependencies between a bundle that owns a service and bundles that use it. For example, when a bundle is stopped, all services registered by that bundle will be automatically unregistered. The OSGi framework maps services to their underlying service objects, and provides a simple query mechanism that enables a bundle to request services it needs. The OSGi framework also provides an event mechanism so that bundles can receive events of services that are registered, modified, or unregistered. The service query and event mechanism can be accessed through the `BundleContext` interface.

State	Description
INSTALLED	A bundle has been successfully installed.
RESOLVED	All dependencies that the bundle requires are present and available. This state indicated that the bundle is either ready to be started or has been stopped.
STARTING	The bundle is being started, i.e. the `BundleActivator`'s `start` method is executing.
ACTIVE	The bundle has been successfully started and is running, i.e. the `BundleActivator`'s `start` method has returned without an exception.
STOPPING	The bundle is being stopped, i.e. the `BundleActivator`'s `stop` method is executing.
UNINSTALLED	The bundle has been successfully uninstalled.

Table 5.1: Bundle Life-Cycle

5.2 Publishing, Querying, & Processing

In the COPAL middleware, Esper is used to transmit context events between publishers, processors and listeners. Thus, whenever a context event is published, it is passed to Esper engine to be delivered to processors that can process event's actions and afterwards to queries that are interested in the event. This integration requires from COPAL middleware:

- to provide an implementation of the publishing service (see Figure 4.7) that will be used to send context events using event streams, and
- to deliver context events to COPAL processors and queries using Esper statements and subscribers.

Publishing

Publishing of context events requires from the COPAL middleware to use event streams to send events. It will create a new event stream for each type of context event and use the appropriate event stream to publish an event.

Figure 5.2 shows a UML sequence diagram for registering COPAL context types with Esper. First, during registration of new context type, the Esper publishing service will receive an asynchronous notification that the context type has been registered. Then, the publishing service will register this type into the Esper runtime engine and retrieve a sender for this type to be used whenever a context event of this type requires publishing. Analogously, whenever the Esper publishing service receives an unregistered notification for a context type from the registry, it

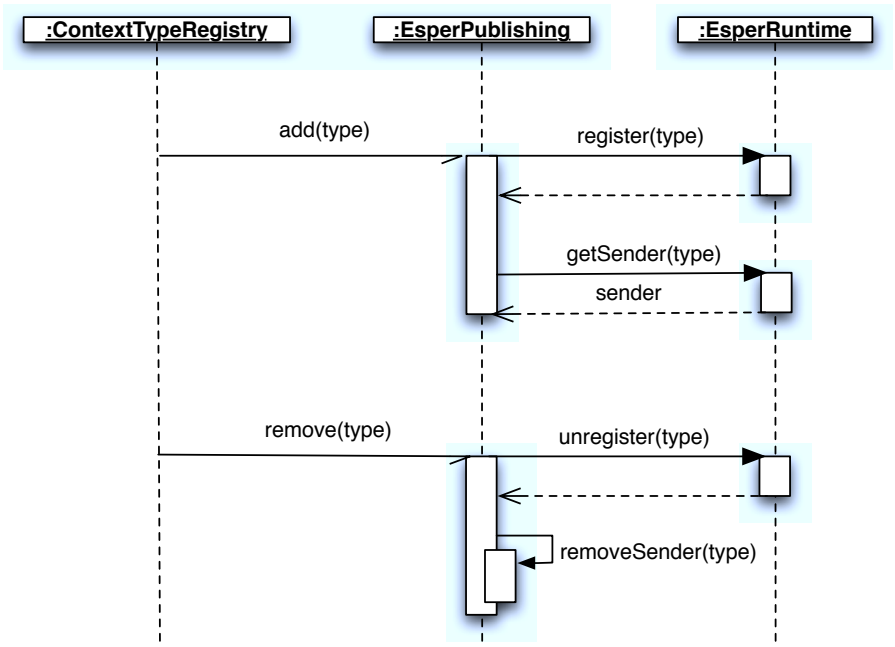


Figure 5.2: Esper Registration of Context Types

will unregister the type from the Esper runtime engine and remove the sender for this type so it cannot be used anymore for publishing.

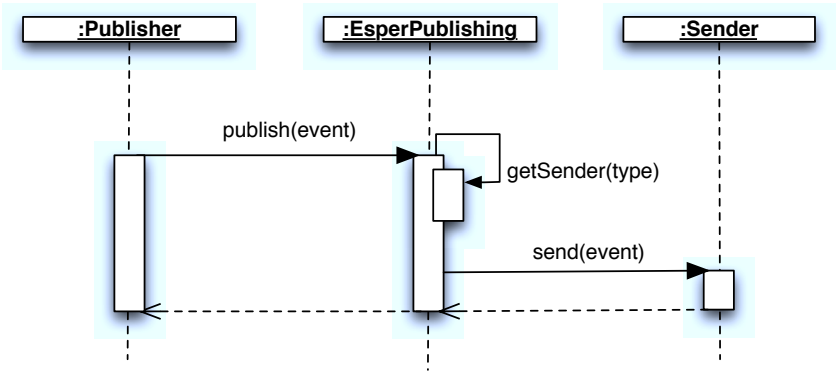


Figure 5.3: Publishing of Context Events

Figure 5.3 shows a UML sequence diagram for publishing context events using Esper. Whenever a context event is published, the Esper publishing service will first find a suitable sender for the event's type. Then, the sender will be used to send the context event into the appropriate event stream.

Querying

Querying of context events requires from the COPAL middleware to select events from an event stream. It will create an EPL `SELECT` statements for each query with the appropriate `FROM` and `WHERE` clause.

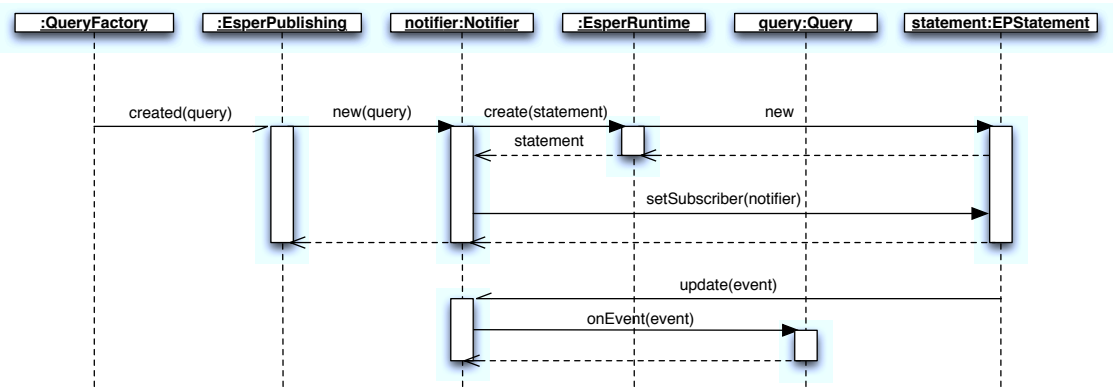


Figure 5.4: Querying of Context Events

Figure 5.4 shows a UML sequence diagram for delivering context events to COPAL queries using EPL `SELECT` statements and notifiers. First, during creation of new query, the Esper publishing service will receive an asynchronous notification that the query has been created. Then, the publishing service will create a new notifier with the query as an argument that will in turn create an EPL `SELECT` statement using the Esper runtime engine and register itself as subscriber to the statement. Finally, whenever the notifier receives an update notification from the statement with a context event as an argument, it will invoke the `onEvent` method in the query with the event that will in turn notify all listeners of the query.

Processing

Finally, processing of context events requires from the COPAL middleware to intercept events that require processing from event streams and delivering them to appropriate processor. Thus, it will create an EPL `SELECT` statement to select context events that require processing from event streams. For this method to function correctly, COPAL queries have to generate EPL statements that only select context event that do not require processing or context events for which processing is already finished. Then, the COPAL middleware can generate for each processor action an EPL statement that selects context events that require this action.

Figure 5.5 shows a UML sequence diagram for delivering context events to COPAL processor using EPL `SELECT` statements and notifiers. First, during registration of new processor, the Esper publishing service will receive an asynchronous notification that the processor has been registered. Then, the publishing service will create for each processor's action a new notifier with the processor and the action as arguments. The notifier will in turn create an EPL `SELECT`

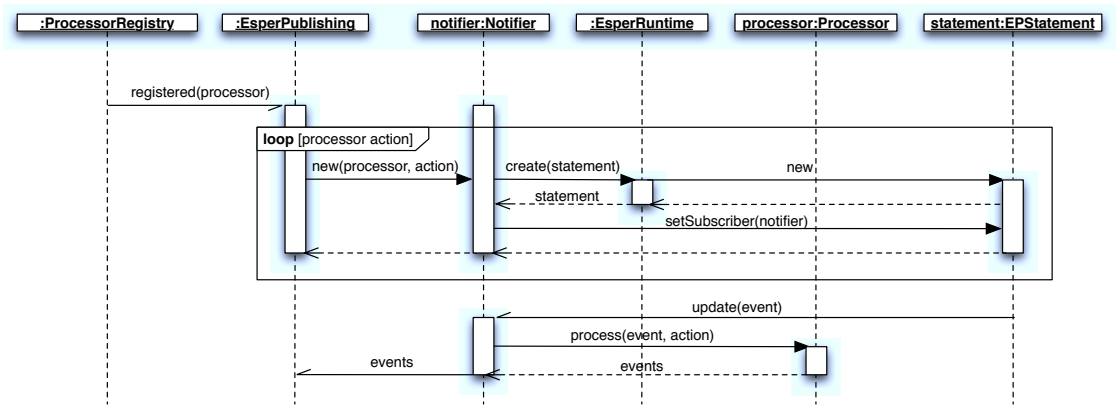


Figure 5.5: Processing of Context Events

statement using the Esper runtime engine and register itself as subscriber to the statement. Finally, whenever the notifier receives an update notification from the statement with a context event as an argument, it will invoke the `process` method in the processor with the action and the event. Any resulting output events from invocation of the process method will be published using the Esper publishing service.

5.3 Distribution

Distributed version of the COPAL middleware requires two tools for it to function correctly:

- A transformation mechanism that can convert any COPAL component into a machine-readable representation.
- A transportation mechanism that is used to transfer COPAL components between distributed COPAL instances.

Marshaling & Unmarshaling

In the COPAL middleware, *marshaling* is defined as process of transforming a COPAL component into a machine-readable representation. The reverse process of marshaling is called *unmarshaling* and it is used to transform a machine-readable representation back to identical COPAL component. Furthermore, we say that a marshaller is a class that is responsible for marshaling an instance of some COPAL type into a specific format, and analogously, an unmarshaller is a class that can unmarshal the COPAL instance back from the same format.

In the COPAL middleware, we use this mechanism to be able to transform COPAL components into an XML DOM (Document Object Model) [45] tree and vice versa. We use XML DOM tree because it provides us with an object-oriented representation of an XML document that can be read and modified in-memory before it is stores as an XML document. The

XML DOM implementation in the Java standard library (package `org.w3c.dom`) defines an `Element` interface that is used in marshaling and unmarshaling of COPAL components. The `Element` interface represents an element in an XML document. XML elements may have attributes associated with them and the interface defines methods `getAttribute` and `setAttribute` to retrieve and store an attribute value by name. Furthermore, XML elements may also have child elements associated with them and the interface defines methods to append, removed, replace, and retrieve child elements.

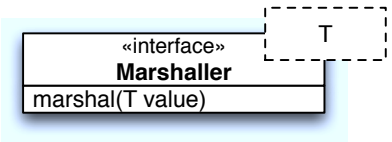


Figure 5.6: Marshaller

Figure 5.6 shows a UML class diagram that defines an interface that all **marshallers** in the COPAL middleware have to implement. It is a generic interface where `T` is type of `Object` that is marshaled. The marshaller interface defines only the `marshal` method that expects an argument of type `T`, which will be marshaled. The important aspect of this interface is that it does not create any dependency on particular format that should be used to marshal the argument and delegates the choice of format to implementations of concrete marshallers. Thus, developers are free to use any format they prefer like XML, JSON, plain text, or a custom binary format.

Listing 5.2: Integer Marshaller

```
1  public class IntegerMarshaller implements Marshaller<Integer> {
2
3      private Element element;
4      private String attribute;
5
6      public IntegerMarshaller(Element element, String attribute) {
7          this.element = element;
8          this.attribute = attribute;
9      }
10
11     public void marshal(Integer i) {
12         element.setAttribute(attribute, String.valueOf(i));
13     }
14 }
```

Listing 5.2 shows a marshaller that can marshal an integer value into an attribute of an XML element. We can see that during the initialization of the class we expect caller to provide as with an `Element` reference and name of attribute where integer values will be saved. Thus, the integer marshaller will only require an `Integer` value to be passed as an argument for the

`marshal` method so the marshaling process can be completed. The actual marshaling is done using the `setAttribute` method on the `Element` instance with the name of attribute and the `Integer` value as arguments for the method.

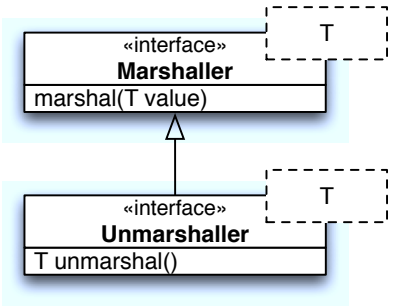


Figure 5.7: Unmarshaller

Figure 5.7 shows a UML class diagram that defines the unmarshaller interface in the COPAL middleware. It is also a generic interface where `T` is type of unmarshaled value, and same as the marshaller interface, it does not specify any reference to particular format. The concrete classes that implement the unmarshaller interface have to define the `unmarshal` method that returns the unmarshaled value of type `T`. You should also notice that the unmarshaller interface extends the marshaller interface as unmarshaling should always come in pair with marshaling.

