uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Yousif Al Ridhawi

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

An Ontology-Based Negotiation Protocol and Context-Level Agreements

TITRE DE LA THÈSE / TITLE OF THESIS

A. Karmouch

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. Nayak

B. Esfandiari

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

An Ontology-Based Negotiation Protocol and Context-Level Agreements

Yousif Al Ridhawi

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MASc degree in Electrical Engineering

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
Ottawa, Ontario

Your file Votre référence
ISBN: 978-0-494-48585-9
Our file Notre référence
ISBN: 978-0-494-48585-9

Table of Contents

List of Figures

List of Tables

Abstract

Services and applications in pervasive environments must adapt to changes occurring in the surrounding environment and meet the needs of mobile users according to the users' changing situations. Existing context-aware architectures and middleware are faced with two problems. First, is their weakness in expressing complex inter-context relationships, stemming from use of less capable approaches to modeling contextual knowledge. Secondly, context dissemination methods used by these systems result in network flooding, unrestricted access to private context information, and the inability of consumers to limit or personalize received context.

This thesis provides an ontology-based context-level negotiation protocol along with a context-aware system architecture. The protocol permits context consumers to personalize their received context information through negotiations with context providers. The thesis illustrates the use of this negotiation protocol through the design and implementation of a context-aware system architecture capable of acquiring, modeling, reasoning and disseminating context information through ontologies.

Acknowledgements

I must first extend my thanks to my supervisor, Dr. Ahmed Karmouch, whose continuous support, guidance, and leadership have helped me throughout my research and thesis work. His commitment to work, and love for knowledge have set an example I will follow for the rest of my life.

I would also like to thank Hamid Harroud, Ibrahim Al-Oqily, Nancy Samaan, and other members of the Intelligence for Mobile Autonomic and Cognitive Networks Laboratory for their help and support during my research. Their companionship and friendship eased many tasks during times of difficulty.

Most Importantly, I would like to extend my thanks and gratitude to my parents for their constant and unconditional love and support. Thank you for helping me prove and improve myself through every step in my life. Thank you for all the sacrifices you made for me. You will always be a part of my heart, and I will be grateful all the days of my life for having the honor of being your son. To you I dedicate this thesis, for teaching me that nothing in life is impossible; if done one step at a time.

Acronyms

AVGPM	Automatic View Generation Policy Modifier
CASS	Context-Awareness Sub-Structures
CC/PP	Composite Capabilities / Preferences Profiles
CFS	Context File System
CIC	Context Information Center
CLA	Context-Level Agreement
CMC	Component Management Core
CMF	Context Managing Framework
CoBrA	Context Broker Architecture
COMANTO	Context Management Ontology
ConCoord	Context Coordinator
CSCP	Comprehensive Structural Context Profiles
CSR	Context Storage and Retrieval unit
CV	Context View
DAM+OIL	DARPA Agent Markup Language and Ontology Interchange Language
ELP	Extended Low Power Mode
ER	Entity Relationship model
ESIE	Expert System Inference Engine
GCO	Global Concrete Ontology
GPS	Global Positioning System
GSO	Global Skeletal Ontology
HP	High Power Mode
IDL	Interface Definition Language
IP	Internet Protocol
ISO	International Organization for Standards
JDOM	Java-based Document Object Model
LP	Low Power Mode
MAC	Media Access Control
MI	Message Interpreter

ORM	Object-Relational Mapping
OWL	Web Ontology Language
PDP	Policy Decision Point
QoC	Quality of Context
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RFID	Radio Frequency Identification
SCE	Semantic Context Entity
SHOE	Simple HTML Ontology Extensions
SIE	Simple Inference Engine
SOCAM	Service-Oriented Context-Aware Middleware
TCP	Transport Control Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language

Chapter 1

Introduction

1.1. Motivation

Until recently, computational systems have been restricted in their ability to adapt to changes within their surrounding environment. This is done through commands entered through human users and system administrators. Systems of this type were unaware of the environment within which they operated and did not have the ability to react or adapt to changes that occurred around them. These systems adequately served users and applications within static, centralized, and organized networks where users resided in small areas over prolonged periods of time. Applications were designed to meet simple user requirements within a stable environment with little or no variations in the device's usage or the physical environment surrounding the devices and their users.

With the introduction of mobile computing [1], there was an apparent deviation from the previous generation of desktop computing, as the absence of wires allows users to move freely without being restrained by wires. Although the advancement of mobile computing was of great benefit to users in constant move, most of the functions and design paradigms used in desktop computers were simply moved to mobile devices - such as laptops and first generation smart phones – regardless of the new requirements mobile services faced as users constantly moved between different networks, and as the surrounding environment and circumstances under which the services were provided underwent constant change.

Pervasive computing [2] took mobility even further and introduced a new concept to computing, by proposing an environment within which devices were interested in and responded to the surrounding environment and *type* of usage. One of the main goals was to improve the user's experience. Pervasive computing envisioned an environment saturated with computing and sensing devices that would capable of assessing the status of their surroundings as well as any changes taking place internally in the devices. The ability to communicate with other devices in the environment and changes in the statuses allow the devices to exchange knowledge and to form and organize decisions. The ability to conceive

the status of the internal or external environment and to use this information to improve the system's functionalities is known as context-awareness. However, a firm understanding of this term requires a clear definition of "context", one that highlights how this context could be used to improve existing services and applications.

1.2. Definition of Context

An understanding of how context information can improve context-aware systems requires a thorough understanding of the meaning of context. According to Webster's [3] dictionary definition, context is "the set of facts or circumstances that surround a situation or event". Due to the generality of this definition many researchers in the field of context-awareness have tried to form their own definitions of context. However, those attempts resulted in definitions that were very limited to the specific circumstances under which the researchers designed their systems, such as the location information of devices or people within a system's environment, or the identities of nearby objects and people [4]. Definitions that are too specific are therefore as difficult to apply in practice as those that are too general. Due to these limitations, researchers began proposing broadly applicable definitions that could meet the requirements of all context-aware systems. One of these definitions was proposed by Albert Schmidt in [5]; he suggested that context information is anything that could "describe a situation and the environment a device or user is in", and that each context is given a unique name and has a set of relevant features, each of which has a range of determined values. In another definition, Schilit [6] declared that context is characterized by three important aspects: the user's location, the person(s) whom the user is with, and the resources in the user's vicinity. Schilit also suggested that context can additionally include information about network connectivity, communication bandwidth and costs, light levels, noise levels, and so on. However, all attempts to define context for use within context-aware systems can be summarized by Anind Dey's work in [7], where he proposed that "context is any information that can be used to characterize the situation of any 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 application themselves". This is the definition we have adopted in our study.

Based on the above definition, a user roaming within the environment of a smart office may be provided with a number of applications and services that depend on utilizing the vicinity's available context information to present users with the highest level of adaptability. The system may monitor all employees as they enter the office building (location context), and, based on their identities as acquired through RFID tags (personal context), determine the workstation that should be prepared for the employee's arrival. By knowing the current time of day (temporal context) the system would set the light level within the office, and temperature and humidity readings (environmental context) sensed from within the room would be used to determine whether heating or air conditioning should be turned on. The system may also use calendar information (personal context) prepared by the employee or work colleagues to determine whether any important meetings or events will be taking place during the course of the day. This information would then be forwarded to the employee's handheld device for notification before arrival at the desk. Once the employee arrives at the desk, the system begins launching applications according to the employee's needs and usage urgency. The system would sense the employee's location and begin forwarding all incoming calls from the handheld device to the office's phone, and during important meetings the system would sense the urgency of the situation (social context) and forward all incoming calls to the voicemail box.

As this example illustrates, it is obvious that our surrounding environment contains an unlimited amount of context information that can be captured, stored, analyzed, reasoned about, and utilized to improve the user's experience. The goal of pervasive computing is an environment where sensors and computing devices have dissolved so well into the environment that they become seemingly invisible to human users. However, limitations inherent in mobile devices must be taken into consideration during the design of context-aware systems. Lack of long-life batteries, CPU power and memory, as well as usually small screen sizes, require the development of mobile applications that have a small footprint on the devices' resources. The incorporation of sensors into mobile devices also introduces strain on power consumption, which forces designers to abstain from necessitating mobile devices to sense the entire spectrum of needed context by themselves. The disadvantages of this first design approach has led researchers to adopt the outsourcing of the context sensing process to widely-distributed sensors that are independent of the mobile devices, for the

majority of the context information needed within context-aware applications. For example, the ability of automatic reconfiguration of network routing paths requires monitoring bandwidth utilization and packet-drop rates for various nodes within the network; a process that would be impractical if done solely by the mobile devices themselves.

A second approach to context-aware system development proposed the use of mobile devices as a distributed network of context sources that share their needed context according to each device's abilities. This approach provided each mobile device the ability to acquire context information they would have not been able to collect alone, given the limited amount of sensors directly attached to each mobile device. Unfortunately this design approach also had several limitations. There is a need for a shared protocol between all devices. This protocol is used by each device to express its interest in context information from a peer device. The protocol should also reduce network congestions and delays by solving problems associated with the path taken by messages to and from other devices. The third problem stems from the most important characteristic of pervasive, ubiquitous systems: high mobility. The continuous movement of mobile devices (nodes) in and out of the system results in the loss and addition of context sources as well as the need for constant reconfiguration of the established network between peer devices. The issue of how existing devices are notified of the loss or arrival of a source should also be addressed.

The aforementioned problems are difficult to resolve and cause unnecessary problems in context-aware system design. The third approach to designing context-aware systems adopted the use of a shared context-aware middleware. Context acquisition, inferring new context from acquired raw context, modeling and storing context, and disseminating context to interested devices and applications are resource-intensive processes. It is therefore advantageous to have a powerful middleware responsible for providing the necessary context information to interested mobile clients.

1.3. Context Delivery

Any approach used to design a context-aware system entails the question of how to deliver the context information to interested clients. One proposed solution is the use of the pull model, where context users must access the context they are interested in from the context provider every time the context are needed. Unfortunately though, this approach imposes

problems in time efficiency that context users experience, and they may also not realize that no changes have occurred to the context values since the previous pull. The pull method also has security and privacy issues that need to be solved. Another traditional approach to context delivery is the use of information dissemination, as described by Ragab K. et al. in [8]. They proposed an application-level multicast to distribute information from its source to all other members of the community. Ragab et al. used what they described as an "efficient approach to information broadcasting" that distributes traffic among network nodes evenly.

In their proposal, a node that receives new information distributes that information to N of its neighbors. Each of these N neighbors will then send the information to N of its neighbors until each node in the network has received the new information. The same approach is used to locate information in which a node has interest. The requesting node sends its request message to N neighbors, who send back the requested information if it is available within their knowledge base. Otherwise, the request is forwarded to each of the nodes' N neighbors until the information is found and sent back to the requesting node.

Context dissemination through broadcasting possesses many problems due to the dynamic nature of context information:

1. Context-aware systems sense and generate tremendous amounts of context information. If a broadcast of the context information is triggered every time new context is sensed, then context users are flooded with unwanted information. Network delays are unavoidable in these cases, and limited resources within mobile devices restrict the devices' ability to store all newly acquired contexts and to maintain storage of old context information for future reference.
2. Privacy in context-aware systems is highly important. Users within a context-aware environment might require their activity to be hidden from other users. The use of broadcast context dissemination as in K. Ragab et al.'s approach does not distinguish between private context (accessible by a limited group of clients), and public context (accessible by all clients within the system).
3. Providing users with context information they have no interest in receiving can waste limited available resources within mobile devices. Moreover, time and processing power

spent while receiving and broadcasting context to other devices limits the devices' abilities to meet the users' needs.

The problems introduced by broadcast-type context dissemination requires more economical and secure context-aware systems that protect the client's privacy and limit context dissemination to only the needed context information. This said, the design of context-aware systems must meet the following requirements:

- Context consumers, who could be referred to as Context Clients within ubiquitous computing systems, are not equal. They differ from each other in their identities and functionalities. Many context-aware systems identify users based on the roles users play within the system. Fore example, a department manager might have access to a wider range of context information than other employees, often for security reasons. This might be due to security reasons. The functions of context clients may also determine the context information the clients would be interested in acquiring. An application running on a 'smart' router would not be interested in knowing the weather conditions outside an office building. The router is instead interested in the packet arrival rates or packet drop rates in neighboring network nodes. Therefore, we propose that any context-aware system must have the ability to make decisions on what context information to provide Clients with, based on the Clients' identities, roles, functionalities, and so on.
- We identified a number of issues with available context dissemination approaches, which follow the broadcasting method. These approaches overflow clients with unwanted context, breach the clients' security by making context knowledge public for all network nodes, and consume network resources by exchanging unnecessary messages. Context-aware systems must therefore only deliver requested context information to clients, which reduces traffic within the network and simplifies context acquisition by clients since fewer messages need to be decoded.
- Context information collected directly through sensors, or produced through inference, composition, or aggregation, tend to differ in their qualities. Different types of context information have different accuracy and certainty levels, are refreshed regularly using different frequencies, and may also differ in their resolution for instance, by having fine-grained context for the exact light level within an area or a more coarse grained context

to show whether lights are ON or OFF. Hence context-aware systems must also be able to bind available context information according to their qualities of context.

1.4. Thesis Contributions

Given the diversity of available context information, the need to personalize delivered context for clients' needs and requirements, and the need to reduce the amount of context information being disseminated within context-aware systems, these systems must offer a solution that allows context Clients to negotiate their context needs with available context providers.

The goal of this thesis is to propose a solution to these problems by introducing the possibility of performing context-level negotiations which would permit Clients to personalize the context information received from the context provider. Negotiations are used to establish what we call Context-Level Agreements (CLA). CLAs are agreements between context providers and context Clients reached through negotiations detailing the context needs of Clients and the responsibilities of context Providers.

The following is an overview of the main contributions of this thesis. First, the use of ontologies to model context information received by context Providers, structure context request messages from context Clients, and create context Views for negotiating clients. Moreover, we illustrate the separation of Clients' access to context information based on the new concept of Client Views.

In addition, we show the design and use of an ontology-based negotiation protocol to express the clients' context interests and the providers' context delivery abilities are exchanged to establish Context-Level Agreements (CLA) [RIDH 08a] [RIDH 08b].

Finally, we describe a complete context-aware system architecture that uses our context-level negotiation protocol and enforces context information delivery through the established context-level agreements.

1.4.1. The Use of Ontologies

Entities within context-aware systems require a common understanding of how key concepts and knowledge is represented. Therefore, we have based our modeling of acquired context, as well as context generated through inference, aggregation, or composition, on ontologies.

Client Views representing accessible context information by a Client or group of Clients also employ ontologies in their structure. We have also designed our context-level negotiation protocol so it is compatible with any ontology the context-aware systems decide to adopt.

Ontologies are used to describe concepts within an existing domain, as well as the relationships that may exist between these concepts. Ontologies are also used to share domain information knowledge between different entities and to simplify the ability to reason and deduce new context based on existing contexts and relationships. They could easily be projected into data structures for use by computers.

A number of markup languages have been used for the purpose of context modeling. The models used by researchers have either adopted some of these languages or tended to form their own methods for representing their acquired context. We have adopted the use of the Web Ontology Language (OWL) [16] to model all ontologies within our negotiation-based context-aware system. OWL describes modeled data by using sets of classes and properties, and provides the flexibility and extensibility necessary within pervasive environments.