Listing 5.3: Integer Unmarshaller

```
1  public class IntegerUnmarshaller extends IntegerMarshaller implements
   Unmarshaller<Integer> {
2
3      private Element element;
4      private String attribute;
5
6      public IntegerUnmarshaller(Element element, String attribute) {
7          super(element, attribute);
8          this.element = element;
9          this.attribute = attribute;
10     }
11
12     public Integer unmarshal() {
13         return Integer.valueOf(element.getAttribute(attribute));
14     }
15 }
```

Listing 5.3 shows an implementation of an unmarshaller that can unmarshal an `Integer` value that was marshaled by the integer marshaller from Listing 5.2. It also requires caller to provide as with an `Element` reference and name of attribute where integer values will be retrieved.

Thus, the integer unmarshaller does not require any other argument to be passed as an argument for the `unmarshal` method. The actual unmarshaling is done using the `getAttribute` method on the `Element` instance with the name of attribute as argument for the method.

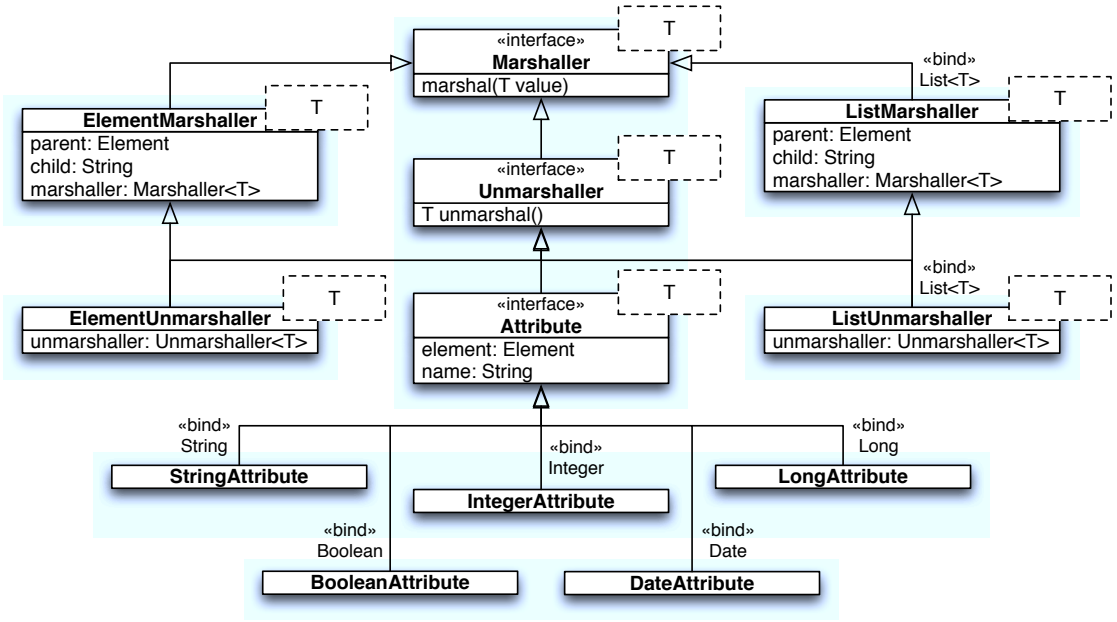


Figure 5.8: Marshallers and Unmarshallers

As we mentioned before, the COPAL middleware uses this mechanism to be able to store and retrieve any COPAL component to and from an XML DOM tree. Hence, we provide a multitude of primitive marshallers and unmarshallers that can store and retrieve values into and from XML elements and their attributes. Figure 5.8 shows a UML class diagram of complete hierarchy of primitive XML marshallers and unmarshallers in the COPAL middleware. First, we have a generic `Attribute` abstract unmarshaller that can retrieve and store a value in an attribute of an element. There are five concrete implementations of the `Attribute` abstract class: string, integer, long, Boolean, and date. Each one can store a specific type of value, `String`, `Integer`, `Long`, `Boolean`, and `Date` respectively, in the attribute of an element. Next, we have the element marshaller and unmarshaller. This class requires a `parent` element, name of child element, and actual marshaller or unmarshaller that will be used to retrieve or store a value of type `T`. The marshaling process is implemented like this: the marshaller will first look if there is already a child element in the parent element that has same name as specified name of child element, and if there is no such element, it will create one. Then, it will invoke the specified marshaller on the child element using the value passed as argument to the `marshal` method. The unmarshaling process is identical except that the unmarshaller will be invoked on the child element to retrieve the marshaled value. Finally, the list marshaller and unmarshaller is almost analogous to the element marshaller and unmarshaller except that it stores or retrieves a `List` of type `T` values to and from child elements of the parent element.

The idea behind the primitive XML marshallers and unmarshallers is that they can be combined using the *Composite* and *Builder* design pattern to create more specific implementations that can marshal and unmarshal a COPAL component to and from an XML element. The composite design pattern is used when we need to describe a group of object that should be treated in the same way as single instance of an object [39]. The intent of a composite class is to compose objects into a tree-like structure to represent a part-whole hierarchy. The composite design pattern fits nicely with our model of marshaling, as we have to combine multiple primitive unmarshallers to create a more specific implementation that can store and retrieve a COPAL component to and from an XML element. The builder design pattern is used to delay the construction of a complex object until all information that is required to instantiate the object is available [39]. In design of marshaller and unmarshaller interface, we deliberately didn't specify any data storage class in the argument list of the marshal and unmarshal methods to make them agnostic of the data format. Thus, we have to provide the data storage during the instantiation of a concrete marshaller. This creates a problem when we want to compose a sub-marshaller with an element (or list) marshaller, because the underlying child element, which will be used with the sub-marshaller, might not be present in the parent element and the element marshaller will have to create it before marshaling can proceed. Thus, the builder design pattern can be used to delay instantiation of the sub-marshaller until the child element has been created.

Listing 5.4: Processor Marshaller

```
1  public class ProcessorMarshaller implements Marshaller<Processor> {
2
3      private Unmarshaller<String> name;
4      private Unmarshaller<List<ProcessorAction> > actions;
5
6      public ProcessorMarshaller(Element element) {
7          this.name = new StringAttribute(element, ”name”);
8          this.actions = new ListUnmarshaller(element, ”Actions”, new
           ElementUnmarshaller.Builder(”Action”, new
           ProcessorActionUnmarshaller.Builder()));
9      }
10
11     public void marshal(Processor p) {
12         name.marshal(p.getName());
13         actions.marshal(p.getActions());
14     }
15 }
```

Listing 5.4 shows the implementation of an XML marshaller for the `Processor` class (see Figure 4.12). This marshaller uses a string attribute unmarshaller and a composite of list, element, and processor action unmarshallers to respectively store `name` and `actions` properties of a `Processor` instance. In the processor marshaller, we can also see the usage of builders in composition of marshaller for the actions property. Because ”Actions” and ”Action” child

elements might not be present in the specified element, the list and element marshallers will have to create these elements before processor action unmarshaller can be used. Finally, if we would use this marshaller to marshal into a `Processor` XML element a temperature calculator processor that can convert a Celsius context event into Fahrenheit and Kelvin context events, we would get the XML element in Listing 5.5 as the result of marshaling.

Listing 5.5: Example Processor XML Element

```
1 <Processor name='TemperatureCalculator'>
2   <Actions>
3     <Action name='ToFahrenheit' input='Celsius' output='Fahrenheit' />
4     <Action name='ToKelvin' input='Celsius' output='Kelvin' />
5   </Actions>
6 </Processor>
```

REST Interface

The COPAL REST interface is used as the transportation mechanism to transfer COPAL components between distributed COPAL instances. It exposes context type, publisher, and processor registries and query factory using JAX-RS annotations together with Apache CXF. Furthermore, the same REST interface can be used to allow applications to remotely interact with the middleware. It uses the XML format as a representation for all COPAL resources.

The COPAL middleware in most situations uses the standard request-replay mechanism for invocation of REST services. This mechanism is suited for many tasks like registering context types, creating publishers, publishing context events and so worth, but there are two cases which require different approach: (1) delivering context event to a remote listener, and (2) processing context event with a remote processor. Both of these cases requires the COPAL middleware to deliver context event to some remote service. Hence, the COPAL middleware requires the remote service to implement a REST endpoint and specify its URL during the registration process. If remote service is registered as a listener, a context event will be delivered in the payload of the HTTP `POST` method on specified URL. Analogously, if remote service is registered as a processor, a context event will also be delivered in the payload of the HTTP `POST` method on specified URL and the COPAL middleware will use the response of the `POST` invocation as the result of processing.

Listing 5.6 shows how to define a JAX-RS interface with which the context type registry (see Figure 4.2) is exposed as a RESTful service. The `@GET`, `@POST`, `@PUT`, and `@DELETE` annotations define which HTTP method will invoke which Java method. The `@Path` annotations define the path part of the URL, and using a `{...}` construct in the path together with the `@PathParam` annotation, we can specify some parts of the path to be passed as arguments to the method. For example, in our case, we want to extract a name of a context type from the URL in the `get` method.

Furthermore, the COPAL middleware provides a library that can be used on the client side to invoke the COPAL REST interface. Listing 5.7 shows, how we can use this library to retrieve

Listing 5.6: REST Interface for Context Type Registry Service

```
1  @Path("/")
2  public interface ContextTypeService {
3
4      @GET
5      ContextType[] getAll();
6
7      @PUT
8      void register(ContextType type);
9
10     @GET
11     @Path("name")
12     ContextType get(@PathParam("name") String name);
13
14     @DELETE
15     @Path("name")
16     void unregister(@PathParam("name") String name);
17 }
```

Listing 5.7: Example Usage of COPAL REST Client

```
1 URL address = ... ;
2 RemoteCOPAL copal = new RemoteCOPAL(address);
3 ContextType temperature = copal.getEventType("temperature");
```

a temperature context type from a remote COPAL instance at some URL. First, we have to create an instance of the proxy class, called `RemoteCOPAL`, which can be used to communicate with the remote COPAL instance. The proxy requires a URL address where the remote COPAL instance can be found. Afterwards, we can use the proxy to invoke the `getEventType` method, which will invoke the remote context type registry and receive the temperature context type. Behind the scenes, the proxy will use information found in the annotations of the `ContextTypeService` class to do correct HTTP method on correct path. In this example, it will do `GET` on the `/events/temperature` path.

5.4 Modules

The COPAL middleware uses an OSGi framework to decouple itself into different modules. This structure allows parts of the COPAL middleware to be implemented independently from other modules using different technologies when different requirements are needed. For example, we can implement registries to store their components transiently in the memory, persistently in a database, or remotely using the COPAL REST interface. Additionally, all services in the COPAL middleware are also registered as OSGi services to provide a way to dynamically find

a COPAL service during runtime. For example, publishers require the publishing service to be able to publish context events. Thus, it uses the interface that defines the publishing service (see Figure 4.7) to find the actual implementation, which can be in different COPAL module or even change during runtime.

The COPAL middleware is separated into eight modules — three essential modules and five extension modules — that allow us to deploy a COPAL instance that only provides functionality that is required in a particular use-case.

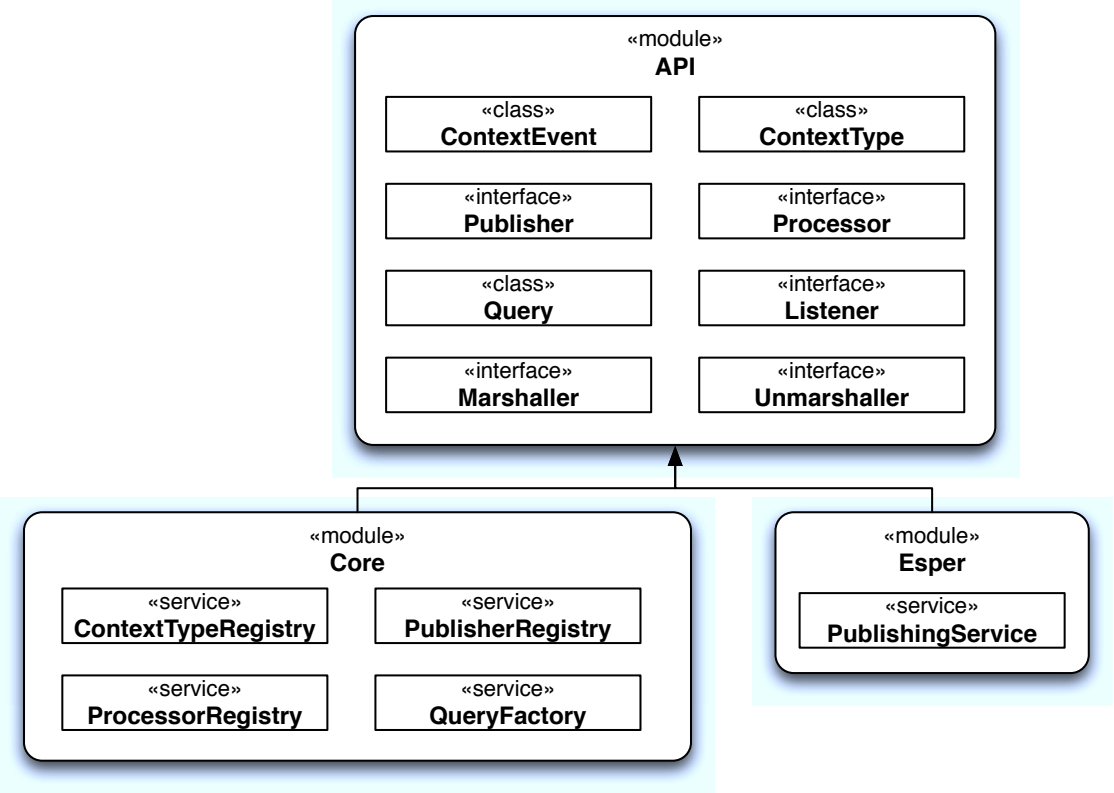


Figure 5.9: Essential COPAL Modules

Three essential modules (Figure 5.9) in the COPAL middleware are: API, core, and Esper integration. They are mandatory in every deployment of the COPAL middleware as they provide us with basic functionality like registering context types, publishing context events, and creating queries.

- *API.* The API module creates the core domain model that all other modules use and depend on. The task of the API module is to decouple all other modules from each other by containing all classes that are reused between the modules. First, the API module holds all interfaces and classes that define component in the COPAL middleware as discussed in the previous chapter (Chapter 4). Additionally, this modules also provides: (1) implemen-

tation of the observer pattern that is used throughout the COPAL middleware to provide mechanism to observe changes in mutable classes like registries, (2) the marshaling and unmarshaling mechanism to transform COPAL components to and from an XML DOM tree, and (3) helper classes for the OSGi framework to automatically find and bind to registered COPAL services.

- *Core*. The core module implements context type, publisher, and processor registries and query factory. It registers them with the OSGi framework as OSGi services, as they need to be used by other modules. Thus, interfaces for these implementations are defined in the API module to decouple the actual implementation of registries and query factory from their clients. This separation allows the core module to be implemented using different requirements like transient storage versus persistent, local storage versus remote, and so forth. The default version of this modules stores context types, publishers, processors and queries only locally in-memory and does not persist them between application executions.

- *Esper integration*. The Esper integration module is a mediator between the Esper framework and the COPAL middleware. It provides us with transportation of context events from publishers to processors and listeners. First, it observes the context type registry and query factory to respectively register new context types with Esper and to create Esper statements from COPAL queries. Second, it provides an implementation of the publishing service (see Figure 4.7) that allows us to publish new context events from publishers using the Esper framework. The publishing service is registered as an OSGi service and the interface is defined in the API module to decouple clients of the publishing service from the actual implementation.

Five extension modules (Figure 5.10) in the COPAL middleware are: REST API, REST server, REST Client, distributed API, and distributed REST. They are optional in deployments of the COPAL middleware as they provide us with additional functionality like using the COPAL middleware as a web service or creating a distributed hierarchy of COPAL instances that share environmental information.

- *REST API*. The REST API module defines the REST interface for the COPAL middleware and provides implementations of JAX-RS message providers and consumers that use COPAL marshaling and unmarshaling mechanism. The task of the REST API module is to decouple the REST clients from the REST server implementation.

- *REST Client*. The REST client module defines the remote COPAL proxy classes that can be used to invoke REST operations on a remote instance of COPAL. It uses REST interface defined in the REST API module together with the JAX-RS to create a seamless client library to be used by COPAL REST clients.

- *REST Server*. The REST server module implements REST interface that is defined in the REST API modules; i.e. it is a mediator between the REST interface and COPAL services. It uses an OSGi's built-in web server and Apache CXF library to deploy RESTful services.

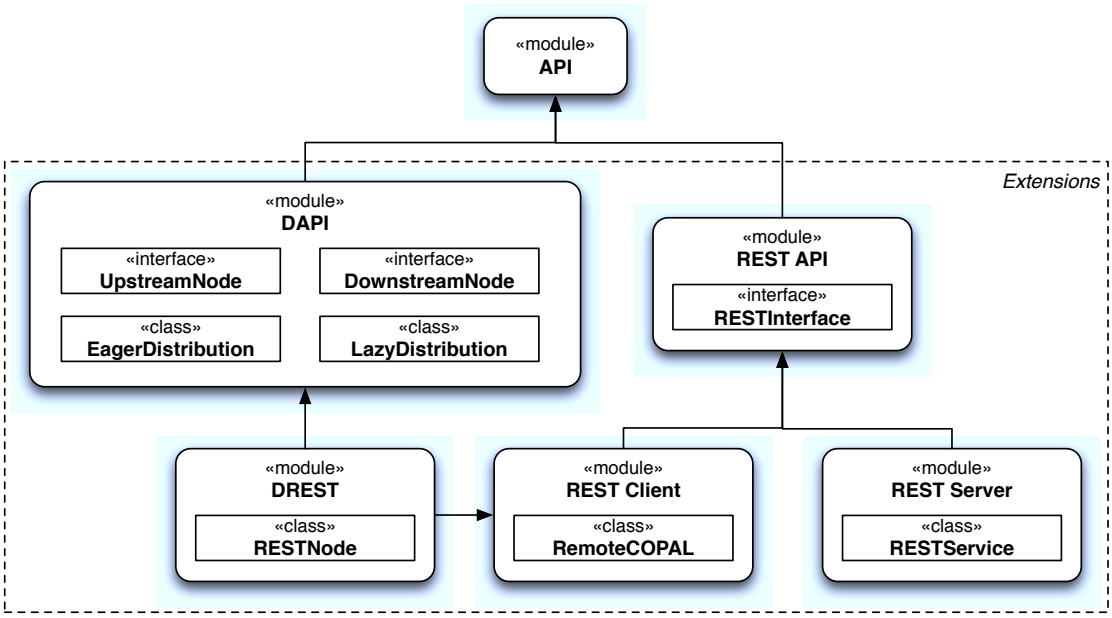


Figure 5.10: Extensions COPAL Modules

- *Distributed API (DAPI).* The DAPI module defines interfaces for upstream and downstream nodes in the distributed hierarchical deployment of COPAL instances (see Section 4.3). These interfaces define proxy classes that should hide the actual transportation layer for distributing information about the environment between the nodes. Furthermore, they allow usage of both eager and lazy mechanisms to distribute the environmental information in a transportation-agnostic way.

- *Distributed REST (DREST).* The DREST module implements the proxy interfaces for downstream and upstream nodes that are defined in the DAPI module. It uses the REST modules to define the transportation layer to transfer COPAL components and to share the environmental information between distributed COPAL nodes.

5.5 Deployment

Because each module in the COPAL middleware is a valid OSGi bundle, we can create different deployment options depending on which bundle is installed and started in the OSGi framework. In the COPAL middleware, we use Equinox[4] that is an OSGi-compliant container used by the Eclipse IDE[5] (Integrated Development Environment) for its plug-in runtime. We separate three deployment options: local-only, client/server, and distributed.

[4]http://www.eclipse.org/equinox/
[5]http://www.eclipse.org/

Local-only

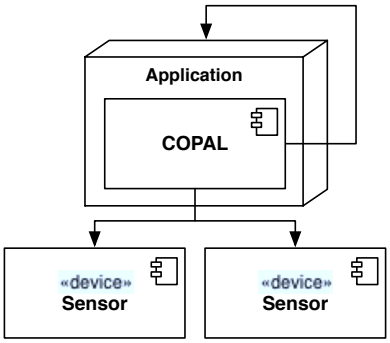


Figure 5.11: Local-only Deployment

The local-only deployment (Figure 5.11) only requires the API, core, and Esper integration modules to be deployed. In this scenario, the COPAL middleware is bundled together with context-aware application and the application does all necessary work to gather, process and react to context changes. Thus, format of context events, publishers, processors, and listeners are specific to the application and the application has to be implemented using the Java programming language and bundled with the COPAL middleware. This deployment does not support any type of remote communication out-of-the-box and context events are passed in-memory only.

Client/Server

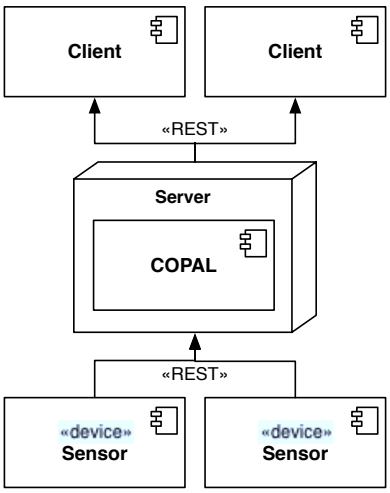


Figure 5.12: Client/Server Deployment

The client/server deployment (Figure 5.12) requires all modules from the local-only deployment to be deployed together with the REST API and REST server modules. In this scenario,

the COPAL middleware is a server and remote publishers, processors, and listeners are clients. The COPAL clients communicate with the COPAL server using its REST interface and can be implemented in any programming language that supports HTTP invocations. This scenario requires developers to define a common format for context events because the COPAL server may be shared between multiple context-aware applications that can have different use-cases and requirements from the context events.

Distributed

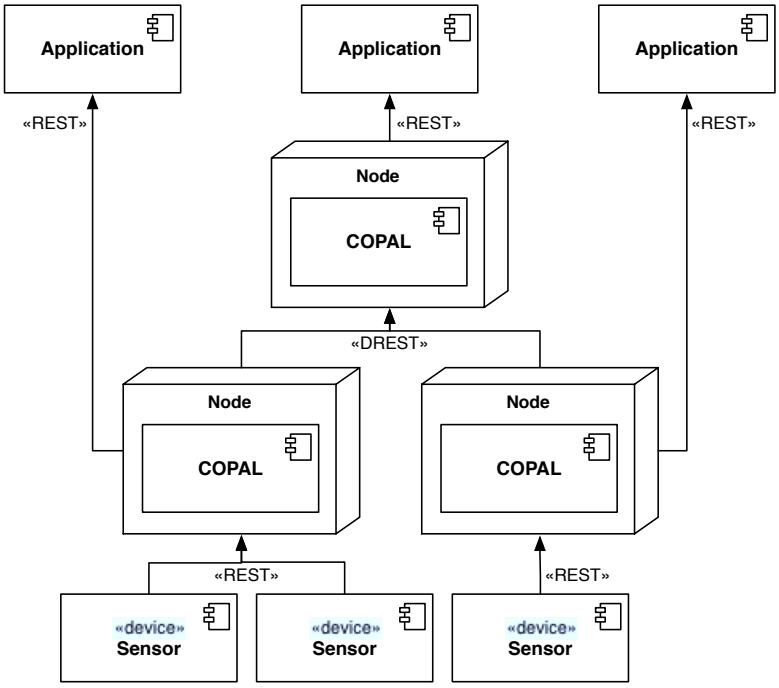


Figure 5.13: Distributed Deployment

The distributed deployment (Figure 5.13) requires all modules from the client/server deployment to be deployed together with the DAPI and DREST modules. In this scenario, we have multiple instances of COPAL middleware deployed in a hierarchical tree-like structure (see Section 4.3) where each instance can have many downstream COPAL nodes and only one upstream node. Publishers are deployed locally on leaf nodes (i.e. most-downstream nodes) or communicate with them over their REST interface. Processors and listeners can be deployed on any node in the hierarchy depending on how much information about the environment they require. This scenario also requires that developers define a common format for context events because multiple context-aware applications may use any of the COPAL nodes.

CHAPTER 6

Evaluation

This chapter presents two evaluations of the COPAL middleware. First, we investigate in Section 6.1 the performance of the COPAL middleware. Then, Section 6.2 demonstrates the usage of the COPAL middleware in the SM4ALL project.

6.1 Performance

For the performance evaluation, we have decided on two separate tests:

- The *latency* test was designed to measure the responsiveness of the COPAL middleware. We measured it as a time delay between publishing a context event and receiving the event in a listener including any intermediate processing steps.
- The *throughput* test was designed to measure the scalability of the COPAL middleware. We measured it as number of context events per second that the COPAL middleware could handle under peak workload.

Furthermore, as we have seen in Section 5.5, the COPAL middleware can be used in three different ways: local-only, client/server, and distributed. Accordingly, we have tested each deployment option separately for both the responsiveness and scalability.

Setup

For the evaluation purposes, we have used two test machines, a personal computer (Machine A) and a dedicated server (Machine B), that were connected in a 100 Mbit LAN. Table 6.1 shows the system profiles of both machines. Because Machine B is more powerful and was at the time only used for the evaluation purposes, most CPU intensive tasks were executed on it and Machine A was used to generate the workload for Machine B.

For each deployment option, we have developed a separate test setup. In each setup, we have used one or more concurrent publishers, one or more processors, and only one listener that

		Machine 1	Machine 2
Operating System	Type	Mac OS	Ubuntu
	Version	10.6.7	8.04 LTS
	Kernel	Darwin 10.7.0 (32bit)	Linux 2.6.32 (64bit)
Java	Runtime	Java SE 1.6.0_26	OpenJDK 1.6.0_20
	Virtual Machine	Java HotSpot (64bit Server)	OpenJDK (64bit Server)
Processor	Type	Intel Core 2 Duo	Intel Xeon
	Cores	2	4
	Core Speed	2.4 Ghz	3.2 GHz
Main Memory	Type	DDR3	DDR2
	Modules	2	6
	Module Size	2 Gb	2 Gb
	Module Speed	1067 MHz	400 MHz

Table 6.1: Test Machines

was recording the performance metrics. In the local-only setup, we ran one COPAL instance with publishers, processors, and listeners on Machine B. In the client/server setup, a COPAL instance with processors and listeners was running on Machine B and stand-alone publishers were running on Machine A. The publishers used the REST interface to publish context events. In the distributed setup, Machine A was running downstream COPAL instance with publishers and Machine B was running the upstream COPAL instance with processors and listeners. The transfer of context events between the machines was handled using the DREST interface. Table 6.2 summarizes characteristic of each setup.

	Local-only	Client/Server	Distributed
COPAL	Machine B	Machine B	Both
Publishers	Machine B	Machine A	Machine A
Processors	Machine B	Machine B	Machine B
Listener	Machine B	Machine B	Machine B
Communication	Memory	REST	DREST

Table 6.2: Test Setups

Latency

The latency test was separated into six test runs. In each consecutive test run, we increased, from 0 to 5, the number of processing actions that were required in context events. Each test run consisted of 5000 context events that were published in consecutive order.