Within our architecture, we have assumed the presence of an OWL-based ontology covering an extended area of knowledge. This ontology is formalized by defining a set of classes and properties from which context-aware systems (Context Providers) would be able to extend and thus create their personalized domain-specific ontologies. Classes within these provider-specific ontologies are used to define sets of individuals present within the providers' domains and to assert properties about these sets. The Context Providers' knowledge is continuously loaded with context values as newly-acquired contexts arrive. The context-level negotiation protocol used to reach these agreements has also been designed in accordance with the structure of OWL-based ontologies.

1.4.2. Separation of Clients Through Views

Our ontology is also used to build Client-specific sub-ontologies which we refer to as Client Views. These define their rightly accessible context information, including classes, properties, relationships, and individuals. Views are used to inform clients of what context information they are permitted to access from within the context Providers, given that access has been accepted by the Provider according to the established context-level agreements (CLAs). These views are used to hide all inaccessible context information from Clients that

do not have the prerequisites for the hidden context. The choice of what Views to send to the clients depends first on the information presented by the Clients within the ontology-based profiles they submit as part of their negotiations, and secondly on the design of the context-aware system.

1.4.3. Ontology-Based Negotiation Protocol

In order for context consumers (Context Clients) to notify context sources (Context Providers) about their interest in receiving certain types of context information, a Context-Level Agreement (CLA) must be reached. These CLAs are formed through the context-level negotiation protocol we have designed, which allows clients to send requests for their needed context while context providers accepted these requests or reject sending back their counter-offers. Our ontology-based context-level negotiation protocol gives Clients the ability to:

- Send their profiles to context Providers for identification purposes and receive their context Views to form their context requests;
- Negotiate the CLA activation conditions;
- Negotiate the context information that Clients are interested in receiving, the conditions under which these contexts will be delivered, and the quality levels the received context information must meet prior to delivery to the Client;
- Negotiate the conditions under which the Clients' Views are updated and sent back to the affected Clients.

1.4.4. Context-Aware System Architecture

Due to the changes that context-level negotiations bring to existing designs of context-aware system architectures, we propose a new architecture that incorporates the ability to acquire context information from different context sources to reason about and infer new context, and to store the context. The architecture can also model the received context information by using ontologies, and employs the ontologies in forming all necessary Clients' Views. Particular entities within the Context Provider have been designed to negotiate with incoming Clients using the results of these negotiations to create CLAs and to enforce those CLAs to supply Clients with the needed context information.

1.5. Thesis Outline

The remainder of this thesis report is structured as follows:

In Chapter 2, we present background information and related work that has motivated us to the design of a new context-aware system architecture and our ontology-based context-level negotiation protocol. This chapter presents the different context models that are currently used, as well as some of the context-aware system architectures and middleware available. We also present their advantages and disadvantages.

Chapter 3 presents our ontologies. We begin by describing an ontology shared by all entities present within a context-aware system discussing a wide variety of concepts from many domains and fields of knowledge as well as their relationships to each other. This ontology, referred to as the Global Skeletal Ontology, gives existing entities the ability to model their acquired knowledge in a way that is simple, sharable, and easily interpreted by computers using the OWL syntax. We also show how context Providers can extend the Global Skeletal Ontology to form their own domain-specific ontologies referred to as Global Concrete Ontologies utilized to model the Providers' updated knowledge from their acquired context information. The concepts of Client Profiles and Views are also discussed in chapter 3. The former provides an ontological method of expressing the context consumers' identities and capabilities, while the latter provides Clients with a clear model describing the context information they have the right to access within the current domain of the Provider.

In Chapter 4, we present our context-aware system architecture. The architecture has been designed to overcome some of the limitations found in earlier context-aware systems and middleware. We discuss the components of the three architecture layers: Sensing layer, Provider layer, Client layer. We also illustrate the importance of using ontologies, and most importantly, the different layers and components involved in context-level negotiations to achieve the Context-Level Agreements that formally describe the Clients' context needs and the Providers' responsibilities.

In Chapter 5, we thoroughly describe our context-level negotiation protocol. We illustrate the steps taken by both Clients and Providers to achieve Context-Level Agreements. We also describe the structural details of protocol messages exchanged for this purpose.

In Chapter 6, we present our detailed implementation of a prototype system illustrating the validity of our context-aware systems architecture and the formation of Context-Level Agreements through our context-level negotiation protocol. We discuss the implementation of each of the three layers in our architecture and present the step-by-step formation of a sample agreement.

In Chapter 7 we conclude this thesis. We summarize our contributions and the lessons learned from our ontology-based context-level negotiation protocol and context-aware system architecture. We also discuss our future research plans and potential enhancements to our current design.

Chapter 2

Background

2.1. Chapter Objectives

This chapter presents an overview of some of the background information used in this thesis. The chapter demonstrates the different context modeling approaches recently used in context-aware systems in terms of their advantages and disadvantages. The chapter then proceeds to discuss the various context-aware system architectures that have been proposed, outlining their main features, the differences between these architectures and the approach proposed in this thesis, and the ways in which these architectures could be improved by the proposed negotiation protocol.

2.2. Context Modeling Methods

Knowledge in context-aware systems requires a unified method of modeling and representation. The different types of clients and providers that need to understand each other require that context information be represented uniformly throughout the system. The system must also be flexible and extensible enough to handle the wide range of context types, as well as the relationships between them. Most context-aware systems are distributed; it is therefore necessary for context models to be easily shared especially in our architecture, given its use of Profiles and Views. Models should also have a high level of formality and be able to represent existing context relationships. For these reasons, researchers have developed a number of different methods for modeling context information. Some of these models do not meet the requirements imposed by context-aware systems, while others can be used as the basis for our ontology and proposed protocol. In the following subsections, we provide an overview of available context models.

2.2.1. Comprehensive Structured Context Profiles

Albert Held and Alexander Schill's Comprehensive Structured Context Profiles (CSCP) [13] are based on the Resource Description Framework (RDF) expressing context information

through the use of session profiles. The RDF-based language used by CSCP has the advantage of eliminating the need to define a fixed hierarchy. Therefore, the natural flow of context profiles could be easily expressed. Unlike CC/PP which required giving unambiguous attribute names, CSCP's attributes are interpreted according to their positions within the profile structure.

Profiles in CSCP refer to RDF-based files that express context information associated with a given client's session at the Mobility Portal. Profiles are divided into device profiles, network profiles, user profiles, and so on, and have to be assembled by the user during session establishment. During the lifetime of the established session, the client updates the existing profiles through the use of differential profiles. These differential profiles contain a reference to the previous profiles as their "default" settings and override any attribute values that have changed since the last update.

CSCP is faced with many limitations when it comes to applying the concept into context-aware environments. The use of RDF syntax for profile representation makes it an unlikely candidate for context representation due to the existence of more powerful ontology languages such as RDF/S and OWL. Secondly, CSCP is still limited by its vocabulary to a limited set of context concepts and relationships. The complex relationships existing within context-aware systems make the process of modeling these concepts and relationships unintuitive for system designers, which complicates the process of building a context-aware system.

2.2.2. Composite Capabilities/Preferences Profiles (CC/PP) Extensions

Like CSCP, CC/PP [13] is based on RDF and focuses mainly on describing capabilities and preferences for wireless devices and mobile phones. Based on the listed preferences and capabilities, profiles are used by servers to deliver customized content to the devices. CC/PP profiles are composed of two-level trees that list components and their attributes. Each statement in the profile is composed of a component with a list of named attributes and the values of these attributes. The types of components and attributes allowed within a profile are dictated by one or more schemata.

The main goal behind CC/PP was to create on which could be based the available features within the computing environment, the user's requirements, preferences, and application capabilities, as well as the requirements of the computing applications. The CC/PP extension introduced by Indulska et al. [14] simply extended the original CC/PP vocabulary to include a wider range of context information about Network Interfaces of devices, Quality of Services, Location, Disconnection Status, and Application Requirements. Indulska et al. also included several relationships and dependencies that grouped related components together in their own profiles. An example of a CC/PP Location extension is shown in figure 2-1. In that extension, Location is broken down into five categories: physical location, logical location, geodetic location, orientation, and an error measurement component called Modifications.

```
[LocationProfile
      [PhysicalLocation [Country, State, City, Suburb]]
      [LogicalLocation [IPAddress]]
      [GeodeticLocation [Longitude, Latitude, Altitude]]
      [Orientation [Heading, Pitch]]
      [Modifications [VeriticalError, Horizontal Error, HeadingError,
      PitchError]]
]
```

Figure. 2-1 CC/PP Extension example [13]

As Indulska had openly admitted there are however many limitations to the use of CC/PP for context modeling. The way in which CC/PP is structured makes it difficult to capture and represent all context information present within pervasive environments such as accuracy, resolution, as well as the temporal characteristics needed to model the freshness of acquired context as an example. The Component-Attribute model used in CC/PP causes many difficulties for multi-layered attributes, and the process of profile creation. CC/PP does not have the ability to constrain the list of elements within a container for a certain cardinality or type. Finally, CC/PP does not define any relational constraints for attributes within a profile component. For instance, it would not be possible to indicate that if an attribute called "resolution" was present within a profile, then it is a requirement to have another attribute called "bitRate" also present.

2.2.3. Graphic-Based Context Modeling

When Henricksen et al. [14] introduced his ORM-based graphical context model, he wanted to overcome the lack of formality and expressiveness present in the available context modeling approaches. Existing graphical models, such as UML and ER, presented some difficulties in modeling all features associated with context information. These features include uncertainties, histories, and dependencies between different types of information.

ORM usage permits the mapping of the graphical models to relational models, as well as the ability to express constraints present within context information. Concepts in ORM are modeled using sets of facts (represented by a sequence of role boxes) and the roles of entities (represented by ellipses) within these facts. Facts are divided into two categories; static and dynamic. Static facts are those facts that remain unchanged during the entity's lifetime and declare a set or sets of roles against which the facts were invariant. In contrast, dynamic facts are those that could change over time and are acquired through profiles, sensors, or derived from other facts.

Temporal facts in ORM are used to capture time-indexed data that acts in a similar way as timestamps, representing the start and end times of the fact (e.g. a person engaged in an activity will have a start and end time for his commitment to that activity, as in Figure 2-2).

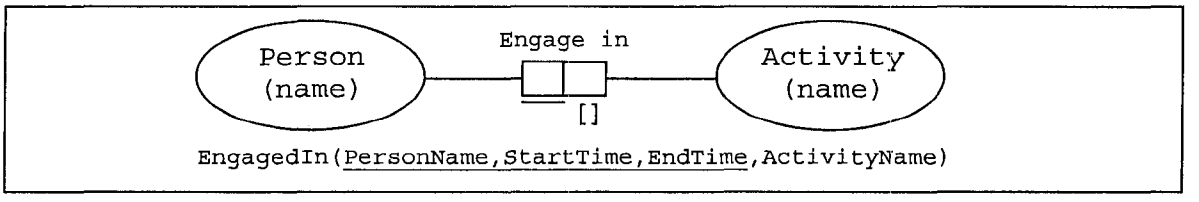


Figure. 2-2 ORM modeling: Example of a user's current activity [14]

Dependency between existing facts in ORM is captured by the *dependsOn* relationship, which indicates that if a state change occurs in one fact, then the state of the dependant fact is also affected. Henricksen also provided an approach to modeling the quality of context associated with each fact by adding quality indicators, such as freshness and accuracy.

The ORM context model can be easily mapped to a context management system through ORM's relational mapping procedure, such that constraints expressed within the model can be enforced by a relational database during runtime. This mapping process is done through

the use of the *Rmap* procedure, where each fact is mapped to a unique relation (except for dependencies, which have no explicit representation in the relational models). Dependencies must be stored and managed separately from other facts, by the management infrastructure.

Graphical models like ORM, have a strong ability to describe the structure of context knowledge and to derive the required code, such as the one associated with ORM's relational code. Unfortunately though, merging distributed context models is not fully effective, given the constraints associated with the act of merging relational databases. Since graphical models are used mainly to facilitate human readability, most graphical-based context modeling approaches present difficulties in merging different context models and would not be suitable for resource-deficient mobile devices. Although graphical approaches simplify the design stage of context models, yet computing devices are indifferent to the existence of context models readable by humans as long as they are easily interpreted by computing entities and can be easily shared between these entities. Therefore the resource-intensive graphical context models are not favorable for use in context-aware systems.

2.2.4. Cues – An Object-Oriented Modeling Approach

Object-oriented context modeling methods employ the encapsulation and reusability capabilities to meet the dynamic nature of context information. One such approach was presented by Albrecht Schmidt et al. in [5] with his use of *cues*. The model was based on the assumption that context described the situation and environment surrounding the users or devices. Each context was given a unique name and a set of relevant features for which a range of values was determined. Based on this model, two main context groups were created; a context group related to Humans and a second group related to the Physical Environment. Each of these two groups was further broken down into subcategories from which further subcategories were derived.

Cues provided an abstraction from physical and logical sensors, by providing a symbolic output based on a function that read the incoming values of a single sensor for a certain time period. Each cue depended on a single sensor, and sets of possible values were defined for each cue. The available cues were then used to derive the current situation on an abstract level, resulting in context information described by two dimensional vectors: one describing the situation the context is in, and the second containing a number used to indicate the

certainty of that situation. Shown below is a Schmidt example of how different acquired cues could be used to derive a context decision. Readings from a light sensor can be fed into statistical functions to determine that the current light status is that of an 'artificial light' source. Similarly, motion detectors can determine that the user is 'walking' in the monitored area, and functions reading values from environmental sensors can be used to determine that the temperature readings fall within the bounds of 'room temperature' and that the humidity level can be classified as 'dry'. This list of derived facts exhibits what is known as 'cues'. The presence of this list of *cues* can then be used to derive the user's current *context* 'in his office'.

Context	*Cues*
In Office	Artificial light, stationary or walking, room temperature, dry

The details used to derive context information are usually hidden from other components, and the context information itself is only displayed through specified interfaces. This presents a drawback to the formal representation of context information, since important low-level context information sometimes is hidden from the clients. Another problem with the use of object-oriented context models, such as the cues model, is the stress put on computing device resources. This is a disadvantage within ubiquitous computing where mobile devices are limited in their computing abilities and available storage.

2.2.5. Rule-Based System

A different approach to context modeling utilized formal logic to model the semantics of the context-aware environment. One such approach was illustrated by Jean Bacon et al. [15] in their location-oriented multimedia system. They developed a system that used an event-based mechanism to support location awareness.