The primary measurement that we were interested in was latency in milliseconds between publishing and receiving a context event with respect to how many processing actions were required. This measurement can give us the estimation of how responsive the COPAL middleware is.

Furthermore, we additionally measured the average time in milliseconds to publish, process, and query a context event. Publishing time was measured from when publisher first created a context event up to time when the event was handed over to the COPAL middleware. The processing time was measured from end of publishing up to when last processor finished its processing of the event. Finally, querying time was measured from end of processing up to when the event was delivered to the listener.

The results of the test runs are shown in Table 6.3. From the test results we can make these observations:

- Publishing of context events increases with each deployment option: $0.02ms$ for local-only, $2.29ms$ for client/server, and $4.983ms$ for distributed setup. The difference in times is due to the fact that in the client/server and distributed setups, a context event has to be transferred over the network and parsed by the COPAL middleware before it is published. Furthermore, in the distributed setup, a context event has to be processed in the downstream node before it is published in the upstream node, which furthermore increases the publishing time.

- Processing of context events increases linearly with respect to number of action that need to be processed. Average time to process a single action was $2.923ms$. Time needed to process a single action does not significantly very between the test setups because in each setup processing is handled locally by the COPAL middleware.

- Querying of context was almost identical between the test setups ($1.851ms$), because, as in processing, querying is handled locally by the COPAL middleware.

- Overall latency increases linearly with respect to number of actions in each setup due to longer processing time. Furthermore, overall latency in the client/server setup is in average $3.625ms$ slower than in the local-only setup due to longer publishing time. Analogously, overall latency in the distributed setup is in average $6.395ms$ slower than in the local-only setup and $2.77ms$ slower than in the client/server setup.

Throughput

The throughput test was also separated into six test runs. In each consecutive test run, we increased, from 1 to 6, the number of concurrent publishers in the COPAL middleware that were competing between each other to publish context events. Each test run consisted of 5000 context events that were published in parallel by all publishers. Context events did not require any processing.

The measurement that we were interested in was throughput that was measured as average number of events that were published in a second (eps for short). This measurement gave us the estimation of how scalable the COPAL middleware is with respect to number of parallel

		Local-only	Client/Server	Distributed
0 actions	Publishing (ms)	0.018	2.057	4.231
	Querying (ms)	1.497	1.864	1.452
	$\sum$	1.515	3.921	5.683
	σ	1.507	0.858	1.292
1 action	Publishing (ms)	0.02	2.205	4.751
	Processing (ms)	2.468	3.064	3.088
	Querying (ms)	1.566	1.939	1.437
	$\sum$	4.054	7.208	9.276
	σ	5.911	1.478	0.958
2 actions	Publishing (ms)	0.018	2.357	5.025
	Processing (ms)	4.953	5.863	6.056
	Querying (ms)	1.646	1.892	2.028
	$\sum$	6.617	10.112	13.109
	σ	1.647	0.844	1.84
3 actions	Publishing (ms)	0.02	2.298	5.188
	Processing (ms)	7.655	9.039	9.465
	Querying (ms)	1.759	2.021	1.879
	$\sum$	9.434	13.358	16.532
	σ	1.967	1.341	1.764
4 actions	Publishing (ms)	0.021	2.355	5.357
	Processing (ms)	10.7	12.379	12.901
	Querying (ms)	1.828	2.05	2.043
	$\sum$	12.549	16.784	20.301
	σ	5.134	1.821	1.265
5 actions	Publishing (ms)	0.022	2.541	5.346
	Processing (ms)	13.868	15.737	15.751
	Querying (ms)	1.945	2.092	2.375
	$\sum$	15.835	20.37	23.472
	σ	2.963	1.45	1.786

Table 6.3: Latency Test Results

publishers. Furthermore, this test can also be considered as a stress test to see the behavior of the COPAL middleware under peak workload, because publishers are competing with each other to publish context events.

The results of the test runs are shown in Table 6.4. From the test results, we can see that in the local-only and client/server setups the highest throughput was recorded when four publishers were used: $1154.76eps$ for local-only and $498.1eps$ for client/server. This number of publishers corresponds to number of cores on Machine B. In that situation, each thread that was handling context events was able to execute on separate core on Machine B. Thus, they were able to publish highest number of events per second because they could work in parallel without

		Local-only	Client/Server	Distributed
1 publisher	Throughput (eps)	680.57	258.14	143.46
	σ	4.419	4.18	0.86
2 publishers	Throughput (eps)	1040.56	428.43	239.14
	σ	7.284	2.6	3.214
3 publishers	Throughput (eps)	1111.38	489.98	316.47
	σ	4.888	3.42	1.767
4 publishers	Throughput (eps)	1154.76	498.1	367.02
	σ	6.358	1.93	1.837
5 publishers	Throughput (eps)	1131.74	483.86	389.61
	σ	4.406	0.92	1.4
6 publishers	Throughput (eps)	1127.46	485.3	405.7
	σ	9.672	1.97	1.67

Table 6.4: Throughput Test Results

competing with each other for processor time. When fewer publishers were used, the processor was not utilized to its fullest power. When more publishers were used, the threads started to compete for processor time and throughput decreased slightly because of scheduling required to leverage utilization of cores between threads. Interestingly, in the distributed setup, the highest throughput was with six publishers ($405.697eps$). This number corresponds to sum of cores on Machine A and Machine B. In that situation, both machines could process context events in parallel, which corresponds to six threads that handle context events in total. This shows that a distributed network of COPAL instances can increase its throughput gradually by adding more machines into the network.

Furthermore, we carried out an additional test that wanted to measure the influence of processing on throughput. We separated the test into six test runs where we increased the number of processing actions that were required in context events from 0 to 5. Each test run consisted of 5000 context events that were published in parallel by four publishers for local-only and client/server setups and six for the distributed setup. We chose four and six publishers as they provided the highest performance in the previous throughput test. This measurement gave us the estimation of how scalable the COPAL middleware is with respect to number of processing actions.

The results of the test runs are shown in Table 6.5. In this test, the worst scalability with respect to number of processing actions had the local-only setup, for which throughput decreased tenfold from 0 to 5 processor actions. For the client/server setup, the throughput decreased by the factor of 0.7 for each processor action. In the distributed setup, the throughput from 0 to 1 processor action only decreased by the factor of 0.9, because Machine A had enough time to handle a context event while Machine B was processing other context events. Afterwards, the processing of actions became a bottleneck on Machine B and Machine A had to wait longer periods of time until Machine B could receive a context event. Thus, the whole system started to behave like the client/server setup and throughput started to decrease by the same factor of 0.7.

		Local-only	Client/Server	Distributed
0 actions	Throughput (eps)	1154.76	498.1	405.69
	σ	6.358	1.931	1.202
1 action	Throughput (eps)	428.6	269.35	364.61
	σ	1.936	3.1	3.117
2 actions	Throughput (eps)	258.33	181.13	225.81
	σ	0.539	1.477	1.517
3 actions	Throughput (eps)	182.56	135.37	158.19
	σ	0.756	1.28	0.608
4 actions	Throughput (eps)	139.25	107.99	120.65
	σ	0.303	0.281	0.146
5 actions	Throughput (eps)	111.9	88.95	96.35
	σ	0.248	0.247	1.143

Table 6.5: Throughput with Processing Test Results

This test shows the importance of distributing processor over the whole distributed network, because it can lead to throughput that is much less influenced by the processing of context events.

6.2 SM4ALL Deployment

The COPAL middleware is part of the embedded and pervasive platform for smart houses developed in the SM4ALL project. The platform's aim is to provide support for inter-working of smart embedded devices in immersive and person-centric environments. The platform is applied for the scenario of private apartments/homes/buildings in presence of users with different abilities and needs (e.g. children, adults, seniors, and disabled people). Users of the platform can interact with services provided by different domotic devices, appliances and sensors through basic and advanced user interfaces like computers, tablets, and brain-computer interaction (BCI) devices. The project vision was to provide users with simple interfaces that allow them to select a goal among a set of possible ones. Goals are proactively offered by the SM4ALL platform on the basis of the available services and the user's current context. Once user specified a desired goal, a composition technique defines the most suitable way of coordinating the available services to satisfy the user's goal.

As part of the SM4ALL project, a prototype housing unit, known as "Casa Agevole"[1] ("Accessible House" in Italian language) has been made available. It is located at Fondazione Santa Lucia in Rome, Italy. The size of Casa Agevole is $60m^2$ and includes: entrance lounge-dining-kitchen, two bedrooms, and two bathrooms. It provides an easy and safe accommodation for

[1] http://www.progettarepertutti.org/progettazione/casa-agevole-fondazione/index.html

people with different abilities and needs. Figure 6.2 shows the floor plan of Casa Agevole with all installed devices.

Platform Architecture

The platform architecture is separated into the user interface, composition, and pervasive layers. The user interface layer provides users with ability to interact with the smart home and the platform through different kinds of user interfaces like touch screens, PDAs and BCI devices. The composition layer is in charge of providing the user layer with complex services by suitably composing already available ones. The pervasive layer constitutes of set of proxies that are software components offering services to the composition layer by "wrapping" and abstracting the real devices in the smart house that offer the functionality.

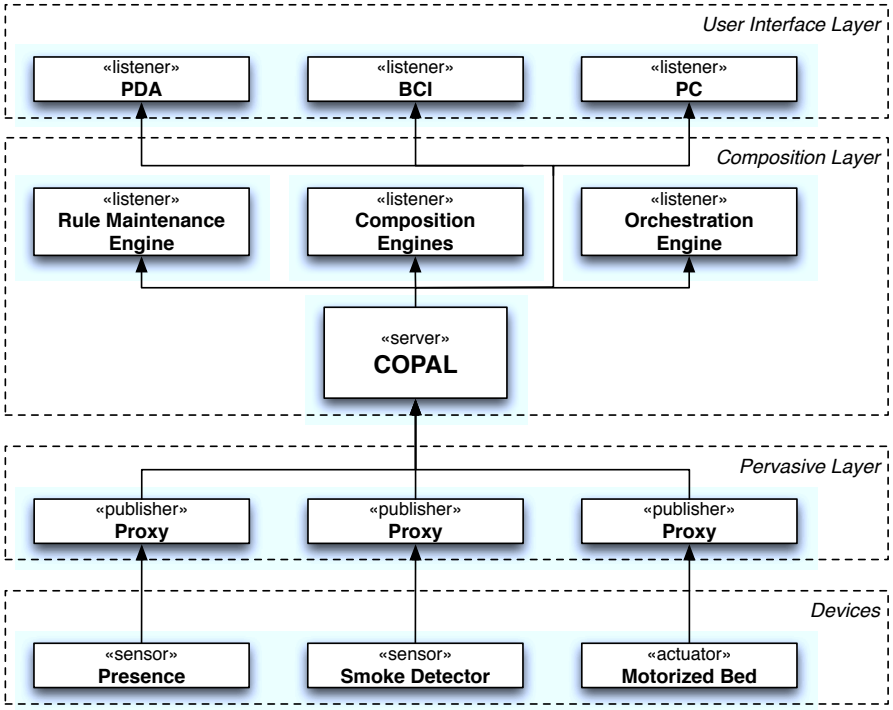


Figure 6.1: SM4ALL Platform Architecture

The COPAL middleware (Figure 6.1) is part of the composition layer in the platform. It provides context-awareness to the user interface layer, the composition engines, the orchestration engine and the rule maintenance engine by communicating with the pervasive layer. It is deployed as a context-awareness server that offers synchronous and asynchronous delivery of changes in context. Context consists of environmental variables that are "sensed" through two types of devices: actuators and sensors. Actuators are devices that are able to make some modifications in the environment like open a window, or turn on lights in a room. Sensors are

devices that are able to detect the modifications in the environment. Both actuators and sensors communicate with the COPAL middleware through proxies that are part of the pervasive layer.

Example Use Case

The localization service in Casa Agevole is provided by the Ekahau Positioning Engine. The Ekahau Positioning Engine is a stand-alone real-time location solution that uses WLAN technology to track people or objects. It runs on a server as a service and receives data from Ekahau tags over the WLAN network. Using the WLAN signal strength, the engine creates a probabilistic estimate of each tag location as an x-y coordinate. A Java SDK allows the Ekahau Positioning Engine to integrate with 3rd party solutions.

In the SM4ALL platform, the COPAL middleware is responsible for integration with the Ekahau Positioning Engine. It polls the Ekahau server periodically to determine coordinates of each monitored Ekahau tag and publishing them as location context events. After publishing, processing steps include determining which location event corresponds to which person in the house, and determining from which room in the house the location event originated from. Afterwards, the user layer can use the location information to provide the user with services that are appropriate for that specific room. For example, if user is in the living room, the user layer would provide him with services to turn on television, home-audio system, or lights in the living room.