Bacon extended the Interface Definition Language (IDL) to handle the occurrence of events such that servers could declare the events they were capable of notifying. For example, incoming clients would be able to register with a location service given the advertised facility declared in IDL as shown below:

```
LocEvent: EVENTCLASS [ person : USER, location : PLACE ];
```

Clients would use the advertised service to provide their 'event template', indicating their range of interest -such as their interest in receiving a notification whenever 'Mike Lee' is seen in the 'Meeting Room' (whereby 'USER' is replaced by 'Mike Lee' and 'PLACE' with 'Meeting Room'). The model proposed was based on an entity-relationship data model implemented in Prolog. The database used was given an initial set of data and was updated through the use of events giving the most recent locations of people and equipment within the environment. Rules were added to the system as they were needed, and searches were done by submitting queries to the database. An example of such queries is shown in the following command, which searches for the presence of devices with video capabilities in close proximity to the favored person.

```
Video_in_room(P):-location(P,L),location(W,L),workstation(W),has_video(W)
```

Logic-based context models have a high level of formality and express context information in the form of facts, expressions and rules where context information facts are added, deleted or updated within the system. Many logic-based models do not seem applicable to many of the existing context-aware systems, and none seem to provide a means for quality of context representation. Representing complex relationships within ubiquitous environments would also be a highly complicated process using logic-based models. Creating a complete set of rules capable of representing the wide range of concepts, relationships, and properties within context-aware environments is a complex and time consuming task, rendering the use of logic-based context models unfavorable.

2.2.6. Ontologies

One of the most widely researched context modeling methods has been that of using ontologies. This is mainly because ontologies can represent concepts and relationships by employing a computer-usable data structure while sharing a common understanding of the domains in which the context-aware system has interest. This ontology knowledge is usually represented through a set of entities, functions, instances and axioms. Ontologies are also known for their normalization abilities and formality, making them a favorable candidate for modeling context knowledge.

Most proposed ontology-based context models have adopted the emerging OWL [16] language as a means of ontology representation. This is due to OWL's superior expressive

abilities over other languages, such as XML, RDF, RDFS, or DAML+OIL. OWL also allows the exchange and comprehension of context information between different entities within context-aware systems. The availability of a number of OWL-based reasoning engines that can interpret context information present within the ontology, or that can infer new context from existing context and relationships, makes OWL an especially effective language for context modeling.

Unfortunately, there is no unified OWL-based ontology for context representation within context-aware systems. However several research groups have tried to find a generic model on which all context models could be based. COMANTO [17], an ontology proposed by Stimpakou, is one such attempt. COMANTO (COntext MAnagement oNTOlogy) is a generic upper-level context ontology that is not specific to any domain, application, or condition. At the very top of the ontology, a class called Semantic Context Entity (SCE) represents the root from which the most widely used concepts extend. These concepts have been categorized into 11 subclasses:

- Person: used along with various properties to describe all human entities, user-related contexts, relationships, and behaviors.
- Place: used to describe the physical space in terms of symbolic or geographical representations.
- Preferences: describes the user's device, service, and network preferences.

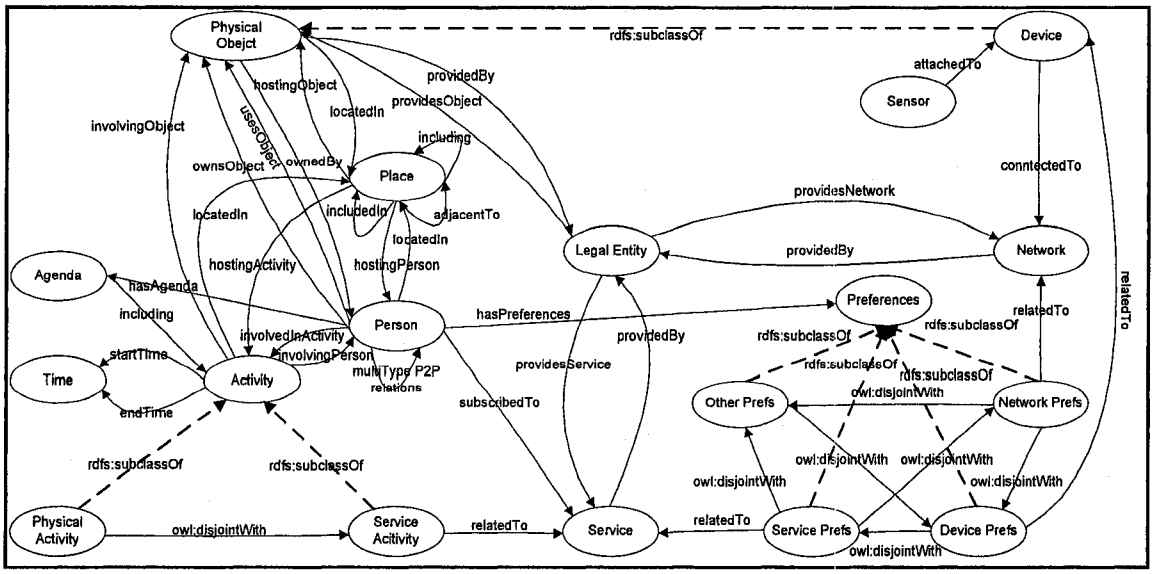


Figure. 2-3 COMANTO upper-level ontology [17]

- Agenda: contains the user's calendar information modeling the user's activities.
- Activity: divided into two types: physical activity and service activity. These two types model the user's tasks whether they are conveyed physically, such as working on a laptop, or part of a service, such as using a particular application.
- Time: contains all necessary information related to the current time (used mainly for timestamps).
- Physical Object: any object that could be described physically (except for devices).
- Sensor: contains the sensors' configuration datatype properties.
- Service: class representing information to all services and applications the user subscribes to.
- Network: models context information about the underlying network.
- Legal Entity: corporate actors involved in the system.

Strimpakou et al. used the COMANTO ontology to describe the properties, relationships, and structure of context objects, while their context management infrastructure employed an object oriented context model for context information communication. An illustration of the COMANTO upper ontology is shown in figure 2-3.

In keeping with the idea presented in COMANTO, researchers developing a Service-Oriented Context-Aware Middleware (SOCAM) [18] have employed an OWL-based ontology that takes the idea presented within COMANTO one step further. SOCAM's ontology employs a similar division of a generalized upper ontology and a set of domain-specific low-level ontologies. The latter can be dynamically plugged and unplugged from the upper ontology, based on changes in the environment, such a user's movement from one domain to another. The upper ontology is broken down into four subcategories: person, location, computational entity, and activity.

SOCAM's ontology also differentiates between the different sources of acquired context information. This is done through the use of a property named '*owl: classifiedAs*' that categorizes context information depending on whether it is sensed, defined, aggregated, or deduced. A dependency relationship property was also included in the ontology; '*rdfs: dependsOn*'. An example of dependency between properties could be the dependency of the

property '*ScheduledActivity*' on '*locatedAt*' and '*weatherCond*'. These properties represent the user's location and the weather conditions, respectively.

An interesting and useful quality of context representation is also included in the SOCAM ontology, allowing it to extend the sensed context with quality constraints, such as accuracy and freshness. Each constrained quality is associated with a number of quality parameters, and each parameter is described by a set of appropriate quality metrics. The quality parameters are accuracy, resolution, certainty, and freshness. SOCAM's upper ontology and quality constraints are shown in figures 2-4a and 2-4b.

SOCAM provided the services COMANTO lacked in quality of context ontology representation. However, SOCAM did not clarify whether the time ontology should be a subclass of the ContextEntity root, or whether it should be defined separately within each sub-domain created.

A slightly different OWL-based ontology approach was presented by Chen and Finin in their Context Broker Architecture (CoBrA) [19]. Their top-level ontology was divided into classes that belonged to one of three categories; Person, Place, or Intention. In total there were 17 classes and 32 property definitions. Given the small number of classes and properties, CoBrA's ontology was limited to people, places, and activities within a context-aware system.

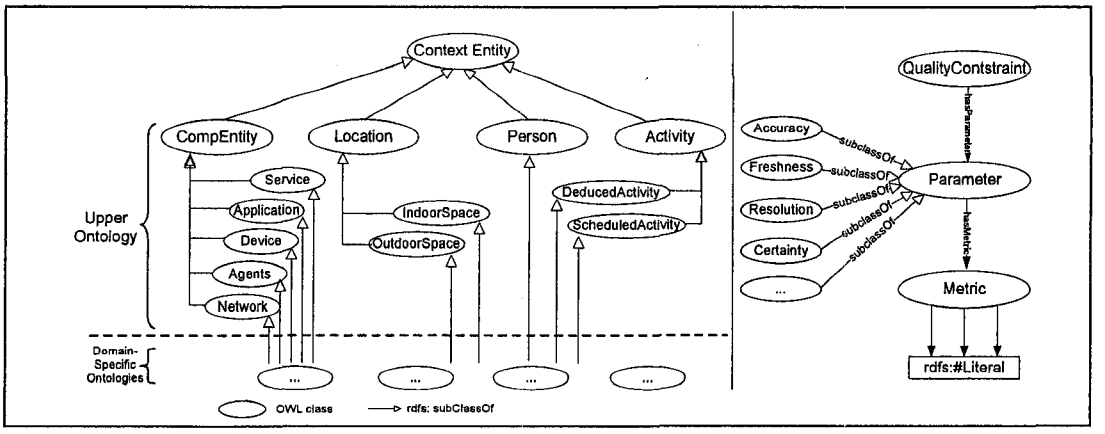


Figure 2-4. a)Upper-layer SOCAM ontology, b)SOCAM quality ontology [18]

The 'Person' class was used to define general properties about a person in an intelligent environment, with subclasses listing several types of people: those currently within a building (*PersonInBuilding*), invited speakers in a meeting (*TalkEventHost*), people invited

to speak in a meeting (Speaker), and nonparticipating people in a meeting (Audience). The subclasses of the *'Place'* class were restricted to UniversityCampus, Building, Room, OtherPlaceInBuilding, and MeetingPlaceInBuilding, since the restricted nature of their system architecture concentrated on smart meeting rooms and university environments. The 'Intention' class contained information concerning a user's current intention (to give a speech). This information was used to provide the needed services to the user based on their future actions.

2.3. Context-Aware Systems Architectures

The diverse approaches to modeling context information discussed in section 2.1 are usually incorporated into context-aware frameworks to provide modeling, reasoning, and sharing of context knowledge within ubiquitous environments.

This section provides an analysis with the advantages and disadvantages of five existing frameworks for context-aware systems. Also discussed are some of the possible benefits if ontology-based context-level negotiations were incorporated into the systems' designs.

2.3.1. Context Managing Framework (CMF)

The Context Managing Framework (CMF) [20] proposed by Korpipää et al. provided a mobile terminal framework for context acquisition and processing provided to applications. Context delivery was event-based and their API used an ontology to define useful context information for clients. The CMF consisted of four entities:

- Context Manager: Functions as a central server for communicating entities (resource server, context recognition service, change detection service) that use the services provided by the context manager. The manager stores context information available to be served to connecting clients.
- Resource Server: Context information stored within the Context Manager arrives from resource servers. The resource server is an entity that connects to sources of context (such as sensors) and posts the collected context to the context manager's blackboard for further processing and delivery to connected clients. Low-level reasoning is done by the resource server over incoming context data before storing it in the context manager at a higher level.

- Context Recognition Service: Used to register plug-in context recognition services, which allows applications to share and interpret higher-level contexts. The recognition service works with the resource server to convert raw context data to a format defined by the context ontology.
- Applications: Seen as context clients or consumers with interest in acquiring part of the context information provided by the context manager. Clients can be served with context information in three ways: directly by querying the manager's database, by subscribing to context change notification services, or by transparently using higher-level contexts when contacting the required recognition services.

The ontology context modeling method used in CMF is inferior to many existing models. Although we have indicated that OWL was more powerful than most existing languages, CMF's RDF ontology was simply divided into a hierarchy of context types and associated values; Korpipää et al. provided no solution for representing relationships or complex properties within the modeled context. Context within the ontology was described by up to six properties: context type (which describes the context category), context value (which refers to the semantic high level or absolute value of the context type), confidence (a value between 0 and 1 describing the uncertainty within the context value), timestamp (the time the context update occurred), and other attributes containing additional details.

Clients' context requests were restricted to two request categories:

1. Requests and responses: Clients using this type of request send the context manager a request for the context type they have interest in receiving and the context manager replies with the context value. Below is a client request for the humidity status and antenna position of a device, along with the manager's reply:

```
Client's request: ContextSetRequest({Environment: Humidity,
                  Device: Activity: Position})
Manager's reply:  {Dry, AntennaUp}
```

2. Subscriptions and notifications: Clients subscribing to context change notifications are given the required context information in an event-based manner. An example may involve a notification every time the devices position changes:

```
Client's request:       ContextChangeSubscribe
                        (Device:Activity:Placement)
```

```
Possible notification: AtHand
```

The Request/Response and Subscription/Notification context acquisition categories are too limited to provide a flexible means of expressing the client's wide range of context interests in ubiquitous environments. Clients should have the ability to set the conditions and time during which notifications and context values must be delivered to them. Clients should also be able to express their context needs by using relationships present between context information that is usually found in context models such as OWL but is missing in the RDF model provided with CMF.

Although Korpipää et al. provided a detailed description of how the confidence quality is calculated within their system, there was no mention of how that information could be utilized by clients in context requests. The use of quality of context measurements in context filtering is an essential part of a context negotiation protocol because it reduces the amount of messages delivered to clients, instead of flooding clients with inaccurate values or values in which the manager itself has no confidence.

2.3.2. Service-Oriented Context-Aware Middleware (SOCAM)

Another middleware for context-aware systems was proposed by Tao Gu et al. in their Service-Oriented Context-Aware Middleware (SOCAM) [18] that has the ability to acquire context from various sources, interpret and reason about the context, and perform context dissemination.

Each component in SOCAM is advertised, located and accessed through a Service Locating Service, and context within the system is formally represented through an OWL-based ontology. The ontology used is similar to that found in COMANTO [17], which was divided into a generalized upper ontology and domain-specific sub-ontologies.

SOCAM's middleware consists of Context Providers, a Context Interpreter, a Context Database, a Service Location Service, and Context-aware Mobile Services. Context Providers provide context abstraction to services and are divided into External and Internal context providers. *External* context providers acquire their context from external sources, such as web servers, while *Internal* context providers acquire their context directly from

sensors. The acquired context could be converted from its low-level context to a higher-level context through the Context Interpreter.

The Context Interpreter is divided into a 'reasoner' that is responsible for deducing the higher-level context from the low-level ones, and a 'Context Knowledge Base' where context knowledge can be queried, added, deleted or modified. All acquired context is modeled and stored according to an OWL-based ontology.

Context-aware services interested in acquiring context information submit a first-order logic query to the Service Location service, such as "*socam:locatedIn(MyCar ?x)*". This query searches through the ontologies submitted by Context Providers during their registration in order to find a provider capable of meeting the client's request.

Context in SOCAM is delivered to clients via two methods: push and pull. An API is provided so that clients are able to query the context provider for context information they are interested in (pull); alternatively, clients could subscribe to the provider for notifications when events occur (push).

SOCAM possesses many of the same problems present within CMF. Allowing clients to freely query the context providers presents a serious security risk if there is no process handling private data from other clients. There must also be a process by which clients who are new to the system can gain access to the provider's API to perform their queries. Finally, the lack of rich context requests formation (for notification pushes) prevents clients from establishing highly flexible and personalized notification requests.

2.3.3. Context-Awareness Sub-Structure (CASS)

Compared to CMF and SOCAM, CASS [21], by Patrick Fahy and Siobhan Clarke, is a somewhat simpler context-aware server-based middleware used to support context-aware applications on small mobile devices. The main goal of CASS middleware is to separate resource-intensive context reasoning from the context-aware applications that usually run on mobile devices. The middleware's architecture is capable of supporting large numbers of context sources, keeping a history of acquired context and converting raw context source data to more useful high-level context. CASS's middleware is event-based and abstracts the process of context acquisition from the various sources. However, unlike CMF and

SOCAM, CASS does not provide context clients with the ability to query the context provider and pull the context information they are interested in. Figure 2-5 demonstrates the CASS middleware architecture.

Context updates arriving from sensors are received by the SensorListener (located within the middleware) while context stored within the database is retrieved by the ContextRetriever for delivery to clients. As with the *SensorListener* within CASS's middleware, ChangeListener within context clients (mobile devices) listens for notifications of context changes and events from the context provider.

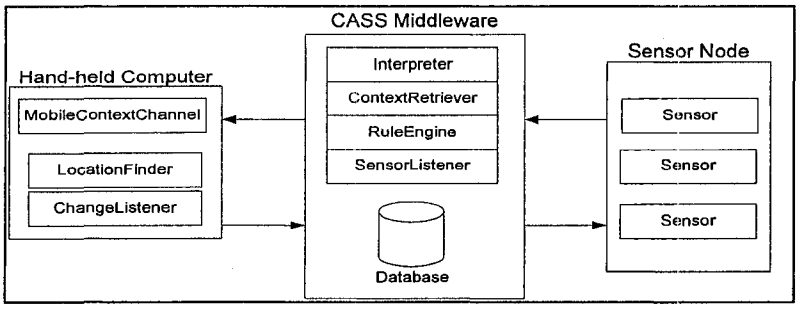


Figure 2-5. CASS architecture [21]

A major disadvantage of CASS is the apparent lack of an ontology for context modeling. The database used for context storage is server-based, but Fahy and Clarke did not fully explain how the stored context should be modeled. It is not clear how applications on mobile devices are made aware of the table structures used within the SQL-based database, or how queries are made for context data within the database. CASS's middleware architecture is therefore missing many of the important features that context-aware middleware is expected to provide – namely, a standardized approach to modeling and sharing context knowledge and a well-defined process through which clients can request or query their context information needs.

2.3.4. Hydrogen

Most of the context-aware middleware and architectures presented thus far depend on a centralized approach to context awareness. However, the Hydrogen [22] architecture proposed by Hofer et al. avoids this approach by introducing a distributed solution. Since the Hydrogen architecture is designed for use within every mobile device interested in acquiring context information, the Hydrogen framework had to be lightweight due to the restrictions

imposed by the devices' limited resources and processing power. The framework had to be able to support connections to remote sensors, since small mobile devices are unable to sense all of their contexts themselves. The Hydrogen firmware located on each device had to be able to share its context knowledge with other devices so that could increase its context acquisition abilities.

The Hydrogen architecture shown in figure 2-6 is divided into three layers: the Adapter Layer, the Management Layer, and the Application Layer. The Adaptor layer is responsible for acquiring physical context information from the sensors and delivering it to the Management layer. Within the Management Layer, a context server stores all incoming contextual information about the current environment of the device, and shares this knowledge with other devices by using peer-to-peer communication.

In keeping with the approach followed by other context-aware middleware, Hydrogen's context server gives applications access to its stored context information through a pull-based or a subscription/notification-based system. The former is performed by applications querying the context server for their needed context, while in the latter system; applications are informed of context changes by the context server. Hofer et al. justified their distributed approach by claiming that having all three layers of the architecture located on one device resulted in an a more robust approach to network disconnections. They also claimed that applications would not have to deal with remote context servers, and instead only communicate with the internal server through an XML-protocol or a set of serialized Java objects.

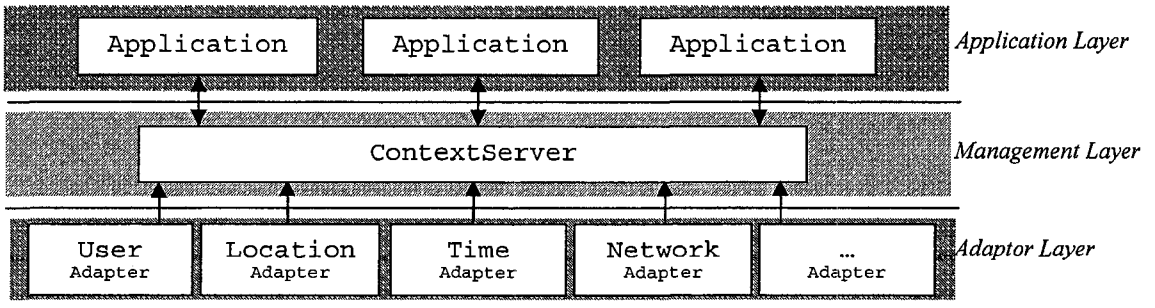


Figure 2-6. Hydrogen architecture [22]

However, the Hydrogen context-aware system architecture lacks two important components for any context-aware system: an ontology on which context information is modeled, and a

protocol by which context is shared between context servers located on different devices. Context within Hydrogen was limited to saving current time, the current location of the devices, the devices' identifier and type, the users' names, and information about available network connections. Hofer et al. had realized that the presence of ontology to model the vast amount of context available within a mobile environment is of high importance, and had set one of their future tasks to use CC/PP as the base for their context model. Unfortunately as already explained in section 2.1.2, CC/PP faces many limitations. A more useful model should therefore be chosen.

The missing protocol for sharing context between devices is of particular interest here since we are proposing a context-level negotiation protocol for establishing Context-Level Agreements. Hofer et al. had suggested this problem be explored in future research. This said, using the system architecture and protocol we describe later could greatly improve the Hydrogen architecture, by establishing Context-Level Agreements (CLA) between context servers located on each mobile device such that each device could acquire context information it would not otherwise sense with its limited onboard sensors.

2.3.5. Gaia

Gaia [23] is a distributed middleware infrastructure whose purpose is to export services. These services permit applications to access and query Gaia's services and stored context such that user-centric and context-aware applications can be developed. Gaia thus acts as a coordinator between software entities and network devices, provides services related to location, context and events, and stores information related to the active space controlled by the Gaia kernel. Gaia is divided into three components: the Gaia kernel, the application framework, and the applications.

The kernel is composed of a 'component management core (CMC), as well as five basic services used by Gaia applications: Event Manager, Context Service, Presence Service, Space Repository, and Context File System. Components and applications are created and destroyed, loaded and unloaded, and transferred dynamically by the CMC.

The event manager service plays the role of a context provider, distributing events in the active space. Events are forwarded to consumers (applications) that have registered with an event channel as the event manager employs a suppliers-based, a consumers-based, and a

channels-based decoupled communication model. A separate channel is created for each type of event, by the events' sources and applications can simply 'tap' into these event channels to view changes occurring within the system. The decoupling of consumers from suppliers increases the system's reliability, since context sources' failure does not necessarily entail the failure of the dependent application.

The context service is connected to several context providers (e.g. physical sensors) that allow applications to query and register for particular context information. The list of available context providers is maintained within a registry so that applications can search for their desired source of context. Unlike many existing middleware, the stored context does not use ontologies to model its context information. Context is modeled by using first-order logic and Boolean algebra, where context is represented by four arguments: `Context (<ContextType>, <Subject>, <Relater>, <Object>)`. ContextType is a reference to the context being described, Subject is the entity (person, place, etc) with which the context is concerned, Relater is a comparison operator, and Object is the subject's associated value. Using this first-order predicate, the temperature within room '3231' can be described as:

```
Context (temperature, room 3231, is, 98 F)
```

New context can also be inferred using quantification, implication, conjunction, disjunction, or predicate negation – for instance inferring the activity taking place in a room based on the number of people and applications running, as in the example below:

```
Context(number of people, room 2401,>,4)
AND Context(application, Powerpoint, is, running)
→ Context (social activity, room 2401, is, presentation).
```

Context information is presented to applications through the Context File System (CFS) in the form of directories. The path components represent context types and values where '/' is equivalent to '=='. Therefore, "/location:/RM2401/situation:/meeting directory" will return the files associated with context location ==RM2401 && situation==meeting.

Although Gaia's architecture was built for a context-aware ubiquitous environment, yet it is not suitable for an environment characterized by mobility. Queries for context information within the CFS required clients (applications) to be aware of the the CFS's directory structure before clients can find the correct path to their needed context. This mainly stems

from the absence of an ontology on which to model the context information. Mobile devices are consequently unaware of Gaia's kernel directory structure, which requires a process through which CFS's directory is transmitted to them using a shared protocol. Since no such protocol exists within Gaia, we can speculate that all clients/applications are developed exclusively for Gaia's enclosed environment (Active Space). The ability of applications to freely query Gaia's context directory presents a security risk, since private information acquired through sensors becomes publicly available to all applications in the Active Space. Because Gaia was developed for applications within an enclosed space, public access to information does not present a problem within the system, but expanding the concept to include mobility will require a review of Gaia's system architecture.

2.4. Comparison Between Our Approach And Available Context Models And System Architectures

In light of this background information on modeling context information, and using a range of approaches that includes CSCP, CC/PP, ORM, cues, rule-based models and ontologies, we have chosen an OWL-based context model for our negotiation-based context-aware system architecture. Our decision is based on the fact that the OWL language provides a simple approach to portraying context knowledge in terms of classes, properties, and relationships. This ontology could be extended to define unique individuals and properties, as well as the ability to reason about existing modeled context information. We also consider the fact that OWL has a more expressive format than other existing ontology languages, and that it simplifies the exchange and sharing of knowledge.

Indulska's CC/PP extensions expanded the vocabulary of CC/PP by a number of component-attribute trees related to modeling network trees, application requirements, location, and various relations and dependencies. However, the underlying use of CC/PP makes the process of capturing contextual relationships becomes highly complex and un-intuitive. The standard CC/PP has a restricted ability to merge and override separated ontologies; a process greatly needed in context-aware systems, since different context sources tend to share their knowledge with each other.

Held's Comprehensive Structured Context Profiles (CSCP) employed the RDF/S syntax language and was capable of addressing the merging and overriding issues within CC/PP.

Unfortunately though, Held restricted his goal to a predefined set of profiles that would need to be created prior to any client-provider communication. These profiles did not go beyond describing the capabilities of the devices, networks, and users capabilities, along with some simple context values and relationships.

The ORM graphic-based model and other graphical models share the ability to describe the structure of context knowledge in terms of both human readability and relational models. The merging of relational models can reduce the performance of systems that might use ORM for context modeling as columns are joined ineffectively resulting in spending unnecessary CPU and I/O operation time in rows that will be filtered out later on. Therefore, mobile devices limited in their computing abilities are not suitable for graphical context models.

Cues and other object-oriented context models provided great handling for incomplete or ambiguous information. They also simplified the process of accessing the needed context information by providing well-defined interfaces but posed a problem since some of the context representation formality was lost through encapsulation. Similar to graphic-based models, object-oriented models require fast processors and large resources - a combination that is usually unavailable within mobile devices.

Ontologies provide an easy method for sharing knowledge and context information between entities within a ubiquitous environment. Ontologies are also strong in normalization and formality, and are easily extended by ontologies shared between different context sources. Building on and utilizing the OWL-based models shown in section 2.1, our ontology-based context model derives four ontology categories: Global Skeletal Ontology, Global Concrete Ontology, Client Views, and Client Profiles. We will further describe these concepts and their purposes in Chapter 3.

The Global Skeletal ontology could be seen as the domain-independent knowledge representation model within all domains or high-level ontologies presented in COMANTO and SOCAM. Each context provider may exist within different domains or even similar domains. Each Provider may therefore have its own personal subset of the Global Skeletal Ontology. Each of these subsets (or sub-domain ontologies) is instantiated to create the context provider's Global Concrete Ontology.

Client Views are further subsets of the Global Concrete ontologies, describing the clients' context access rights within each provider. Views are hybrids of the providers' sub-domain ontologies and their Global Concrete Ontologies.

Clients' profiles borrow some of the ideas underlying CC/PP where profiles describing the abilities of the applications' and services' were presented in an xml-based language. Similarly, clients' Profiles are OWL-based documents that contain information such as the clients' identities, types, abilities, version, and so on; any information that could be used by the providers to determine the fittest View to provide Clients with.

The context-aware system architecture we are proposing within this thesis is not meant as a replacement for many of the existing middleware architectures already presented in this chapter. Instead we are proposing an architecture that could be potentially incorporated into these context-aware architectures in order to improve their context dissemination methods, to meet the context consumers' needs through context-level negotiations, and to make current architectures usable within ubiquitous mobile environments.

Because it was divided into context types and values, CMF's weak RDF context model lacked any relationship representation between context concepts, and was limited to a small set of context classes. Restricting clients to two methods of context acquisition from the middleware - Request/Response and Subscription/Notification - is not a flexible approach to meeting clients' needs, which might be influenced by time, actions taken by clients, or changes of context information.

SOCAM's middleware architecture had problems similar to CMF's, but with one additional problem: the ability of clients to pull context information from the provider. This problem occurs because clients are able to query the context provider's context database by using first-order logic. Two problems consequently arise. The first is clients' freedom to query any information that may be restricted or private. The second problem is related to mobility. The fact that clients can query the provider's database entails their prior knowledge of the database's structure. We can therefore assume that all clients are built and executed in the same enclosed space as the context provider, or that a protocol must be used to provide clients with the database's structure as it changes over time. Sharing such structures is not an

intuitive process and requires a detailed description of the database's design - something that was not provided by the authors.

Moreover, CASS lacked an ontology for context modeling and relied on an SQL database to store and model its context information. The same problems within SOCAM were thus introduced again with CASS.

In addition, Hydrogen did not provide an ontology for context modeling and did not provide a clear approach to sharing context information between the various context servers on each device that was a member of the system. The benefits of using an ontology have already been exhaustively detailed earlier. Hofer has admitted that there should be a process by which context servers can share their context knowledge but did not propose a possible solution. We propose that by using our ontology-based context-level negotiation protocol to establish Context-Level Agreements, servers within Hydrogen can easily share their context knowledge amongst themselves.

Finally, there were numerous problems within Gaia's architecture. The first problem is as in most other architectures shown, the lack of an ontology for context modeling. The second problem concerns the method of context access. Accessing context information through directories limits clients to those directories built specifically for the Gaia architecture, and any changes to the directory within Gaia means that the client's design should also change.

In light of these shortcomings in the existing architectures and models, we propose a new context-aware system architecture where clients can be built independently from the context provider's design. Context exchange is based on a context-level negotiation protocol based on shared ontologies, and the enforcement of the clients' context needs is achieved through Context-Level Agreements (CLAs). Our protocol provides clients with the ability to describe their context needs freely by using the provided ontologies. Context providers can also share context knowledge with each other by playing the dual role of context clients and context providers.

2.5. Chapter Summary

In this chapter, we have presented some of the available context modeling methods suggested by researchers in the field of context awareness. Many of the proposed models

lacked the ability to model complex context relationships while others were extensively general and abstract, thus complicating the process of creating domain-specific ontologies for any context-aware systems being designed.

In chapter 3 we will present our method of building an OWL-based ontology that provides ontology users with an ontology that is richer in its content and extension abilities than those seen in [17] and [18]. We will also illustrate the ontology's usage in order to develop domain-specific ontologies as well as Client Profiles and Views.