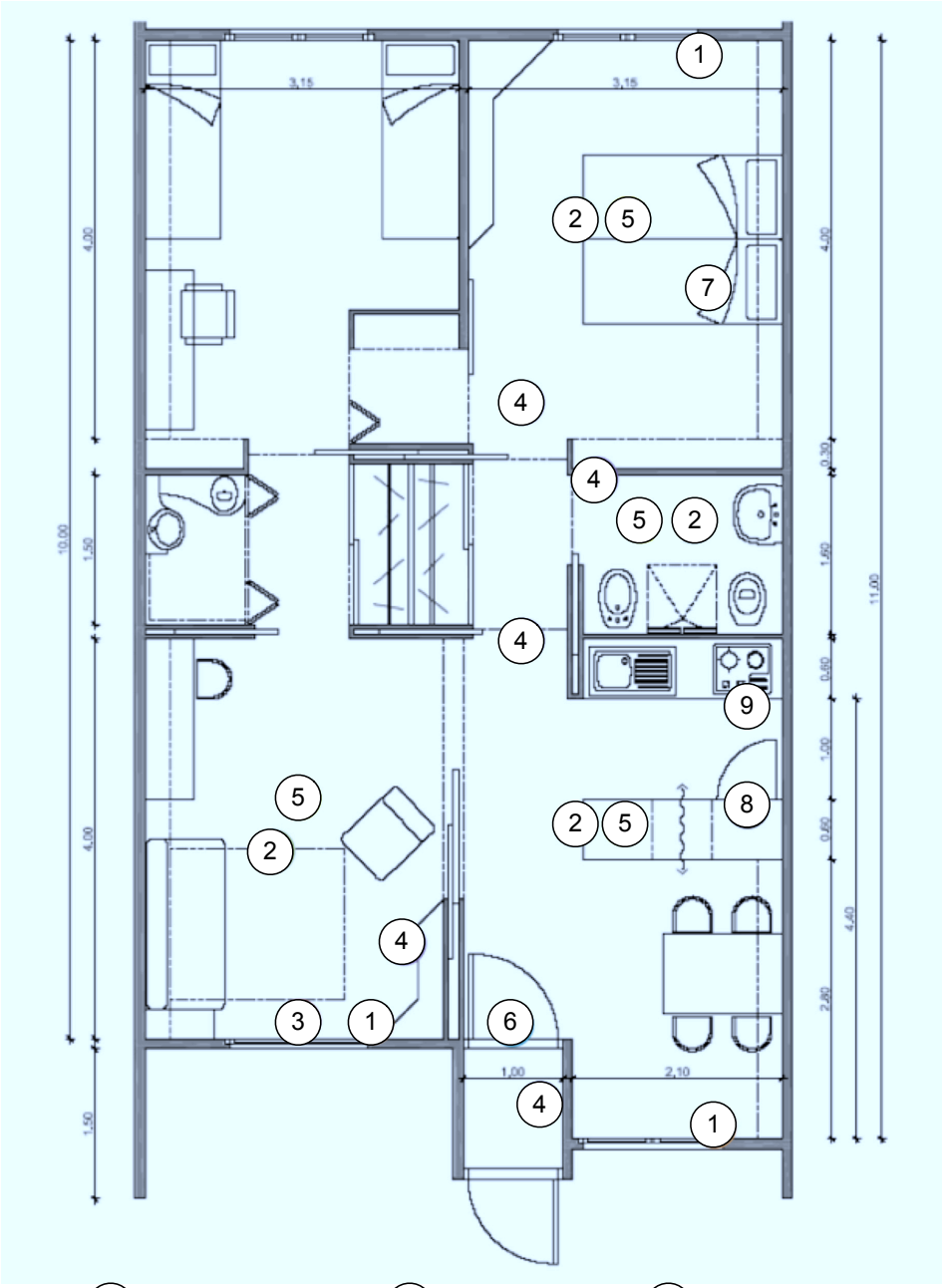


Figure 6.2: Floor Plan of Casa Agevole

CHAPTER 7

Conclusions and Future Work

The paradigm of personal computing is currently in the transition from using stationary, personal computer to having a ubiquitous presence of computing resources. In a ubiquitous system, human environment is thoroughly fitted with computing power and the whole system is interconnected into one coherent computing environment. Humans interact with many machines in the environment simultaneously without even being aware of it. Context-awareness, as a fundamental quality of ubiquitous system, uses "context to provide task-relevant information and/or services to a user." [27]

This thesis presented the design and implementation of the COPAL middleware — a context-aware service platform for context-provisioning. It introduced a context model based on context events, which provide a flexible mechanism to represent and customize context information. Based on context-events, context provisioning is separated into three tasks: publishing, processing, and reacting to context events. For each task, a suitable component was designed. The component design of the COPAL middleware allows the system to be progressively extended to support new types of context information and to build various context-aware services. Processing has been designed to dynamically bind context events with its patterns. Five processing patterns that can be used to create complex processing schemes have been explained. The COPAL middleware has been implemented as a set of loosely-coupled modules, which enabled customized deployment for various use-cases. Distributed version of the COPAL middleware has addressed how multiple COPAL instances can be deployed on multiple machines, and how to form a hierarchical network to share context information. Distributed COPAL allows the system to scale across many heterogeneous context sources and context-aware applications. The performance evaluation has shown high responsiveness and throughput of context events. The deployment evaluation has demonstrated usefulness of the COPAL middleware in a real-world system.

The COPAL middleware is already a powerful solution to build context-aware applications. However, the implementation could be enhanced with additional functionality that would make the system even more useful in wider variety of situations. Thus, we plan to further extend the COPAL middleware in several directions:

- *Predicate logic*. Query criteria can be further extended from current model that uses prepositional logic to model that supports predicate logic. One benefit of this is that it would allow creation of queries that use universal quantifier like "when all lights are on" or a Boolean function like "when it is midday" where midday is a function that is defined by a user.

- *Mobile devices*. Interesting future extension of the COPAL middleware will be to deploy it on mobile devices. Today's mobile devices are equipped with multitude of sensors and the COPAL middleware could use them as context sources to support other devices. For example, we could use GPS sensor in a mobile device to provide the user location information to a nearby personal computer. This would provide the personal computer with same level of location-awareness as the mobile device and allow users to use location-aware service on the computer.

- *Ad-hoc peer-to-peer network*. Current distributed network of COPAL instances uses a static tree-like hierarchy to connect the nodes. This model will be further extended to support a true dynamic peer-to-peer network of COPAL instances. One benefit of this model is that it will allow creation of ad-hoc networks of COPAL nodes. Ad-hoc networks would permit the COPAL nodes to automatically create groups that are merged and dissolved as the topology of the network changes. One example where ad-hoc networks could be used is to dynamically connect mobile devices to share their environmental information based on their proximity.

- *Cloud deployment*. The biggest benefit of cloud systems is its elasticity. Elasticity allows the system to automatically assign and release resources as they are needed, i.e. to automatically scale up and down. Under heavy workload, the COPAL middleware can use the elastic nature of cloud system to adaptively start new virtual machines in the cloud and redistribute the workload among them. This will provide the COPAL middleware with a self-adaption mechanism that would allow it to be even more scalable.

- *Distributing processors*. In current COPAL architecture, we can already use the distributed COPAL middleware to deploy processors on different nodes in a distributed network and thus maximize the processing capabilities. The problem with this approach, as we identified it in the evaluation section, is when one node is under heavy processing workload, the whole distributed network may suffer, because other nodes have to wait to be able send new context events. This creates a possibility for processing to be further extended with solutions like replication of processors and/or decomposition of processor actions into smaller units that can be distributed among the nodes. This would allow the COPAL middleware to distribute processors over the network to increase processing capacity.

APPENDIX A

Tutorials

During the development of the COPAL middleware, we wrote three tutorials that were used in the SM4ALL project to get the developers familiar with the COPAL middleware. This appendix contains these tutorials, which should be used as exercises for readers who want to use the COPAL middleware to develop context-aware applications.

A.1 Hello World

In this tutorial we will create one publisher and one listener. The publisher will create multiple EHello events that listener will receive and print out on the standard output. This tutorial will explain: (1) creating a publisher, (2) creating a listener, (3) defining a simple event, and (4) defining a "catch-all" query.

The preferred way of completing this tutorial is to download the skeleton project from the COPAL home page and use it to implement functionality of this tutorial.

Publisher

First, we will create a publisher for the EHello event. The easiest way is for the publisher to extend the BasePublisher class. This class implements the ContextPublisher interface, which COPAL expects for all publishers to implement and it abstracts away the boilerplate code for registration and unregistration process with COPAL and finding the required OSGi services. If you need more control during this process or the BasePublisher class does not behave as you want it to, you can always implement ContextPublisher interface and use it instead of BasePublisher.

Lets move forward with our implementation task, BasePublisher contains two abstract methods, which our publisher needs to implement:

```
protected abstract boolean start(ContextEventType type);
protected abstract void stop(ContextEventType type);
```

These two methods notify the publisher when it should respectively start and stop publishing the events of specified type. If we try to publish an event of some type for which the start method was not called, a ContextException will be thrown. The result of the start methods should return the Boolean value telling the underlying BasePublisher if the publisher has successfully started publishing events of specified type. The stop method will be invoked only when the start method returns true for specified type.

As first step we create a publisher in the publishers subproject that just print a message on the standard output when these methods were invoke:

```
import at.ac.tuwien.infosys.sm4all.copal.api.event.ContextEventType;
import at.ac.tuwien.infosys.sm4all.copal.api.publisher.BasePublisher;

public class HelloPublisher extends BasePublisher {

  public HelloPublisher() {
    super("HelloPublisher", "EHello");
  }

  @Override
  protected boolean start(final ContextEventType type) {
    System.out.println("publisher started for events: " + type.getName());
    return true;
  }

  @Override
  protected void stop(final ContextEventType type) {
    System.out.println("publisher stopped for events: " + type.getName());
  }
}
```

In the constructor, we must pass two values ”HelloPublisher” and ”EHello” to underlying BasePublisher‘s constructor. The first value tells the source ID, which will be associated with every published event from this publisher. You can think of it as a name of the publisher, because it has to be unique within the namespace of all publisher source IDs. The second value tells all types of published events. To distinguish any other names in the system from event names, we made a simple, unenforced convention that it should start with a letter ’E’. We could have easily just named the event ”Hello” and it would not clash with the source ID of the publisher. ContextPublisher can specify that it is publishing more than one type of events, but in our example we are publishing only EHello events. The start method will be invoked separately for each published type.

We also need to register the publisher with COPAL. We can do this by extending the PublishersActivator class that implements the OSGi’s BundleActivator interface and override its start method. The PublishersActivator class provides us with the mechanism to automatically register and unregister BasePublishers whenever the ContextPublisherRegistry becomes available, or unavailable respectively, as a service to underlying OSGi. Its start method is called when the bundle is activated, so we can use it to tell the activator which publisher to automatically register and unregister with COPAL.

```
import at.ac.tuwien.infosys.sm4all.copal.api.publisher.PublishersActivator;

public class Activator extends PublishersActivator {

  @Override
  protected void start() {
    System.out.println("publishers activator started");
    register(new HelloPublisher());
  }
}
```

If you used the skeleton project for this tutorial, you have to set the full path to this activator in the publishers subproject' pom.xml file as the value of the bundle.activator property. We can run the bundles with `mvn install pax:run` and you should see only this message on the standard output:

```
Publishers activator started
osgi> close
```

You should notice that our publisher's start method was never invoked. The publishers was never invoked because COPAL does not know anything about the EHello event, which our publisher claims to publish, and waits for somebody to register the EHello event before it allows the publishing of it. This is where the publishers.cfg.xml file is used, because it allows us to define the published events. The PublishersActivator class will automatically read this file and register all events that are defined in it with COPAL. In the skeleton project, an empty publishers.cfg.xml file is located in the src/main/resources directory of the publishers subproject.

Defining EHello Event

To define the EHello event, the publishers.cfg.xml file should look like this:

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Event name='EHello' />
</Context>
```

The EHello event does not carry any information within itself that COPAL should be aware of and therefore it does not need any additional configuration elements. This configuration does not tell that EHello event cannot carry any additional information, it just tells that COPAL should not care about the content of the EHello event, e.g. if the EHello event contained a Message element the COPAL system would just ignore it.

The above definition of the EHello event is actually an abbreviated version of this definition:

```
<Event name='EHello' rootElement= 'EHello' />
```

We can remove rootElement whenever the name of the event and the name of the root element are same value. If we wanted to define an EHello event, which has a different name of the root element (e.g. HelloWorld), we would need to explicitly state the name of the root element using rootElement. COPAL requires that each event has configured name of its root element, but

in special cases we can use the COPAL default behavior that name of the event and name of the root element are same value.

As we can see, the minimal definition of event is to give it a name, and only requirement is that name must be unique in the namespace of all event names. There would be no sense if there were two different EHello events, because how would listener state in which one it is interested.

If we run the bundle now, we would see the expected messages:

```
publishers activator started
publisher started for events: EHello
osgi> close
publisher stopped for events: EHello
```

From this output, we can also see that the publisher's stop method is invoked when the bundle is stopped. This is what one should expect, because when OSGi is stopping, it will invoke the stop method in the PublishersActivator class, which automatically stops all registered publishers. This makes the stop method good place for any cleanup code that publisher needs to do before it stops publishing events of specified type.

Publishing EHello Event

Now is time to actually publish EHello events. We will need to instantiate an XML Document-Builder class because all events in COPAL are published as XML documents, and then we will create a simple Timer that publishes an EHello event every two seconds in the start method and stop it in the stop method. We use the BasePublisher's publish(ContextEvent) method to publish created EHello events.

```
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

import at.ac.tuwien.infosys.sm4all.copal.api.ContextException;
import at.ac.tuwien.infosys.sm4all.copal.api.event.ContextEventType;
import at.ac.tuwien.infosys.sm4all.copal.api.event.xml.XMLEvent;
import at.ac.tuwien.infosys.sm4all.copal.api.event.xml.XMLEventType;
import at.ac.tuwien.infosys.sm4all.copal.api.publisher.BasePublisher;

public class HelloPublisher extends BasePublisher {

  private Timer timer;

  public HelloPublisher() {
    super("HelloPublisher", "EHello");
  }

  @Override
  protected boolean start(final ContextEventType type) {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
```

```
    try {
      final DocumentBuilder builder = factory.newDocumentBuilder();
      this.timer = new Timer();
      this.timer.schedule(new TimerTask() {

        @Override
        public void run() {
          final Document event = builder.newDocument();
          event.appendChild(event.createElementNS(null, "EHello"));
          try {
            publish(new XMLEvent((XMLEventType) type, getSourceID(), event));
            System.out.println("EHello event published");
          } catch (final ContextException ex) {
            System.out.println("Something went wrong");
            ex.printStackTrace();
          }
        }
      }, 0, 2000);

      System.out.println("publisher started for events: " + type.getName());
      return true;
    } catch (final ParserConfigurationException ex) {
      return false;
    }
  }

  @Override
  protected void stop(final ContextEventType type) {
    this.timer.cancel();
    System.out.println("publisher stopped for events: " + type.getName());
  }
}
```

When we run the bundle, we should see something like this on the standard output:

```
publishers activator started
publisher started for events: EHello
EHello published
EHello published
EHello published
...
```

The creation of same event over and over again can be very tedious, so we provided you with a helper method createEvent(DocumentBuilder) in the XMLEventType class. This method creates an instance of XML document with a root element that has local name set to value of rootElement from event's definition. Remember that if the rootElement is missing than it has same value as name of the event. The createEvent method plus the fact that the XML document is cloned before it is published means that we can reuse one XML document when publishing. Each invocation of publish will actually publish a cloned instance of specified XML document, therefore, any change to the XML document after it is published will not change the published event, because it is a different instance of XML document. The improved version of the start method looks like:

```
@Override
public boolean start(final ContextEventType type) {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);

  try {
    final DocumentBuilder builder = factory.newDocumentBuilder();
    final Document event = ((XMLEventType) type).createEvent(builder);

    this.timer = new Timer();
    this.timer.schedule(new TimerTask() {

      @Override
      public void run() {
        try {
          publish(new XMLEvent((XMLEventType) type, getSourceID(), event));
          System.out.println("EHello event published");
        } catch (final ContextException ex) {
          System.out.println("Something went wrong");
          ex.printStackTrace();
        }
      }
    }, 0, 2000);

    System.out.println("publisher started for events: " + type.getName());
    return true;
  } catch (final ParserConfigurationException ex) {
    return false;
  }
}
```

If we run this example we should see the same result as in previous run.

Query & Listener

Now we can focus on receiving published EHello events. We do this by creating a query that receives all EHello events and registering a listener with it. You can register more than one listener with each query, and each listener can be registered to more than one query. The listeners subproject will hold the listener.

We define queries in the listeners.cfg.xml file (similarly to how we defined EHello event in the publisher.cfg.xml file). In our case, listener wants to be notified whenever an EHello event is published, therefore, the listeners.cfg.xml file looks like:

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Query name='EHello.All' event="EHello' />
</Context>
```

When defining events, each query must have an unique name in the namespace of query names and the name of the event for which it receives notifications. The name of the event is the same name that is used in the publishers.cfg.xml to name the event. This query will be

notified whenever an EHello event is published, because it does not have any other configuration elements.

A listener can be implemented either by implementing the ContextListener interface or extending the BaseListener class. The BaseListener is very simple and only implements the getName() method from the ContextListener interface. BaseListener and the interface also require subclasses to implement the onEvent(ContextEvent) method, which is called by the query, to which the listener is registered, whenever an event is received. Our listener will just print out a message when it receives an event:

```
import at.ac.tuwien.infosys.sm4all.copal.api.event.ContextEvent;
import at.ac.tuwien.infosys.sm4all.copal.api.listener.BaseListener;

public class SimpleListener extends BaseListener {

  public SimpleListener() {
    super("SimpleListener");
  }

  @Override
  public void onEvent(final ContextEvent event) {
    System.out.println(event.getType().getName() + " received");
  }
}
```

In the constructor, we pass the name of this listener to underlying BaseListener‘s constructor. The name of the listener must be unique within each query i.e. we can have two different listeners with same name as long as they are not registered to same query.

To register this listener with the query defined in the listeners.cfg.xml file, we use the ListenersActivator that also implements the OSGi’s BundleActivator interface and override its start method. Similarly to the PublishersActivator class, the ListenersActivator class provides us with the mechanism to automatically create queries defined in the listeners.cfg.xml file and to register and unregister ContextListeners with it. Same as before, if you used the skeleton project for this tutorial, you have to again set the full path to this activator in the listeners subproject’ pom.xml file.

```
import at.ac.tuwien.infosys.sm4all.copal.api.listener.ListenersActivator;

public class Activator extends ListenersActivator {

  @Override
  protected void start() {
    System.out.println("listeners activator started");
    register("EHello.All", new SimpleListener());
  }
}
```

As you can see we use the name of the query to register our listener. If we run the listener and publisher bundles, we would see something similar to this on the standard output:

```
publishers activator started
publisher started for events: EHello
listeners activator started
EHello received
EHello event published
EHello received
EHello event published
EHello received
EHello event published
...
```

This finishes the tutorial that implements a simple Hello World program using COPAL to pass the EHello events between publisher and listener.

A.2 Advance Event & Query Configuration

In this tutorial we will create one publisher and two listeners. The publisher will publish current temperature in Celsius. The first listener will receive each temperature event and print out on standard output the values of temperature in Celsius. The second listener will only receive the temperature events that are below zero degrees Celsius. This tutorial will explain: (1) defining an event with namespace and XML Schema, and (2) defining a query with logical criteria.

The preferred way of completing this tutorial is to download the skeleton project from the COPAL home page and use it to implement functionality of this tutorial.

Publisher & Listener

First step is to define an ETemperature event and create publisher and listener that receives all ETemperature events. We start by defining the ETemeprature event in the publishers.cfg.xml file.

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Event name='ETemperature' />
</Context>
```

The publisher will simulate a temperature sensor by publishing a random temperature between -20 and 35 °C every 5 seconds. Its structure is similar to the EHello publisher we created in the Hello World tutorial:

```
public class TemperatureSensor extends BasePublisher {

  private final Random random = new Random(System.currentTimeMillis());
  private Timer timer = null;

  public TemperatureSensor() {
    super("TemperatureSensor", "ETemperature");
  }

  @Override
  protected boolean start(final ContextEventType type) {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);

    try {
      final DocumentBuilder builder = factory.newDocumentBuilder();
      final Document event = ((XMLEventType) type).createEvent(builder);

      this.timer = new Timer();
      this.timer.schedule(new TimerTask() {

        @Override
        public void run() {
          // temperature in Celsius is between -20 and 35.
          int temperature = TemperatureSensor.this.random.nextInt(56) - 20;
          event.getDocumentElement()
```

```
                .setAttribute("celsius", String.valueOf(temperature));

            try {
              publish(new XMLEvent((XMLEventType) type, getSourceID(), event));
            } catch (final ContextException ex) {
              System.out.println("Something went wrong");
              ex.printStackTrace();
            }
          }
        }, 0, 5000);

        return true;
        } catch (final ParserConfigurationException ex) {
          return false;
        }
    }

    @Override
    protected void stop(final ContextEventType eventType) {
      this.timer.cancel();
    }
}
```

When the publisher is started, it creates an ETemperature XML document using the createEvent(DocumentBuilder) method in the XMLEventType class. Remember that a XML document is cloned before publishing, therefore we can reuse the XML document that we created at the beginning and just update the celsius attribute whenever we want to publish new temperature.

The OSGi Activator for the bundle looks identical to the publisher's Activator in the Hello World tutorial:

```
public class Activator extends PublishersActivator {

  @Override
  protected void start() {
    register(new TemperatureSensor());
  }
}
```

The second bundle will hold a listener that will receive the published ETemperature events. The listener just prints the value of the celsius attribute on the standard output:

```
import at.ac.tuwien.infosys.sm4all.copal.api.listener.Events;
import at.ac.tuwien.infosys.sm4all.copal.api.listener.Event;
import at.ac.tuwien.infosys.sm4all.copal.api.util.Name;

@Name("TemperatureListener")
public class TemperatureListener {

  @Event(type = "ETemperature")
  public void onTemperature(final XMLEvent event) {
    final Document document = event.getDocument();
```

```
    final String temperature = document.getDocumentElement()
                                       .getAttribute("celsius");

    System.out.println("New temperature");
    System.out.println("Celsius: " + temperature);
  }
}
```

The difference between EHello listener developed in the Hello World tutorial and this one is that we not extending a BaseListener and instead are using the annotations to define the listener. First the class must be annotated with the @Name annotation that specifies the name of the listener and it must have at least one method annotated with the @Event or @Events annotation. The @Event annotation can optionally specify which type of event the method should be invoked with and when type is not used than the method is invoked with any event type. The @Events annotation is just a composition annotation when you need to annotate the method with multiple @Event annotations and it is not used in this example.
The rules for method to be annotated with @Event or @Events annotation are:

- The method must be public.
- The method must have exactly one parameter.
- The parameter must be ContextEvent or a subclass (in our example it is the XMLEvent).

The rules, for which methods are invoked when an event is received, are:

- The event must have class equal to or a subclass of the method parameter.
- If the method is annotated with the @Event annotation that specifies a type than the event must be of that type.
- If the method is annotated with the @Event annotation that does not specify a type than the event can be of any type.
- If the method is annotated with the @Events annotation than the event must match any @Event annotation.
- All methods for which above rules hold will be invoked with the event.

Example of a method that catches all events (because we don't specify any type in the @Event annotation and the parameter is ContextEvent that is the superclass of all events) would be:

```
@Event
public void onAnyEvent(final ContextEvent event) {
  //do something
}
```

Finally we define the query for the ETemperature events in the listeners.cfg.xml file and register the listener to it using the OSGi Activator. The listener.cfg.xml file looks like:

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Query name='ETemperature.All' event='ETemperature' />
</Context>
```

And the activator should look like (do not forget also to set bundle.activator property in the listeners subproject's pom.xml file). Notice that we must wrap the TemperatureListener with the AnnotatedListener because it doesn't implement the ContextListener interface:

```
import at.ac.tuwien.infosys.sm4all.copal.api.listener.AnnotatedListener;

public class Activator extends ListenersActivator {

    @Override
  protected void start() {
    register("ETemperature.All",
             new AnnotatedListener(new TemperatureListener()));
  }
}
```

If we run the example now, we would see something similar to this on the standard output:

```
New temperature
Celsius: 28
New temperature
Celsius: 0
New temperature
Celsius: -4
...
```

Namespace

Next step is to put the ETemperature into its own namespace. The configuration of the namespace for an event is not mandatory, but having events in their own namespaces has advantage like preventing name clashes of the root elements between events.

Adding a namespace to the ETemperature event is very simple. In the publishers.cfg.xml we add a namespace attribute to the ETemperature's Event element, which specified the event's default namespace. Only consequence is that your root element must be in this namespace; the root's child element does not have to be in same namespace, but it is preferable.

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Event name='ETemperature'
         namespace='http://www.sm4all-project.eu/COPAL/Tutorial' />
</Context>
```

Only difference in the publisher is that DocumentBuilderFactory must be aware of namespaces. Here we can see the advantage of using the createEvent(DocumentBuilder) method, because it automatically will define the default namespace of the created XML document to be the event's namespace and use it when creating the root element.

```
@Override
protected boolean start(final ContextEventType type) {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);

  try {
    final DocumentBuilder builder = factory.newDocumentBuilder();
    final Document event = ((XMLEventType) type).createEvent(builder);

    ...

    return true;
  } catch (final ParserConfigurationException ex) {
    return false;
  }
}
```

There should be no change in the output if we run the example now. The advantage when we implement publishers and listeners in this way is that we can change the event's namespace in the publishers.cfg.xml and we do not have to recompile the example. We can even remove the namespace configuration and it should still work. This also holds for changing the name of the root element, because we used the createEvent(DocumentBuilder) method; it would just create the XML Document with new name for the root element. You should play with the definition of the ETemperature event, changing the name of the root element and event's namespace, and the example should always work without touching the implementation of the publisher and the listener.

Schema

The last configuration for events we can use is to define the XML Schema for the ETemperature event. First we have to create the XML Schema file that defines the ETemperature XML element and we put it into the publishers subproject src/main/resource directory and name it ETemperature.xsd:

```
<xsi:schema
  xmlns='http://www.sm4all-project.eu/COPAL/Tutorial'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.sm4all-project.eu/COPAL/Tutorial'
  elementFormDefault='qualified'>

  <xsi:element name='ETemperature'>
    <xsi:complexType>
      <xsi:attribute name='celsius' type='xsi:decimal' use='required' />
    </xsi:complexType>
  </xsi:element>
</xsi:schema>
```

If you are familiar with the XML Schema you will see that this schema file is simple. We define that the root element has name ETemperature and it has one and only one attribute named

celsius, which is of decimal type. Root element is in its own namespace: http://www.sm4all-project.eu/COPAL/Tutorial.

In the publishers.cfg.xml we have to tell the location of this XML Schema file for the ETemperature event. We do this with Schema element, which has one of these children: URL or ClassPath. The URL is used when we have an URL location for the schema file like a web address. The ClassPath contains the name of the schema file that can be located in the bundle's classpath and can be read using standard Java class loading facilities. Both URL and ClassPath elements require a location argument, which respectively specifies URL and classpath location where to look for the XML Schema file. In our example, we have put the ETemperatude.xsd file into the publishers subproject, and we need to use the Schema with ClassPath in the publishers.cfg.xml:

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Event name='ETemperature'
         namespace='http://www.sm4all-project.eu/COPAL/Tutorial'>
    <Schema>
      <ClassPath location='ETemperature.xsd' />
    </Schema>
  </Event>
</Context>
```

If you rerun the example, you should see the same output as before.

Logical Criteria

You will probably wonder what is advantage of defining XML Schema files for our event. Basically, it helps COPAL know about the structure of your events and their types and provides the facility to create more advance queries using logical criteria to select events that listeners are interested in. In our example, we can now use the celsius attribute to only receive temperature events that are below zero degrees Celsius. This is all possible because we defined that the ETemperature event has celsius attribute that is of decimal type.