In light of the advantages and disadvantages of the context-aware middleware and architecture discussed here, we will introduce our own context-aware system architecture in chapter 4. In chapter 5 we will introduce our context-level negotiation protocol that can be used to establish Context-Level Agreements to dynamically provide Clients with the context information they require.

Chapter 3

An Ontology for Modeling Context Information

3.1. Chapter Objectives

Context is the information used to describe the situation of entities present within an environment whether these entities are human users, physical objects, or logical objects. Context is any information that can be sensed, inferred, or measured and might be of interest to a context consumer (Client) within a context-aware system. There is an insurmountable amount of context existing in our environment. This said, there should be a process by which this context knowledge can be easily stored, accessed, manipulated, and shared. Knowledge within context-aware systems is usually exchanged between different entities for adaptation purposes. These adaptations are performed through reasoning over the received context information in order to improve the functionality of the context consumer (Client).

Context Clients come in many forms. A Client could be an application that changes its graphical interface based on the device's remaining resources, or according to the preferences of the current user. A Client could also be a service provider that adapts its services according to its users' current situation, current system status, or changes in the surrounding physical or virtual environment. Or, mare simply, a Client could be another context Provider acquiring its context knowledge from nearby context Providers within the system, in order to further expand its own knowledge base and to share that context knowledge with other entities. The possibilities for context usage are endless, and there is a need for that knowledge to be easily shared. Thus context models should be not only *light*, but also *sharable*.

The development of context Providers is usually separated from that of context Client, due to two reasons:

1. *Separation of Interest*: Context Providers are usually components specialized in acquiring context information, modeling it, using it to infer higher-level context, storing it, and disseminating the resulting context to interested consumers. Unlike context

Providers, Clients tend to have an interest in acquiring only a small subset of the available context information in order to improve their functionalities or tomeet the users' interests. For this reason, Clients and Providers are usually developed separately. This separation of interest can be visualized through the general hierarchical structure of most context-aware systems, as shown in figure 3-1a. Three system levels exist, and each level has a unique set of modules with unique responsibilities (figure 3-1b). The *Sensor-Adaptor Layer* contains all sensors within the system. These sensors could be physical (motion detectors, temperature sensors, noise sensors, etc...), or logical (sensors monitoring the network bandwidth, or the remaining memory space in a server). Sensor readings differ in their formats according to the type of sensors used, the manufacturer, or the delivery protocol for readings delivered over the network. There is therefore a need for modules that can translate the sensor readings' raw data into a format that is recognizable by the above *Context Management Layer.*

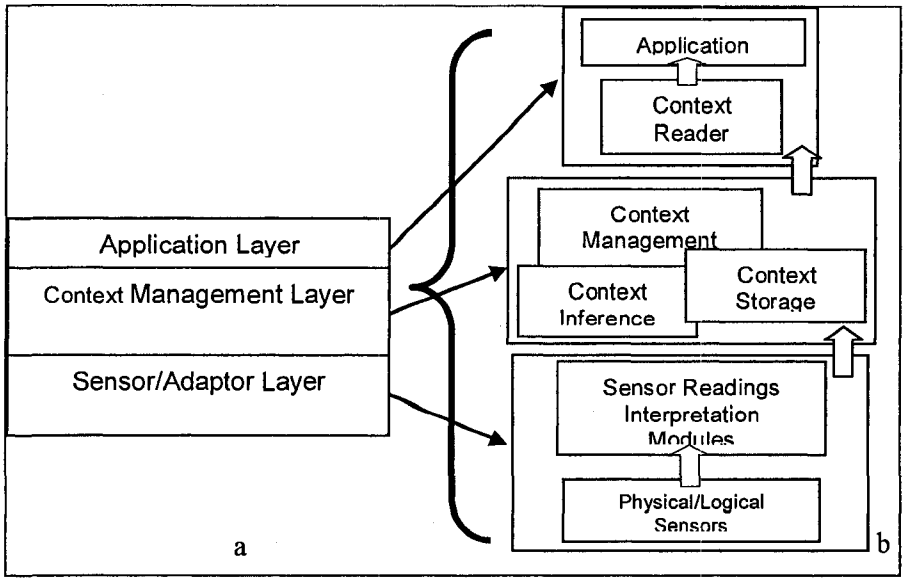


Figure 3-1. General Structural Levels of Context-Aware Systems

Context data received by the management layer from the Sensor/Adaptor Layer could be used to produce more useful context information through aggregation, inference or composition. The *Context Management Layer* is also the level at which context information is stored for use by the Application Layer. The storage contains updated values for context information being collected, and it may also store old context values for future reference or to infer new context.

The uppermost level, the *Application Layer*, is the layer interested in retrieving context information stored within the *Context Management Layer* for personal use or delivery. The *Context Reader* component listens for incoming context from the management level and delivers that context to all applications or services that have requested it.

Each layer within the general architecture of context-aware systems has its own set of interests and concerns. The distinction between interests allows us to distinguish the development of context providers from that of context consumers.

2. *High Management Costs*: Since the Context Management Layer receives context data from the Sensor Layer, context managers must communicate with possibly thousands of sensors embedded within the environment. This context data is stored, used to infer higher level context, and retrieved so it can be shared with multiple Clients (e.g. different context-aware applications). Mobile devices on which context-aware applications are usually running have restricted communication capabilities, slow CPUs, and limited storage space. These limitations make it impossible for mobile devices to perform all the Context Management Layer responsibilities. Also, to have each context-aware application communicating directly with the needed sensors would result in exclusive locking of system resources, since serial interfaces can only be accessed from one application at a time. Consequently, resource-intensive processes such as communication with context sources, acquiring and inferring context, and storing large amounts of context information should be moved to more capable components. It is for this reason the Context Management Layer is usually located on components that have large storage space, fast processors, and strong communication capabilities.

Separating the components that acquire, infer, and store context from components that use context brings with it an important requirement: a unified context modeling approach. The model is used to allow context Clients (e.g. applications) and context Providers (context management layer) to share their context knowledge independently from their specific designs and implementations.

In chapter 2, we described the various approaches used to model context information. The modeling approach used should have a high level of formality and the ability to represent the complex relationships between different context concepts. The model should be easily sharable and support the distributed composition of ubiquitous systems. We also indicated

that we have chosen an OWL-based ontology to model all our context information within our context-aware architecture and in our context-level negotiation protocol. We employ that ontology at four different levels within our architecture:

- Global Skeletal Ontology
- Global Concrete Ontology
- Client Views
- Client Profiles

The remainder of this chapter is dedicated to describing the details of each of these levels before introducing our system architecture in chapter 4.

3.2. Global Skeletal Ontology (GSO)

The Global Skeletal Ontology, or GSO, represents the base onto which all context information within the context-aware system should be modeled. As seen in COMANTO [17] and SOCAM [18], ontologies were divided into an abstract high-level ontology and a lower domain-specific ontology. Using this concept of domain separation, our Global Skeletal Ontology can be seen as our modified version of the Upper or High-Level Ontology presented in both of these works.

The approach adopted by Tao Gu et al. and Strimpakou et al. in their high-level ontologies was to create the most abstract set of classes at the high-level ontology and to leave the remaining details for the domain-specific ontologies. While creating the upper ontology is an easy task, however, designing domain-specific ontologies becomes more cumbersome, since many of the missing concepts and relationships must be created from scratch.

In [18], the high-level context ontology was restricted to a set of classes: Person, Place, Preferences, Agenda, Activity, Time, Physical Object, Sensor, Service, Network, and Legal Entity. No specific details concerning subclasses and properties descending from these high-level context classes were provided. Only a limited list of inter-class relationships were shown in this high-level ontology as was seen earlier in 2-3. A similar ontology was presented in [17] where the upper ontology defined the basic concepts of person, location, computational entity and activity. All context relationships and properties were left for the domain-specific ontologies to determine and model.

We have managed to extend these upper ontologies to create our own version of the upper ontology, or what we call the Global Skeletal Ontology (GSO). Within this GSO we attempted to contain the majority of contextual concepts that could possibly exist within any context-aware environment. Therefore many of the classes that existed within [17] and [18] were imported and expanded; their hierarchical organization was reordered to allow for easier extensions within the low-level domain-specific ontologies, and an extensive list of inter-class relationships and properties was defined to ease the development of domain-specific ontologies.

Figure 3-2 shows the topmost classes within our GSO. We will express each class's details in the following subsections. All Object properties (properties representing the relationships between classes) and Data type properties have also been omitted in this figure for practical purposes, and will also be presented in further detail within the following subsections.

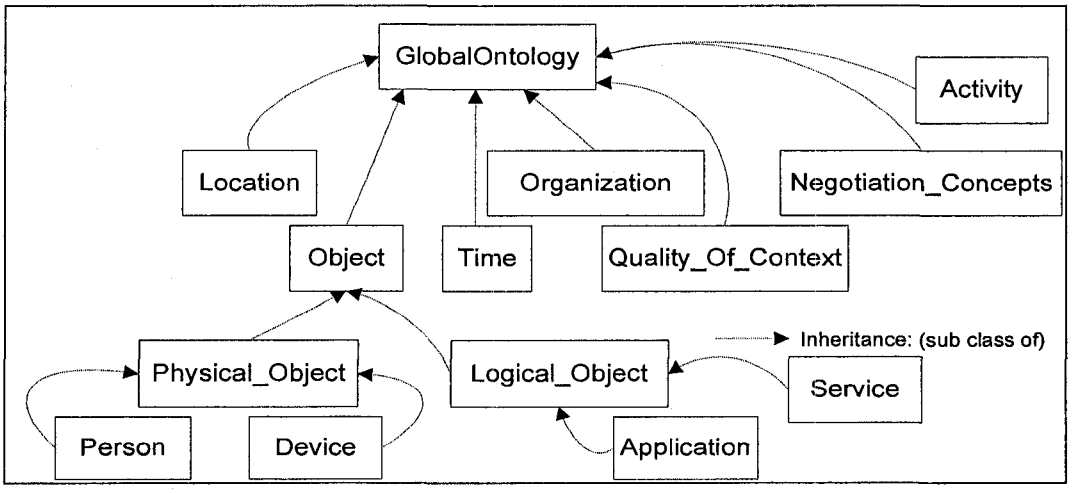


Figure 3-2. Topmost classes of the Global Skeletal Ontology

The *GlobalOntology* class represents the root from which all context classes and properties extend. It provides an entry point for declaring all domain-specific ontologies. Seven major classes extend from the *GlobalOntology* class, each representing a major concept needed within context-aware systems:

- *Location*
- *Object*
- *Time*
- *Organization*
- *Quality_Of_Context*
- *Negotiation_Concepts*

• and *Activity*

The Object class is further divided into *Physical_Object*, *Logical_Object*, and *Atmospheric_Object*.

3.2.1. Location Ontology

The most important context information within context-aware systems is location, and the majority of early context-aware systems utilized location-awareness as the exclusive type of sensed context for application adaptability. The increased mobility of users following the introduction of ubiquitous computing made knowledge of the user's location an essential element in adapting to the users' needs.

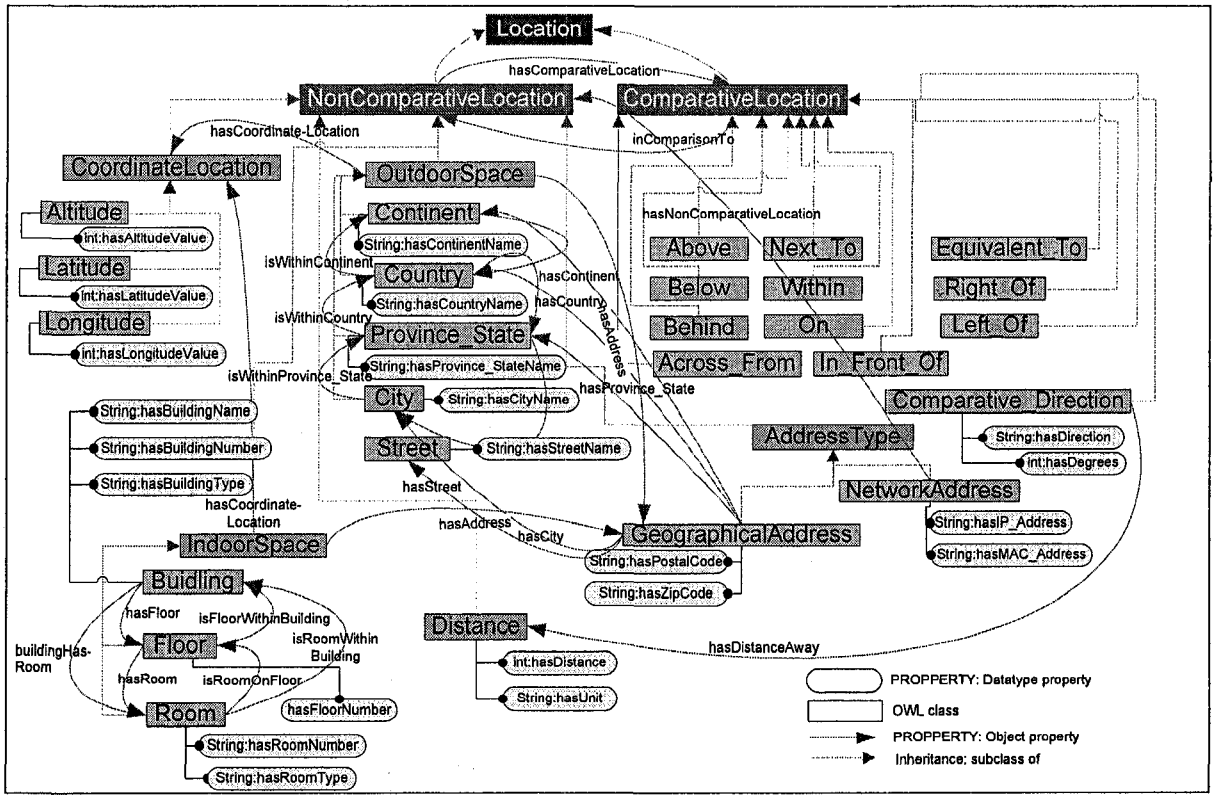


Figure 3-3. Location Ontology

Using knowledge about location, the user's intended actions could be inferred. Therefore given the location a great deal of context information can be induced, such as the activity taking place or the service to be provided. If a group of work colleagues is located within the same area, it can be inferred that there is a meeting taking place, and services such as overhead boards and PowerPoint slides can be presented to the users since most meetings require these services.

Most ontologies presented do not provide any details on how a location ontology should be designed, and researchers assumed this problem would be solved at the level of domain-specific ontology. Conversely, we believe that at least the most commonly used location-describing concepts should be included within the upper ontology (GSO). To make intuitive the process of defining location within context-aware systems, we have divided the Location ontology shown in figure 3-3 into two subclasses: *ComparativeLocation* and *NonComparativeLocation.*

3.2.1.1. NonComparativeLocation

Locations present within our physical surroundings that could be can be classified as subclasses of the *NonComparativeLocation* class. The rooms of a house within which a user is located, a device's GPS location, and the address of a particular building are all non-comparative locations because they are not described in relation to other nearby entities. We have divided non-comparative locations into four classes: *OutdoorSpace*, *IndoorSpace*, *CoordinateLocation*, and *AddressType*.

The *OutdoorSpace* class refers to any spaces beyond the boundaries of a building. Parks, cities, states, and countries all fall within the *OutdoorSpace* class category. Context-aware systems that monitor the movement of entities in the external environment can use the provided subclasses (*Continent*, *Country*, *Province_State*, *City, and Street*) or extend *OutdoorSpace* by providing their personal sub-classes to facilitate the systems' abilities to describe an item's external location. Many context-aware systems utilize the external location of entities to adapt services and applications. GPS systems located on vehicles are an example of such systems. For instance, as drivers move through city streets in their vehicles, the context-aware systems within the vehicles could use that information to notify drivers about nearby stores and shopping malls.

The use of *OutdoorSpace* subclasses is not only limited to the current, future, or past location of entities within the system, but is also extended for use within the *AddressType* subclass. The widely accepted form of representing addresses includes references to the country, city, street, and postal code of the unit being described. All these elements are grouped within the *GeographicalAddress* class. To emphasize the reusability of concepts that are present within our ontology, we provided several object-type properties that link the *GeographicalAddress*

class to concepts already provided within the *OutdoorSpace* class in order to describe the geographical addresses of buildings. Thus, *hasCountry*, *hasProvince_State*, *hasCity* and *hasStreet* refer to the country, province or state, city, and street information that the address contains. Additional information, such as the postal or zip codes are contained within their respective data-properties, *hasPostalCode* and *hasZipCode,* which are directly connected to the *GeographicalAddress* class.

Network devices have a unique representation for addresses. Any device connected to a network usually has an IP address used for identification and communication, as well as a MAC address attached to each network adapter. These concepts were grouped in the *NetworkAddress* class. Since each device that has an IP or MAC address is also located within a corresponding physical location, such as a server room or a mobile device, we have provided a *hasNonComparativeLocation* relationship that permits a network address to relate to other types of location whether they are external (as was described earlier for *OutdoorSpace)*, or internal (in *IndoorSpace)*.

Most early context-aware systems have been restricted to enclosed spaces because of financial and technological limitations. Consequently, there should be an ontology that permits the representation of indoor spaces. The *IndoorSpace* class has been divided into three subclasses representing the three general levels into which indoor spaces are divided: *Building*, *Floor*, and *Room*. Object-type properties have been provided to model the relationships between these three subclasses. A floor can contain a number of rooms, while a room belongs to a floor. Similarly, a floor belongs to a building, while a building contains a number of floors. The *IndoorSpace* class has also been given a relationship to the *GeographicalAddress* class since any building's address would have information about the street and city it belongs to.

We have included a useful location representation within our ontology: *CoordinateLocation.* This could be used to represent the global position of any entity along the *Latitudes* and *Longitudes*, as well as the entity's elevation in the *Altitude* subclasses.

The classes and properties we provided in the Location ontology within our Global Skeletal Ontology do not constitute an exhaustive list of possible representations. Designers responsible for building a particular context-aware system are required to reuse this provided

ontology and to extend it wherever they see a necessity that would fit their personal domain-specific ontologies.

3.2.1.2. ComparativeLocation

The way in which we describe locations in everyday life does not entail the low-level format seen within the *NonComparativeLocation* class in section 3.2.1.1. Instead, we often refer to the location of objects in relation to other nearby objects. For example, we could say that a user, 'Tom', is currently standing 'next to' another user, 'Marry'. Alternatively, we could say that 'Tom's office' is 'across from' the 'coffee room'. We have provided twelve possible choices for modeling comparative locations (*Above*, *NextTo*, *Below* ...). These high-level relative representations have all been grouped into the *ComparativeLocation* sub-class.

A unique *ComparativeLocation* subclass is the *Comparative_Direction*. We often refer to the relative location of an entity in terms of direction and distance, in number of degrees from another entity. For example, a coffee shop could be located 'North, 30° west, and 50 meters' from a mobile user. This relative location can facilitate human readability and clarity, while GPS readings must be converted to a format (usually graphical) comprehended by average users. We have therefore provided a *Comparative_Direction* subclass. The *hasDirection* property is used to indicate the direction towards which the object is located, such as north, south, east, west, northeast etc... The *hasDegrees* property is used to indicate the number of degrees the object is located away from the reference point e.g. 'south, ***50°*** east'. Knowledge of direction is sometimes insufficient without knowledge of distance. For this reason we also included a distance subclass used to indicate the distance value (*hasDistance*) and representation units (*hasUnits*) used through the *hasDistanceAway* relationship.

3.2.1.3. Location Example

In figure 3-4, we provide an example of the use of the Location ontology. The figure illustrates the use of comparative and non-comparative location models to represent the location of a room with a number - '*B502*' being used as a *research laboratory*. Through the *isRoomWithinFloor* property, a relationship is established between the room and an instance of the *Floor* class with floor number = '*5*'. A comparative relationship exists between the

fifth floor and another floor numbered = '*4*' using an instance of the *Above* class. This specifies that the fifth floor is the floor *above* the fourth floor. The *isFloorWithinBuilding* places both floors within the "Colonel By" building that represents its address by exploiting the *GeographicalAddress* class. This class states that the room is located at 161 Louis Pasteur Street, in the city of Ottawa, in the Canadian province of Ontario.

```
<Room rdf:ID="Room_B502">
  <hasRoomNumber rdf:datatype="&xsd;string">B502</hasRoomNumber>
  <hasRoomType rdf:datatype="&xsd;string"
  >Research Laboratory</hasRoomType>
  <isRoomWithinFloor rdf:resource="#Fifth_Floor"/>
</Room>
<Floor rdf:ID="Fifth_Floor">
  <hasFloorNumber rdf:datatype="&xsd;int">5</hasFloorNumber>
  <isFloorWithinBuilding rdf:resource="#ColonelBy"/>
  <hasComparativeLocation rdf:resource="#Above_1"/>
</Floor>
<Above rdf:ID="Above_1">
  <inComparisonTo rdf:resource="#Fourth_Floor"/>
</Above>
<Floor rdf:ID="Fourth_Floor">
  <hasFloorNumber rdf:datatype="&xsd;int">4</hasFloorNumber>
  <isFloorWithinBuilding rdf:resource="#ColonelBy"/>
</Floor>
<Building rdf:ID="ColonelBy">
  <hasBuildingType rdf:datatype="&xsd;string"
    >Engineering Building</hasBuildingType>
  <hasBuildingName rdf:datatype="&xsd;string">Colonel By</hasBuildingName>
  <hasBuildingNumber rdf:datatype="&xsd;string">161</hasBuildingNumber>
  <hasAddress rdf:resource="#CBY_Address"/>
  <buildingHasRoom rdf:resource="#Room_B502"/>
</Building>
<GeographicalAddress rdf:ID="CBY_Address">
  <hasPostalCode rdf:datatype="&xsd;string">K1N 6N5</hasPostalCode>
  <hasCity rdf:resource="#Ottawa_City"/>
  <hasStreet rdf:resource="#Louis_Street"/>
  <hasCountry rdf:resource="#Canada"/>
  <hasProvince_State rdf:resource="#Ontario_Province"/>
</GeographicalAddress>
<Street rdf:ID="Louis_Street">
  <hasStreetName rdf:datatype="&xsd;string">Louis Pasteur</hasStreetName>
  <isWithinCity rdf:resource="#Ottawa_City"/>
</Street>
<City rdf:ID="Ottawa_City">
  <hasCityName rdf:datatype="&xsd;string">Ottawa</hasCityName>
  <isWithinProvince_State rdf:resource="#Province_State"/>
</City>
<Province_State rdf:ID="Ontario_Province">
  <hasProvince_StateName
  rdf:datatype="&xsd;string">Ontario</hasProvince_StateName>
  <isWithinCountry rdf:resource="#Country"/>
</Province_State>
<Country rdf:ID="Canada">
  <hasCountryName rdf:datatype="&xsd;string">Canada</hasCountryName>
</Country>
```

Figure 3-4. Location ontology example

3.2.2. Time Ontology

Context-awareness cannot be separated from time. Actions, events, and acquired context are all time-dependent. Timestamps used to mark important events, such as the arrival of sensor readings, the start and end times of user activities, and the representation of birthdates and appointments, all require the presence of a time ontology. Entities within a pervasive computing environment should have a unified method for representing and understanding any exchanged context information that is time-related. We have therefore extended the concepts presented in [17] to model our time ontology (figure 3-5).

The numeric representation of date and time used in international communication is specified in the International Standard ISO 8601 [24]. Following this standard is advantageous in computer and application usage, and prevents the confusion or misunderstanding that can result from various existing notations.

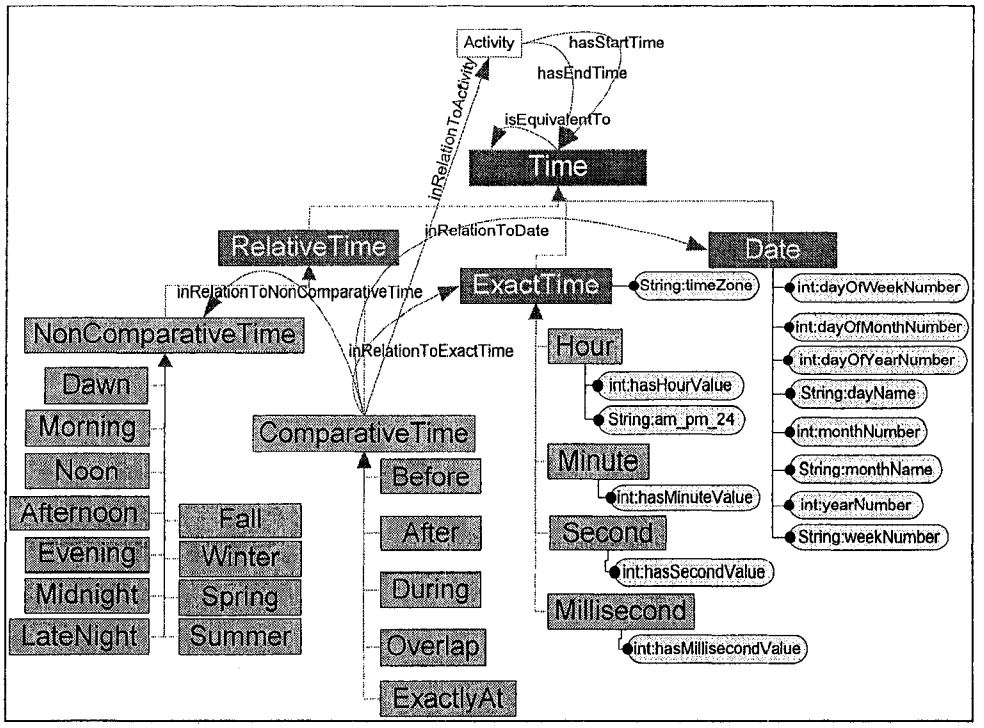


Figure 3-5. Time ontology visualization

The time ontology in figure 3-5 dedicates two subclasses, *Date* and *ExactTime*, to providing the necessary concepts on which to model any date or time information. The international standard date notation is YYY-MM-DD, and is represented in our ontology by the *yearNumber*, *monthNumber*, and *dayOfMonthNumber* properties of the Date class. Commercial and industrial applications often require a reference to the week of a year such as 2008-W02 meaning the second week of the year 2008. This industrial use of week references is directly related to the weekly rotations of manufacturing shifts. The *weekNumber* property serves this purpose.

The ISO 8601 standard also provides the notation for representing the time of day: hh:mm:ss. Separate subclasses for the hour, minute, and second have each been provided with properties that exhibit their value representations. Since time is usually represented according to the local time zone, the *timeZone* property is used to indicate the time zone associated with the indicated time, such as EST for Eastern Time and UTC for the Universal Time. Other properties such as *dayName* and *monthName* were added as well, to provide system designers more flexibility in terms of modeling time through different formats.

However, using the standard time representation format within ubiquitous systems does not always meet all user requirements. Human users tend to refer to time as a more abstract concept than that represented by the international standard. For example, the user 'Tom' could say that he has an appointment this 'afternoon'. The concept of 'afternoon' is understood to be between 12:00 p.m. and 6:00 p.m. but there is no reference to the exact time this appointment will take place. Such loose concepts of time have been grouped into a subclass of *Time* called *NonComparativeTime* which includes *Dawn*, *Morning*, *Evening*, *Fall*, *Summer*, and references to various time spans.

Loose references to scheduled events, to exact time representations, even to loose time representations are also possible within context-aware environments. A meeting could be scheduled to take place 'after' an earlier meeting scheduled for 5:45 p.m. The duration of the first meeting is unknown, thus the second meeting could have an infinite set of possible start times making the use of exact time representation inefficient. This comparative time concept has been modeled within the *ComparativeTime* class allowing activities' exact times and dates to be compared to each other loosely through comparative concepts such as *'before'*, *'after'*, *'during'*, amongst others.

```
<Course rdf:ID="CEG_3180">
    <hasStartTime rdf:resource="#ceg3180_StartHour"/>
    <hasStartTime rdf:resource="#ceg3180_StartMinute"/>
    <hasStartTime rdf:resource="#ceg3180_StartSecond"/>
    <hasDate
</Course>
<Course rdf:ID="CEG_4150">
    <hasStartTime rdf:resource="#After_25"/>
</Course>
<After rdf:ID="After_25">
     <inRelationToActivity rdf:resource="#CEG_3180"/>
 </After>
<Hour rdf:ID="ceg3180_StartHour">
    <timeZone rdf:datatype="&xsd;string">EST</timeZone>
    <hasHourValue rdf:datatype="&xsd;int">2</hasHourValue>
    <am_pm_24 rdf:datatype="&xsd;string">p.m.</am_pm_24>
</Hour>
<Minute rdf:ID="ceg3180_StartMinute">
    <hasMinuteValue rdf:datatype="&xsd;int">30</hasMinuteValue>
</Minute>
<Second rdf:ID="ceg3180_StartSecond">
    <hasSecondValue rdf:datatype="&xsd;int">0</hasSecondValue>
</Second>
<Date rdf:ID="yousif_birth">
    <yearNumber rdf:datatype="&xsd;int">1983</yearNumber>
    <monthNumber rdf:datatype="&xsd;int">5</monthNumber>
    <dayOfMonthNumber
rdf:datatype="&xsd;int">30</dayOfMonthNumber>
```

Figure 3-6. Time ontology example.

3.2.2.1. Time Example

The time ontology example we provide in figure 3-6 illustrates the use of exact time, comparative time and calendar date time representations. A computer course offered at the university of Ottawa (CEG 3180) uses the *hasStartTime* property to proclaim that the course lecture starts at 2:30:00 p.m. EST. If the lecture's end time is not known, another course (CEG 4150) uses comparative time to declare that its lecture would start 'after' the CEG

3180 course. Birth dates are expressed using the *Date* subclass, as in the example showing May 30th, 1983 as the birth date of an individual within the system.