First let us create a simple listener for our minus temperature events that prints out a message on the standard output just to show that it works:

```
public class MinusTemperatureListener extends BaseListener {

  public MinusTemperatureListener() {
    super("MinusTemperatureListener");
  }

  @Override
  public void onEvent(final ContextEvent event) {
    System.out.println("Too cold!");
  }
}
```

We can now proceed with definition of the below zero degrees Celsius ETemperature query in the listeners.cfg.xml and if we register this listener with it same way we registered the TemperatureListener with the ETemperature.All query and it will work, but we will create a query

and register this listener programmatically to show that you can also define queries and register listeners dynamically inside bundles at runtime. You do not need to use the listeners.cfg.xml file at all, but you should because it hides a lot of boilerplate code as we will see now. Just for the sake of demonstration the configuration of the ETemperature.BelowZero query in the listeners.cfg.xml would look like and notice that less-than sign has to be escaped for the listeners.cfx.xml to be valid XML file:

```
<Query name='ETemperature.BelowZero'
       event='ETemperature'
       criteria='celsius &lt; 0' />
```

The programmatic way of creating queries in COPAL is to override the ListenersActivator's start(ContextQueryFactory) method and use the provided factory to create queries. Both start methods can be overridden simultaneously as we will see in our example. This start method is called when an OSGi service is registered which implements the ContextQueryFactory interface and it comes in a pair with the stop() method that is called when the ContextQueryFactory service becomes unavailable and should be used to destroy all queries that we created in the start method.

The listeners' activator now looks like:

```
public class Activator extends ListenersActivator {

  private ProcessedEventQuery belowZeroQuery;

  @Override
  protected void start() {
    register("ETemperature.All",
             new AnnotatedListener(new TemperatureListener()));
  }

  @Override
  protected void start(final ContextQueryFactory queryFactory) {
    try {
      this.belowZeroQuery = queryFactory.create("ETemperature.BelowZero",
                                                "ETemperature",
                                                "celsius < 0");
      this.belowZeroQuery.register(new MinusTemperatureListener());
    } catch (final RedefinitionOfQueryException ex) {
      System.out.println("ETemperature.BelowZero query is already defined!");
    } catch (final AlreadyRegisteredException ex) {
      System.out.println("Listener with same name is already registered");
    } catch (final QueryDestroyedException ex) {
      System.out.println("Below zero ETemperature query is destroyed");
    }
  }

  @Override
  protected void stop() {
    if (this.belowZeroQuery != null)
      try {
```

```
        this.belowZeroQuery.destroy();
      } catch (final QueryDestroyedException ex) {
        System.out.println("Below zero ETemperature query is destroyed");
      } finally {
        this.belowZeroQuery = null;
      }
  }
}
```

Now you can see why it is so much easier define queries in the listeners.cfg.xml file and use the start() method to register listener.

The first line in the try-catch block of the start method creates the query with name ETemperature.BelowZero that receives ETemperature events that have celsius attribute less than 0. The create method can throw a RedefinitionOfQueryException if there is already a query with same name that receives different events or has different logical criteria, therefore, it is meaningful to put the name of the event in the name of the query, as we have done in all queries that we have previously defined. This ensures that if we create another temperature event with different name, the queries for the second temperature event would not clash with queries for the ETemperature event. The second line registers the listener with the query. The registration can throw AlreadyRegisteredException or QueryDestroyedException. The AlreadyRegisteredException is thrown when there is a listener with same name already registered with the query, and the QueryDestroyedException is thrown when the query has been previously destroyed.

The first line in the try-catch block of the stop method destroys the query. It can throw also the QueryDestroyedException, which in this case can be ignored, because it was already our intention to destroy the query.

When we run the example, we can see that the "Too cold" message is only printed when a temperature is below zero:

```
New temperature
Celsius: 11
New temperature
Celsius: -11
Too cold
New temperature
Celsius: 32
New temperature
Celsius: -5
Too cold
...
```

This finishes the tutorial that explains the more advance configurations for the events and queries.

A.3 Processors

In this tutorial we will reuse the code for publisher and listener from the Advance Event & Query Configuration tutorial to create two processors on top of it. The first processor will add fahrenheit attribute into ETemperature events and set its value to temperature in Fahrenheit by calculating it from Celsius. The second processor will add kelvin attribute in the ETemperature event and set its value to temperature in Kelvin. This tutorial will explain: (1) defining required and optional default actions for an event, and (2) creating a processor that can handle an event action.

The preferred way of completing this tutorial is to reuse your implementation for the Advance Event & Query Configuration tutorial to implement functionality of this tutorial.

Defining Default Actions

The TemperatureSensor publishes ETemperature events in celsius and we would also like to have the published temperature in fahrenheit and kelvin. One solution is to change the publisher and calculate these values for each published ETemperature event. The better and more flexible solution is to create two processors, which will calculate Fahrenheit and Kelvin temperature for each published ETemperature event. The publisher is not aware of the processors and processors are not aware of each other; we separate the task of reading the sensor and publishing the event, and computing temperature in Fahrenheit and Kelvin from Celsius in their own implementations. Only the listener will be aware of the consequence of processor action, but not their existence, because it can now read the temperature in Celsius, Fahrenheit and Kelvin.

First task is to define the default actions for ETemperature event. We do this in the publishers.cfg.xml file:

```
<Context xmlns='http://www.sm4all-project.eu/COPAL'>
  <Event name='ETemperature'
    namespace='http://www.sm4all-project.eu/COPAL/Tutorial'>
    <Actions>
      <Action name='AddFahrenheit' required='true' />
      <Action name='AddKelvin' required='false' />
    </Actions>
    <Schema>
      <ClassPath location='ETemperature.xsd' />
    </Schema>
  </Event>
</Context>
```

Difference between the ETemperature event definition in the Advance Event & Query Configuration tutorial and this definition is in the Actions element. This element has one or more Action elements. Each Action defines one action. Each separate event instance actually carries its own actions but in the event definition we can define which actions it will each event instance carry by default. In our case we define two actions: AddFahrenheit and AddKelvin. The AddFahrenheit action is required and the AddKelvin action is optional. If an action is required or optional is set using the required attribute where true means the action is required and false that action is optional.

When we run the example we should not see any output from the TemperatureListener and instead we will get error messages telling us that there is no processor for the required AddFahrenheit action. If the event is missing a processor for the required action, that this event will be dropped by COPAL because it is considered uncompleted. To fix this error we have to create at least one processor that can handle the AddFahrenheit action.

The output of running the example should look something like this:

```
ERROR ... - Dropped event 'ETemperature'! No processor for the required
 'AddFahrenheit' action.
ERROR ... - Dropped event 'ETemperature'! No processor for the required
 'AddFahrenheit' action.
ERROR ... - Dropped event 'ETemperature'! No processor for the required
 'AddFahrenheit' action.
...
```

Before we go any further we must also extend the XML Schema file for the ETemperature event. We have to define two additional attributes: fahrenheit and kelvin. These attributes will hold the value of the temperature in fahrenheit and in kelvin and the ETemperature.xsd file should now look like:

```
<xsi:schema
  xmlns='http://www.sm4all-project.eu/COPAL/Tutorial'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.sm4all-project.eu/COPAL/Tutorial'
  elementFormDefault='qualified'>

  <xsi:element name='ETemperature'>
    <xsi:complexType>
      <xsi:attribute name='celsius' type='xsi:decimal' use='required' />
      <xsi:attribute name='fahrenheit' type='xsi:decimal' use='optional' />
      <xsi:attribute name='kelvin' type='xsi:decimal' use='optional' />
    </xsi:complexType>
  </xsi:element>
</xsi:schema>
```

Both elements must be optional, because when an ETemperature event is published it does not have either of them and the validation of the event against this schema file would fail and COPAL would just drop the event and never publish it.

Calculate Fahrenheit

A processor is a combination of publisher and listener in one. The difference from publisher is that it has an input event and can publish new events only as a result of processing an action on the input event. It can publish same event modified or not, it can publish new events, or it does not have to publish any event. If it does not publish a new event and this processor is only processor for this action than the input event will be lost. The difference between a processor and listener is that listener does not publish new events and that listener receives events for which at least all required actions are already processed.

The actions are, as we mentioned before, carried by the events. This makes possible to route each event instance differently at runtime. The event definition only tells which actions are in each event instance by default. It does not mean that the action will be executed on an event, because we can change or even remove actions at runtime for each event instance. The final thing worth mentioning is that multiple processors can be registered for same action and all registered processors will be executed if an event instance requires this action.

First step for us is to create a processor for the AddFahrenheit action. We start by creating a new processors subproject (you can copy processors from the generic skeleton project from the COPAL home page and modify its pom.xml file). The processors can extend the BaseProcessor class, implement the ContextProcessor interface, or use annotations. The simplest way to implement the AddFahrenheit processor is to use annotations and just relay the input ETemperature event unchanged to output. It does not do what is expected, but for now we just need to see how to define and register a processor with COPAL.

The code for this simple AddFahrenheit processor looks like:

```
import at.ac.tuwien.infosys.sm4all.copal.api.processor.Action;
import at.ac.tuwien.infosys.sm4all.copal.api.util.Name;

@Name("CelsiusToFahrenheitCalculator")
public class FahrenheitCalculator {

  @Action(name = "AddFahrenheit",
          input = "ETemperature",
          output = "ETemperature")
  public XMLEvent calculate(final XMLEvent event) {
    return event;
  }
}
```

First we specify the name of the processor with the @Name annotation. Same as with the annotated listeners, the processor must have at least one method annotated with the @Action or @Actions annotation.The @Action annotation specifies the name of action the processor can handle, the type of input event and optionally types of output events. In our case, this processor can process the AddFahrenheit action on an ETemperature event and the output can only be an ETemperature event. We can also specify more than one type of output event telling COPAL that output can be any of specified events (i.e. output can be one or more events of any specified output types). We can also omit the output argument, which means that processor will never publish any event. If we do not want to produce any output for some specific input, we just return null or an empty array from the process method. The @Actions annotation is just a composition annotation when you need to annotate the method with multiple @Action annotations and it is not used in this example.
The rules for method to be annotated with @Action or @Actions annotation are:

- The method must be public.
- The method must have one or two parameter.

- The required parameter must be ContextEvent or a subclass (in our example it is the XMLEvent).
- The other (optional) parameter must be of type ProcessorAction (used when the method must know which action it must do on the event).
- The method must be void or return ContextEvent (or a subclass), an array of ContextEvents (or an array of any ContextEvent's subclass), or a java.util.Collection of ContextEvents (or a collection of ContextEvent's subclass).

The rules, for which methods are invoked when an event is received, are:

- The event must have class equal to or a subclass of the method parameter.
- The name of event's current action must be equal to the name of processed action.
- The type of event must be equal to the input type.
- If the method is annotated with the @Actions annotation than the event must match any @Action annotation.
- All methods for which above rules hold will be invoked with the event.

The final step for us is to create the OSGi Activator that will register this processor with COPAL. This activator looks almost identical to the activator for publishers and only difference is that it knows how to register processor and not publishers. Notice that we must wrap the FahrenheitCalculator with the AnnotatedProcessor because it doesn't implement the ContextProcessor interface.

```
import at.ac.tuwien.infosys.sm4all.copal.api.processor.AnnotatedProcessor;
import at.ac.tuwien.infosys.sm4all.copal.api.processor.ProcessorsActivator;

public class Activator extends ProcessorsActivator {

  @Override
  protected void start() {
    register(new AnnotatedProcessor(new FahrenheitCalculator()));
  }
}
```

If we run the example, we should receive the expected output from the TemperatureListener telling us the temperature in Celsius:

```
ERROR ... - Dropped event 'ETemperature'! No processor for the required
 'AddFahrenheit' action.
New temperature
Celsius: 21
New temperature
Celsius: 16
...
```

First thing you should notice is that the TemperatureListener receives the event, even though an AddKelvin processor was never created nor registered, and this is because the AddKelvin action is optional. If COPAL finds an event that needs an action that is optional and there is no processor registered for this action, it will just pass the event unchanged to next action. In our case, there are no further actions and listeners receive the event.

The output also has one error message. The error message is because the OSGi system started the publishers before it started the processors and when the publisher started publishing there was no processor for the AddFahrenheit action. We can see that for second and third ETemperature event the processor was registered and therefore the listener was able to receive the ETemperature event. We cannot do much about this except perhaps tell the OSGi system to start the publishers after the processors using the higher start level for publishers then for the processors.

Now is also good time to extend the TemperatureListener to print out the temperature in Fahrenheit and Kelvin. The new onEvent(Document) method in the listener now looks like this:

```
@Event(type = "ETemperature")
public void onTemperature(final XMLEvent event) {
  final Document document = event.getDocument();
  final Element rootElement = document.getDocumentElement();
  final String celsius = rootElement.getAttribute("celsius");
  final String fahrenheit = rootElement.getAttribute("fahrenheit");
  final String kelvin = rootElement.getAttribute("kelvin");

  System.out.println("New temperature");
  System.out.println("Celsius: " + celsius);
  if (fahrenheit != null) {
    System.out.println("Fahrenheit: " + fahrenheit);
  }
  if (kelvin != null) {
    System.out.println("Kelvin: " + kelvin);
  }
}
```

Notice that we are printing the text content of the fahrenheit and kelvin attributes only if they are present because they are defined as optional in the ETemperature XML Schema file. If we run the example now, we should get only the temperature in Celsius, because the AddFahrenheit processor never calculated the temperature in Fahrenheit.