3.2.3. Object Ontology

Serving users and adapting services to their needs is one of the most important goals of any context-aware system. The presence of an ontology on which to model users is therefore highly important. The process of modeling all possible users in all context-aware systems is a difficult task and requires prolonged research on the way human societies function and how they are structured. For this reason, we have restricted our ontology to a set of limited OWL classes and properties that model the general concept of the *Person* class. The most logical approach to modeling users' information is to do so according to the roles these users play within different domains. The needs and interests of users tend to change according to their role. If we assume that 'Tom' is currently in his house, then the context information he is interested in acquiring, or the context information he is permitted to access, may depend on his position within the family. For instance, parents are usually given more access to information than children, and residents are given more access than visitors. The same concept applies once 'Tom' moves out of his house domain and arrives at his university domain. His position within his own family becomes irrelevant in this new domain, and the system's interests shift to modeling his relationship with his university colleagues, professors, courses, departments, and university labs and offices. Figure 3-7 shows the *Person* class split into three domains: *HomeDomainUsers*, *UniversityDomainUsers*, and *CompanyDomainUsers*. Although context-aware systems require other domains, we are interested in providing a subset of only those which are most widely used.

There always exists a set of context properties that all users share regardless of which domain they belong to. All Individuals could be divided into the categories male and female (*String:hasSex*); all human users have first and last names (*String:hasFirstName, String:hasLastName*), all have a date of birth (*hasBirthDate*), most have relationships with other individuals (*knows*), and all users could have many other possible common properties.

Since reusability helps maintain an ontology and reduce the time spent creating ontology models from scratch, many classes have property relationships with different parts of the ontology. For instance, *hasBirthDate* links the *Person* class to the *Time* ontology,

ownerOfRoom links *Person* to the *Location* ontology, and so on. This reusability reduces the time needed and the complexity associated with creating ontology classes and relationships.

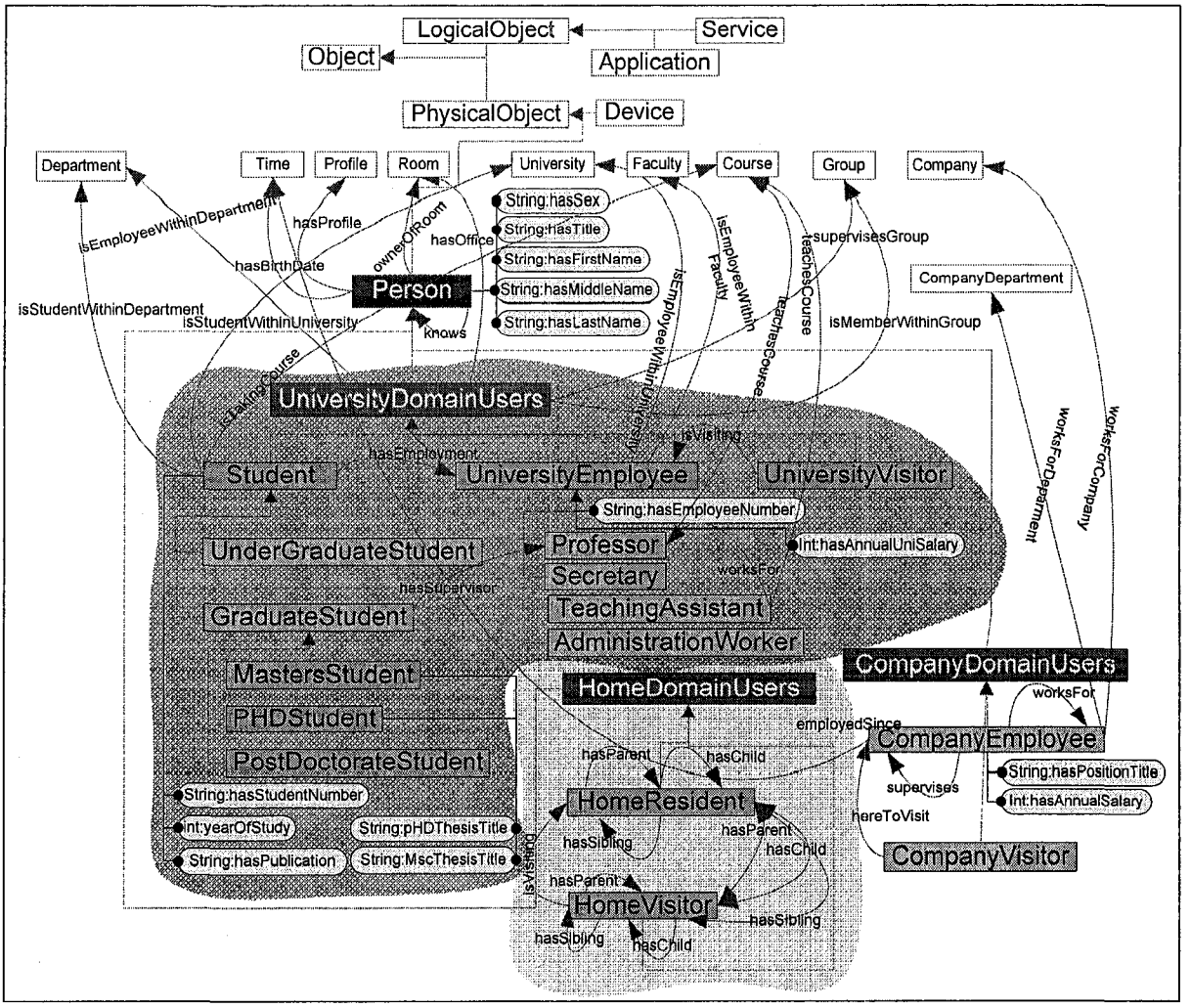


Figure 3-7. Person Ontology

The *HomeDomainUsers* subclass is used to model human users for context-aware systems enclosed within the domain of an individual home or of a set of interconnected homes. In most homes, context interests and accessibility rights are directly influenced by the hierarchical organization of the family; the parents are typically at the top, followed by the children according to their ages in descending order, while at the very bottom are visitors, who are usually only permitted access to a limited amount of context information. Consequently, only two subclasses were needed: *HomeResident* and *HomeVisitor*, and only four object properties were required: *hasParent*, *hasChild*, *hasSibling*, and *isVisiting*. All other context information can be inferred from these classes and properties. For example,

determining the identity of the father simply involves finding an individual who has a *hasChild* relationship to another resident or visitor, and who has *hasSex*= "male". Searching for other individuals with different roles entails a similar approach.

Building context-aware systems within university educational campuses requires modeling the different classes of users that could exist in the environment, as well as any other information that might be useful. In general, university users could be classified as students (*Student*), employees (*UniversityEmployee*), or visitors (*UniversityVisitor*), with some individuals sharing two current roles, such as graduate students employed as teaching assistants.

Students were divided into undergraduate and graduate students, and graduate students were further broken down into Master's, Ph.D., and postdoctoral researchers. This division is needed since access to many facilities within campuses differ for undergraduate and graduate students, and some of the information required for graduate students such as the presence of a supervisor[*hasSupervisor*] or thesis topic [*String:MscThesisTopic, String:PhDThesisTopic*] are not necessary for undergraduate students. We have provided a general ontology model for users within a university domain.

The third user domain category we provide is related to company employees. Users within a company domain can be divided into employees who work within the company's buildings or visitors on these premises. The relationship between employers and employees is represented by the *supervises* and *worksFor* relationships. Other properties could be intuitively expected, such the company and department for which an employee is working (*worksForCompany*, *worksForDepartment*), the date the employee was hired (*employedSince*), or their job title (*String:hasPositionTitle*).

As seen in figure 3-7, the *Person* class and its respective domain-specific subclasses utilize a set of relationships to external classes such as *Location*, *Organization* (*Company*, *University*, *Department*), *Time*, *Profile*, and others. The *Person* class we have provided does not provide a complete representation of human users in a context-aware environment; rather our attempt was meant to provide a generalized description of 1) the approach that should be taken when designing a Person's ontology and of 2) the minimum set of properties and relationships that should be present in such ontology. It is important to break the Person ontology into

different sections representing the domains in which clients can exist. Users in each domain should be categorized according to the roles they play in that domain. In addition, data type properties and relationships must be extended from those present in the high-level ontology to model the interaction of users in different roles.

3.2.3.1. Person Example

The Person ontology example used in figure 3-8 shows a possible scenario of individuals within a university domain. Defining a student within the master's program requires the creation of an instance of the *MastersStudent* class such as the "*Yousif*" instance provided here. The example provides a description of a Master's student within the University of Ottawa, displaying the student's full name, thesis title, year of study, and relationships to other individuals (such as his supervisor "*Karmouch*", and another Masters student, "*Ismaeel*", whose details were omitted). The Person example shows that Yousif has an office located in Room_B502, an indoor location whose details were provided earlier in figure 3-4, in the Location example. The example also shows that the birth date example presented within the Time ontology in figure 3-5 is reused to represent Yousif's date of birth.

```
<MastersStudent rdf:ID="Yousif">
    <hasMiddleName
    rdf:datatype="&xsd;string">Kadhim</hasMiddleName>
    <hasFirstName rdf:datatype="&xsd;string">Yousif</hasFirstName>
    <hasStudentNumber
    rdf:datatype="&xsd;int">2984293</hasStudentNumber>
    <MscThesisTitle rdf:datatype="&xsd;string"
    >Ontology-Based Negotiation Protocol And Context-Level
    Agreements</MscThesisTitle>
    <hasLastName rdf:datatype="&xsd;string">Al Ridhawi</hasLastName>
    <yearOfStudy rdf:datatype="&xsd;int">2</yearOfStudy>
    <hasSex rdf:datatype="&xsd;string">Male</hasSex>
    <isStudentWithinDepartment rdf:resource="#electrical_engineering"/>
    <hasOffice rdf:resource="#Room_B502"/>
    <isStudentWtihinFaculty rdf:resource="#graduate_faculty"/>
    <hasSupervisor rdf:resource="#Karmouch"/>
    <isStudentWtihinUniversity rdf:resource="#univeristy_ottawa"/>
    <hasBirthDate rdf:resource="#yousif_birth"/>
    <isMemberWithinGroup rdf:resource="#IMAGINE"/>
    <knows rdf:resource="#Ismaeel"/>
    <engagedInActivity rdf:resource="#ELG_5460"/>
</MastersStudent>
<MastersStudent rdf:ID="Ismaeel"/>
<Professor rdf:ID="Karmouch">
    <hasFirstName rdf:datatype="&xsd;string">Ahmed</hasFirstName>
    <hasLastName rdf:datatype="&xsd;string">Karmouch</hasLastName>
    <hasSex rdf:datatype="&xsd;string">Male</hasSex>
    <isEmployeeWithinDepartment rdf:resurce="#electrical_engineering"/>
    <hasOffice rdf:resource="#A_508"/>
    <supervisesGroup rdf:resource="#IMAGINE"/>
    <isEmployeeWithinUniversity rdf:resource="#univeristy_ottawa"/>
</Professor>
```

Figure 3-8. Person ontology example

3.2.4. Organization and Activity Ontologies

The roles that users play within context-aware systems tend to reflect their apprentice to certain organizations. A client having the role of a professor is a member of a university educational organization, as are the students. Meanwhile, an engineer in a Research and Development laboratory is usually a member of a private I.T. business organization.

Therefore, an important concept within the Global Skeletal Ontology should be to provide an ontological method for modeling organizations. To reflect the university and company sub-domains of the *Person* ontology, we have extended two sub-classes from *Organization*: *EducationalOrganization* and *BusinessOrganization.* We have limited our organization ontology to these two classes, but further classes should be extended by system developers according to their domain-specific needs.

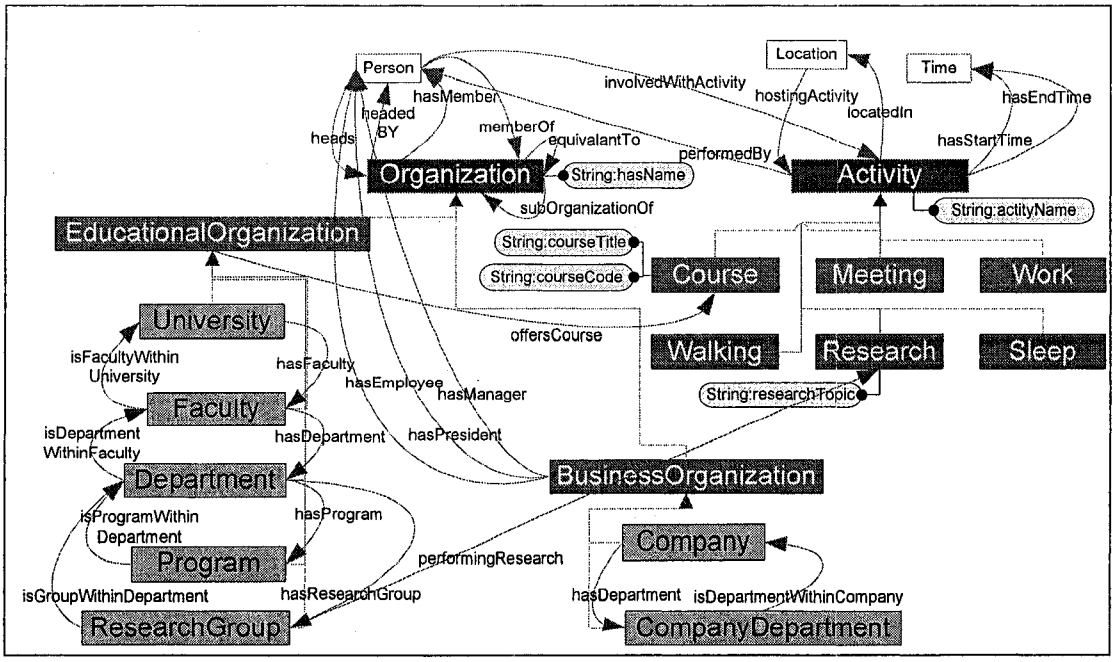


Figure 3-9. Organization and Activity ontologies

Educational organizations include elementary schools, high schools, colleges and universities, but we have also limited this subclass to universities. As we proceed to describe our GSO, it becomes evident that developing ontologies capable of modeling every aspect of our environment is a complex task in need of extensive research and connecting ontologies developed by various researchers to create a complete ontology model. To create our Organization ontology we have therefore reused the concepts presented in [25] which in turn used the machine-readable SHOE form.

The *EducationalOrganization* class modeled the usual breakdown of universities into faculties, departments, and programs, as well as the presence of research groups. The *BusinessOrganization* class represented the division of companies into various departments as well as the different roles existing within companies such as employees, managers, and

presidents. Organization ownership and membership were represented by the object-type relationships *heads/headedBy* and *memberOf/hasMember* relating the Organization to the Person class.