Only missing part is to actually calculate the temperature in Fahrenheit. For this task we first need to extract the temperature in Celsius and use the formula $\text{Fahrenheit} = \text{Celsius} \cdot \frac{9}{5} + 32$ to get the temperature in Fahrenheit. After that, we add a new attribute with name fahrenheit that has the temperature in Fahrenheit as its value.

```
@Override
@Action(name = "AddFahrenheit",
        input = "ETemperature",
        output = "ETemperature")
public XMLEvent calculate(XMLEvent event)
    throws MalformedDocumentException {
```

```
  final Document document = event.getDocument();
  final Element rootElement = document.getDocumentElement();

  BigDecimal celsius = new BigDecimal(rootElement.getAttribute("celsius"))
                           .setScale(1, RoundingMode.UNNECESSARY);
  // fahrenheit = celsius * 9 / 5 + 32
  final BigDecimal fahrenheit = celsius.multiply(BigDecimal.valueOf(9))
                                       .divide(BigDecimal.valueOf(5))
                                       .add(BigDecimal.valueOf(32));

  rootElement.setAttribute("fahrenheit", fahrenheit.toPlainString());
  return new XMLEvent(event.getType(), document);
}
```

Finally, the TemperatureListener should print the temperature in Fahrenheit beside the temperature in Celsius:

```
New temperature
Celsius: 4
Fahrenheit: 39.2
New temperature
Celsius: -8
Fahrenheit: 17.6
...
```

Calculate Kelvin

A processor should not care if the event action is required or optional. This can be easily understood: some event instances will have this action as required and some not. We already mentioned that it is single event instance that carries the information about the actions that should be processed on it. The processor should only care how to do its processing of events it receives.

The AddKelvin processor looks almost identical to the AddFahrenheit processor and only difference is in name, which action it can process and how to calculate the temperature in Kelvin from temperature in Celsius:

```
@Name("CelsiusToKelvinCalculator")
public class KelvinCalculator {

  @Action(name = "AddKelvin",
          input = "ETemperature",
          output = "ETemperature")
  public XMLEvent calculate(XMLEvent event)
      throws MalformedDocumentException {
    final Document document = event.getDocument();
    final Element rootElement = document.getDocumentElement();

    BigDecimal celsius = new BigDecimal(rootElement.getAttribute("celsius"))
                             .setScale(2, RoundingMode.UNNECESSARY);
    // kelvin = celsius + 273.15
    final BigDecimal kelvin = celsius.add(new BigDecimal("273.15"));
```

```
    rootElement.setAttribute("kelvin", kelvin.toPlainString());
    return new XMLEvent(event.getType(), document);
  }
}
```

Next we register the AddKelvin processor in the bundle activator:

```
@Override
protected void start() {
  register(new AnnotatedProcessor(new FahrenheitCalculator()));
  register(new AnnotatedProcessor(new KelvinCalculator()));
}
```

And when we run the example the TemperatureListener should print the temperature in Celsius, Fahrenheit and Kelvin.

```
New temperature
Celsius: -13
Fahrenheit: 8.6
Kelvin: 260.15
New temperature
Celsius: 6
Fahrenheit: 42.8
Kelvin: 279.15
...
```

This finishes the tutorial that explains how define the default actions for an event and create processor that can process these actions.

APPENDIX B

Query Criteria EBNF

Listing B.1 defines the EBNF for criteria in COPAL queries. Criteria is a logical expression (line 1) that consists of logical terms, tests and predicates connected with a 'and', 'or' and 'not' operators (lines 2–4). Predicate is either a new logical expression or a comparison (line 5). Comparison is a '=', '!=', '<', '<=', '>', or '>=' operation between two values, or a 'is null' or 'is not null' test for an event field (line 6). Values can either be a numeric expression or a string expression (line 7). Numeric expression consists of terms and factors connected with a '+', '-', '*', '/', or '%' operation (lines 8–9). Factors are either an event field, number or a new numeric expression (line 10). String expression consists of string terms connected with a '||' operation (line 11). String term is either an event field, string, or a new string expression (line 12).

Listing B.1: Query Criteria EBNF

```
1  criteria = [logical-expression] ;

2  logical-expression = logical-term | ( logical-expression , 'or' , logical-term ) ;
3  logical-term = logical-test | ( logical-term , 'and' , logical-test ) ;
4  logical-test = ['not'] , predicate ;
5  predicate = comparison | ( '(' , logical-expression , ')' ) ;

6  comparison = ( value , ( '=' | '!=' | '<' | '<=' | '>' | '>=' ) , value ) | ( ? field ? , ('is null' |
     'is not null') ) ;
7  value = numeric-expression | string-expression ;

8  numeric-expression = numeric-term | ( numeric-expression , ('+' | '-') , numeric-term ) ;
9  numeric-term = factor | ( numeric-term , ('*' | '/' | '%' ) , factor ) ;
10 factor = ? field ? | ? number ? | ( '(' , numeric-expression , ')' ) ;

11 string-expression = string-term | ( string-expression , '||' , string-term ) ;
12 string-term = ? field ? | ? string ? | ( '(' , string-expression , ')' ) ;
```

Bibliography

[1] ISO 8879: Standard generalized markup language (SGML), October 1986.

[2] ISO/EIC 14977: Extended BNF (1st edition), 1996.

[3] WAG UAProf. Technical report, Wireless Application Forum (WAP), October 2001.

[4] Unified modeling language (UML): Superstructure (version 2.3). Specification, Object Management Group (OMG), May 2010.

[5] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A bobile context-aware tour guide. *Wireless Networks — Special Issue: Mobile Computing and Networking: Selected Papers from MobiCom '96*, 3(5):421–433, October 1997.

[6] The OSGi Alliance. OSGi service platform: Core specification release 4 (version 4.3). Specification, April 2011.

[7] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[8] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007.

[9] Tim Berners-Lee. Uniform resource locators (URL). RFC 1738, Internet Engineering Task Force (IETF), December 1994.

[10] Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Network Working Group, January 2005.

[11] Tim Berners-Lee, Roy Thomas Fielding, and Henrik Frystyk Nielsen. Hypertext transfer protocol — HTTP/1.0. RFC 1945, Network Working Group, May 1996.

[12] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *ACM SIGOPS Operating Systems Review*, 21(5):123–138, November 1987.

[13] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A data-oriented survey of context models. *ACM SIGMOD Record*, 36(4):19–26, December 2007.

[14] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (5th edition). W3C recommendation, World Wide Web Consortium (W3C), November 2008.

[15] Martin Brown. Supporting user mobility. In *IFIP World Conference on Mobile Communications*, pages 69–77. Chapman and Hall, 1996.

[16] Peter J. Brown. The stick-e document: A framework for creating context-aware applications. In *Electronic Publishing '96*, pages 259–272, June 1996.

[17] Peter J. Brown. Triggering information by context. *Personal and Ubiquitous Computing*, 2(1):18–27, March 1998.

[18] Peter J. Brown, John D. Bovey, and Xian Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.

[19] Jay Budzik and Kristian J. Hammond. User interactions with everyday applications as context for just-in-time information access. In *5th International Conference on Intelligent User Interfaces*, pages 44–51. ACM Press, New York, New York, USA, January 2000.

[20] Maher Chebbo. Eu smartgrids framework "electricity networks of the future 2020 and beyond". In *Power Engineering Society General Meeting, 2007. IEEE*, pages 1–8. IEEE Computer Society, Washington, DC, USA, June 2007.

[21] Guanling Chen, Ming Li, and David Kotz. Data-centric middleware for context-aware pervasive computing. *Pervasive and Mobile Computing*, 4(2):216–253, April 2008.

[22] Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems — Special Issue: Papers from the International Conference on Very Large Data Bases*, 1(1):9–36, March 1976.

[23] Ekaterina Chtcherbina and Marquart Franz. Peer-to-peer coordination framework (P2PC): Enabler of mobile ad-hoc networking for medicine, business, and entertainment. In *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, January 2003.

[24] Jeremy R. Cooperstock, Koichiro Tanikoshi, Garry Beirne, Tracy Narine, and William Buxton. Evolution of a reactive environment. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 170–177. ACM Press, New York, New York, USA, July 1995.

[25] Douglas Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, Network Working Group, July 2006.

[26] Anind K. Dey. Context-aware computing: The cyberdesk project. In *AAAI 1998 Spring Symposium on Intelligent Environments*, pages 51–54. AAAI Press, March 1998.

[27] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. In *1st International Symposium on Handheld and Ubiquitous Computing*, pages 304–307. Springer-Verlag, London, UK, 1999.

[28] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, December 2001.

[29] Anind K. Dey, Gregory D. Abowd, and Andrew Wood. Cyberdesk: A framework for providing self-integrating context-aware services. In *3rd International Conference on Intelligent User Interfaces*, pages 47–54. ACM Press, New York, New York, USA, January 1998.

[30] Scott Elrod, Gene Hall, Rick Costanza, Michael Dixon, and Jim Des Rivières. Responsive office environments. *Communications of the ACM — Special Issue: Computer Augmented Environments: Back to the Real World*, 36(7):84–85, July 1993.

[31] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[32] EsperTech. *Esper: Reference Documentation (Version 3.3.0)*, 2009.

[33] Patrick Fahy and Siobhan Clarke. CASS — middleware for mobile context-aware applications. In *2nd International Conference on Mobile Systems, Applications, and Services — Workshop on Context Awareness*, June 2004.

[34] Stephen Fickas, Gerd Kortuem, and Zary Segall. Software organization for dynamic and adaptable wearable systems. In *1st International Symposium on Wearable Computers*, pages 56–63, October 1997.

[35] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, 2000.

[36] Roy Thomas Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, Network Working Group, June 1999.

[37] Lev Finkelstein, Evgeniy Gabrilovich andYossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited. *ACM Transactions on Information Systems*, 20(1):116–131, January 2002.

[38] David Franklin and Joshua Flachsbart. All gadget and no representation makes jack a dull environment. In *AAAI 1998 Spring Symposium on Intelligent Environments*, pages 155–160, 1998.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley Professional, 1st edition, November 1994.

[40] Richard M. Gustavsen. Condor — an application framework for mobility-based context-aware applications. In *Workshop on Concepts and Models for Ubiquitous Computing*, September 2002.

[41] Hugo Haas and Allen Brown. Web services glossary. W3C working group note, World Wide Web Consortium (W3C), February 2004.

[42] Marc Hadley and Paul Sandoz. JAX-RS: Java™ API for RESTful web services (version 1.1). Specification, Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054 USA, September 2009.

[43] Terry A. Halpin. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufmann, 1st edition, April 2001.

[44] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Altmann Leonhartsberger, and Werner Josef Retschitzegger. Context-Awareness on Mobile Devices — the Hydrogen Approach. In *36th Annual Hawaii International Conference on System Sciences*, pages 292–302, Washington, DC, USA, January 2003. IEEE Computer Society.

[45] Arnaud Le Hors and Philippe Le Hégaret. Document object model (DOM): Core specification (level 3). W3C recommendation, World Wide Web Consortium (W3C), April 2004.

[46] Richard Hull, Philip Neaves, and James Bedford-Roberts. Towards situated computing. In *1st International Symposium on Wearable Computers*, pages 146–153, October 1997.

[47] Cédric Kiss. Composite capability/preference profiles (cc/pp): Structure and vocabularies (version 2.0). W3C working draft, World Wide Web Consortium (W3C), April 2007.

[48] Michael Knappmeyer, Nigel Baker, Saad Liaquat, and Ralf Tönjes. A context provisioning framework to support pervasive and ubiquitous applications. In *4th European Conference on Smart Sensing and Context*, pages 93–106, Guildford, UK, September 2009. Springer-Verlag, Berlin, Heidelberg.

[49] Fei Li, Sanjin Sehic, and Schahram Dustdar. Copal: An adaptive approach to context provisioning. In *6th International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 286–293. IEEE Computer Society, Washington, DC, USA, October 2010.

[50] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, Boston, MA, USA, 1st edition, May 2002.

[51] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *4th International Conference on Web Information Systems Engineering*, pages 3–12. IEEE Computer Society, Washington, DC, USA, December 2003.

[52] Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *2nd International Symposium on Wearable Computers*, pages 92–99, October 1998.

[53] Elin R. Pedersen and Tomas Sokoler. AROMA: Abstract representation of presence supporting mutual awareness. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 51–58. ACM Press, New York, New York, USA, 1997.

[54] Paul Prekop and Mark Burnett. Activities, context and ubiquitous computing. *Computer Communications — Special Issue: Ubiquitous Computing*, 26(11):1168–1176, March 2003.

[55] Jun Rekimoto, Yuji Ayatsuka, and Kazuteru Hayashi. Augment-able reality: Situated communication through physical and digital spaces. In *2nd IEEE International Symposium on Wearable Computers*, pages 68–75. IEEE Computer Society, Washington, DC, USA, October 1998.

[56] Tom Rodden, Keith Chervest, Nigel Davies, and Alan Dix. Exploiting context in HCI design for mobile systems. In *1st Workshop on Human Computer Interaction with Mobile Devices*, May 1998.

[57] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October 2002.

[58] Nick S. Ryan. Mobile computing in a fieldwork environment: Metadata elements, 1997.

[59] Nick S. Ryan, Jason Pascoe, and David R. Morse. Enhanced reality fieldwork: The context-aware archaeological assistant. In *Computer Applications in Archaeology 1997*, British Archaeological Reports. Tempus Reparatum, October 1998.

[60] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. Ubiquitous computing: Defining an HCI research agenda for an emerging interaction paradigm. Technical report, Georgia Tech, 1998.

[61] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Conference on Human Factors in Computing Systems*, pages 434–441. ACM Press, New York, New York, USA, May 1999.

[62] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[63] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *1st Workshop on Mobile Computing Systems and Applications*, pages 85–90, December 1994.

[64] Bill N. Schilit and Marvin M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, September 1994.

[65] Guy Sharon and Opher Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, April 2008.

[66] Thomas Strang and Claudia L. Popien. A context modeling survey. In *6th International Conference on Ubiquitous Computing — Workshop on Advanced Context Modelling, Reasoning and Management*, September 2004.

[67] Chen Wang, Martin de Groot, and Peter Marendy. A service-oriented system for optimizing residential energy use. In *7th International Conference on Web Services*, pages 735–742. IEEE Computer Society, Washington, DC, USA, July 2009.

[68] Roy Want, Andy Hopper, Veronica Falc ao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[69] Andy Ward and Alan Jones. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[70] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.