Context-aware systems exist within continuously changing environments. Changes in the environment could be affected by activities taking place by users or devices. Also, changes in the surrounding context may affect current or future activities. Hence, our Global Skeletal Ontology must also include a model for activities. Since activities, organizations, and users are closely interconnected within context-aware environments, we have illustrated these interconnections within figure 3-9.

```
<University rdf:ID="univeristy_ottawa">
  <hasName rdf:datatype="&xsd;string">The University Of
  Ottawa</hasName>
  <hasFaculty rdf:resource="#graduate_faculty"/>
</University>
<Faculty rdf:ID="graduate_faculty">
  <hasName rdf:datatype="&xsd;string">Faculty Of Graduate and
  Postgraduate Studies</hasName>
  <isFacultyWithinUniversity rdf:resource="#univeristy_ottawa"/>
  <hasDepartment rdf:resource="#electrical_engineering"/>
</Faculty>
<Department rdf:ID="electrical_engineering">
  <hasName rdf:datatype="&xsd;string">Department of Electrical and
  Computer Engineering</hasName>
  <hasMember rdf:resource="#Yousif"/>
  <hasMember rdf:resource="#Ismaeel"/>
  <isDepartmentWithinFaculty rdf:resource="#graduate_faculty"/>
  <hasResearchGroup rdf:resource="#IMAGINE"/>
  <hasLocation rdf:resource="#within_cby"/>
</Department>
<ResearchGroup rdf:ID="IMAGINE">
  <hasName rdf:datatype="&xsd;string">Intelligence for Mobile
  Autonomic and Cognitive Networks Laboratory</hasName>
  <hasMember rdf:resource="#Yousif"/>
  <hasMember rdf:resource="#Ismaeel"/>
  <headedBy rdf:resource="#Karmouch"/>
  <isGroupWithinDepartment rdf:resource="#electrical_engineering"/>
</ResearchGroup>
<Within rdf:ID="within_cby">
  <inComparisonTo rdf:resource="#ColonelBy"/>
</Within>
```

Figure 3-10. Organization ontology

We have provided a list of possible activities, and the list could be expanded to include other activities depending on the developer's needs. The key concepts needed for activities are linked to the *Time*, *Person* and *Location* classes. Most activities start and end at pre-set, inferred, or unknown times, therefore requiring relationships to the Time class. Events may take place at different locations, which explains the need for the *locatedIn* and *hostingActivity* relationships with the *Location* class. Users participating or hosting events utilize the *involvedWithActivity* and *hostedBy* properties.

3.2.4.1. Organization Example

In figure 3-10, we provide a simplified example of an organization ontology for a university environment. The ontology describes the relationship between the University of Ottawa, the Faculty of Graduate and Postgraduate Studies, the Department of Electrical and Computer Engineering, and the IMAGINE research group. The IMAGINE research group is shown as

having three members one of whom heads the group (Dr. Karmouch). Use of the comparative location ontology is also visible in the "Within" class relating the department's location to the CBY building whose address was shown earlier in figure 3-4.

3.2.5. Quality of Context Ontology

The inconsistencies stemming from dynamicity and sensor imperfections within acquired context information in pervasive systems requires a model for representing the quality of the context that the system supplies to its clients. For this reason, we reused a quality of context ontology provided in [18] that was adequate for modeling any needed quality of context concept. Context classes within the Global Skeletal Ontology can be associated with instances of the *QualityParameter* class, such as *Accuracy* or *Freshness*. Each parameter establishes one or more relationships with a *QualityMetric* class that allows describing the value, unit, and type of the context parameter quality. A trouble scenario may rise when measurements performed on physical objects have a margin of error. This error could be modeled by creating an instance of the ErrorMargin parameter class and associating it with an instance of the QualityMetric class with *hasUnit*=centimeter, *hasType*=integer, and *hasValue*=5.

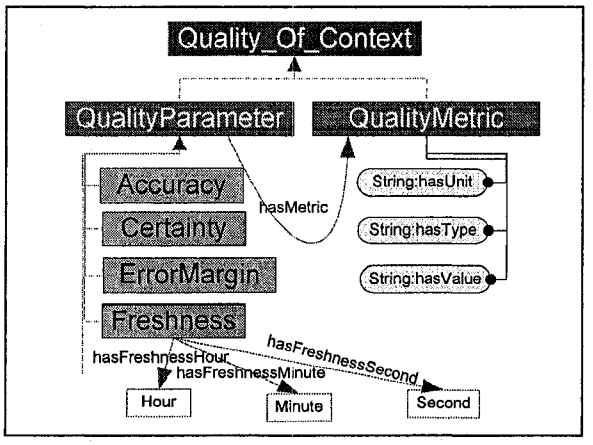


Figure 3-11. Quality of Context ontology

3.2.6. Negotiation Concept

Negotiating entities within a ubiquitous context-aware environment require an ontology capable of modeling concepts that are key to their context-level negotiation protocol. Since the details of our proposed context-aware system architecture and ontology-based protocol will not be discussed until chapters 4 and 5, we illustrate in figure 3-12a two important phases in our negotiation protocol, both of which occur prior to context negotiations. These are the *Profile* exchange and *View* exchange phases. As mentioned in section 3.1, ontologies within our system are used in our Global Skeletal Ontology, Global Concrete Ontology, Client Profiles, and Client Views. Since we had indicated that the Global Skeletal Ontology

(GSO) represents the base on which all context information is modeled, providing a model for *Profiles* and *Views* within the GSO is not an exception.

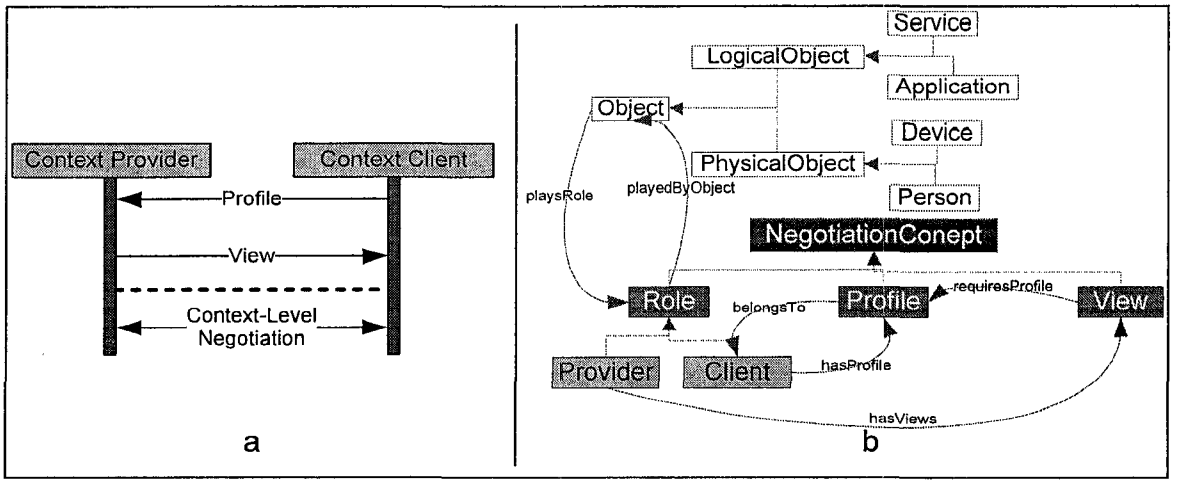


Figure 3-12. a) Three main stages of context negotiation. b) Negotiation Concepts ontology

Within the Action ontology, our GSO indicated that objects within a context-aware environment may take any physical or logical form such as that of person, device, service or application. A Context Client within a context-aware system may itself be a human user, a video application, or a mobile device. Similarly, a context Provider could be a context-providing service located on a device. Therefore, context Clients and Providers are themselves *Objects* within the system. What differentiates these objects from other objects in the system is the former's ability to play a negotiation *'role'*. If the object is providing context information to consumers, then it is playing the role of a context *Provider*. Otherwise, if the object is consuming context information, then it is said to be playing the role of a *Client*. These concepts are displayed within the ontology of figure 3-12b.

Client *Profiles* are our solution to Client identification. They are used by Clients to supply context providers with enough information for them to make a decision concerning what context information these Clients should be permitted to access. Profiles do not introduce any new concepts within the GSO. Clients simply reuse concepts present within the GSO to supply context Providers with as much or as little personal information as they wish. If the Client was a human user, then classes and properties present within the Person ontology are used to create the profile, and if the Client was an application, then classes and properties

within the Application ontology are used. Further details and examples of Client Profiles are provided in section 3.4.

Clients' interests in context information differ depending from Client to Client. They may also differ in their access rights. This means that context information accessible by one set of clients may not be accessible to others for privacy and security reasons. Client *Views* are OWL-based documents derived from the Global Skeletal Ontology and the Global Concrete Ontology (section 3.3) informing *Clients* of the context information they may access. As part of the negotiation process, Clients are provided with Views that supply them with enough information to shape their context requests. These Views allow Clients to avoid the unnecessary process of requesting information only to receive a rejection from the context Provider because the information was inaccessible in the first place. Client Views will be further discussed in section 3.4.

The *NegotiationConceps* class within our ontology is used to indicate that objects can participate in context-level negotiations within a context-aware system by playing different *Roles*, where Clients supply Providers with their *Profiles* and where Providers supply Clients with their respective *Views*.

3.2.7. Summary

In section 3.2, we have provided a detailed description of our Global Skeletal Ontology (GSO). We believe that the development of useful context-aware systems requires the establishment of universally accepted ontologies that permit the modeling and sharing of knowledge. The current unorganized formation of multiple ontologies by research groups that differ in their structures and organization requires that all these efforts be combined into a unified global ontology model applicable to all fields of study. Most existing global ontologies are either designed to work exclusively in small domains, or are too abstract to provide system designers with any useful knowledge for effective domain-specific ontologies.

In this section we have attempted to provide a solution to this problem by combining and expanding many of the existing ontologies to create a global ontology that is not restricted to specific domains and less abstract than the current global ontologies. We are aware that our ontology is not exhaustive but could be modified and extended to reach a complete ontology.

In summary, the high-level ontologies proposed in [17] and [18] are unnecessarily abstract, and they do not provide ontology designers with an initial set of helpful concepts to extend for their domain-specific ontologies. The objective of our GSO is to pave the path to our proposed ontology-based context-level negotiation enforced system architecture and protocol, and to provide designers of context-aware systems with a larger set of ontology concepts, thus simplifying the task of system development.

3.3. Global Concrete Ontology (GCO)

Context-aware systems differ in their interests, responsibilities and capabilities depending on their designs and the domains within which they exist. These differences entail the existence of differences between the ontologies that each context-aware system uses to model its acquired context information. Context management entities represented by context Providers communicate with various context sources to collect context information, model it, and distribute its knowledge to interested consumers. The Providers' knowledge bases are continuously updated with new context, and Context-Level Agreements (CLAs) established with Clients in the system may be affected by changes within the knowledge base.

Evidently, modeling context within context Providers requires ontologies that are suited to the situation or domain at hand. Accordingly, the generic nature of the Global Skeletal Ontology (GSO) described in section 3.2, makes this ontology normally too generic for use within the provider's specific domains - hence the need for domain specific ontologies. We refer to these domain-specific ontologies as Global Concrete Ontologies (GCO).

To create GCOs, context Providers begin by retrieving a subset of the Global Skeletal Ontology. This step is necessary because GSOs cover a wider area of knowledge than is necessary for most Providers. A context Provider within a hospital domain does not usually need to collect context information about users within a university's domain. This separation of interests shows the absence of concern the hospital's context provider has within its GCO for the *UniversityDomainUsers* extension to the *Person* ontology presented in section 3.2.3. The first step to creating a GCO is therefore to retrieve a useful subset from the GSO for use within the particular context Provider's domain.

Although the GSO should cover a wide range of context concepts, relationships and properties, its context-modeling capabilities are broader than they are deep. Particular

ontology classes needed by the context Providers may be missing, along with interclass relationships and properties. For this reason, the second step to creating a Provider's GCO is

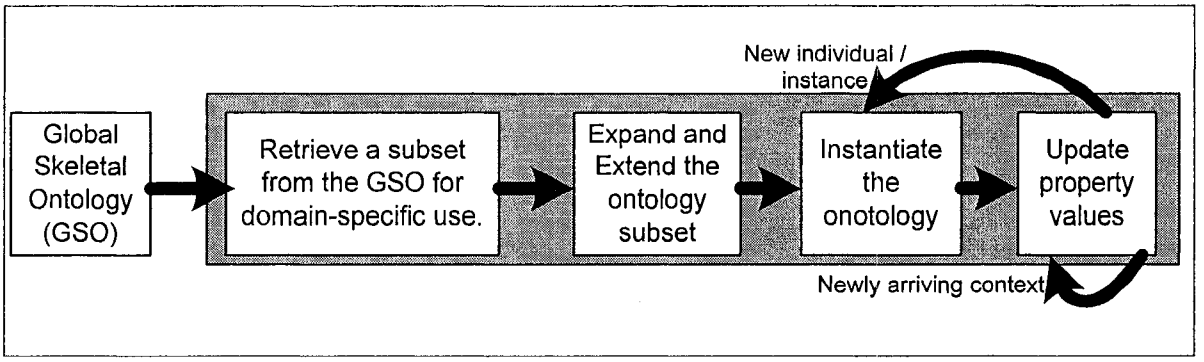


Figure 3-13. Global Concrete Ontology lifecycle

to extend the retrieved GSO subset by adding all the necessary classes, interclass relationships, and datatype properties.

Once the structure of the GCO has been defined according to the context Provider's needs, the ontology is instantiated for each individual being modeled within the system. Using the same hospital domain, instances for each doctor, nurse, patient, visitor - or any other form of users - are created. An instance for each room whose context information is collected is also created, along with instances for all necessary relationships and properties. In summary, an initial set of instances from all needed classes, interclass relationships, and class properties is derived from the Provider-specific Global Concrete Ontology.

The derived instances are used to store up-to-date context values acquired from sensors, hardcoded values, or values derived through inference. These values are stored in their respective formats (strings, integers, floats, etc...). The combination of figures 3-4, 3-6,3-8, and 3-10 constitutes a possible instantaneous Global Concrete Ontology. Instances created for Master's student users like "Yousif" represent individuals within the context Provider's domain of interest. All respective relationships and properties of that individual, whose values

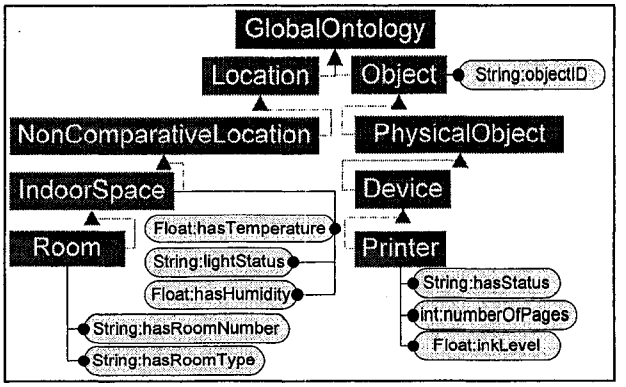


Figure 3-14. Possible GCO

the context provider can acquire are placed within the GCO and updated as new values arrive.

The presence of a context Provider within a small office room illustrates the GCO in a simple way. The system is able to monitor the state of various environmental conditions within the room, such as temperature, humidity, and light status. We can assume that there are two printer devices whose properties are also sensed, such as their ink levels, the remaining number of sheets within the trays, and the printers' statuses (on, off, ready, in use, etc...). This simple example provides important insight into the steps for creating a Global Concrete Ontology.

The first step involves expanding the Global Skeletal Ontology. Since printers are electronic devices, one would expect a subclass of the Device class named Printer. Unfortunately though, this subclass does not exist, therefore, the context Provider must extend the GSO to include the needed Printer class and all necessary properties related to the ink level, number of sheets, and status. The Location class of the GSO in section 3.2 provided information about indoor spaces, such as rooms, but had no properties related to environmental conditions like light and temperature. Thus further extension to that part of the ontology is required within the GCO. In figure 3-14, we provide a visual representation of a possible GCO for this example.

```
<Room rdf:ID="OfficeRoom">
<hasTemperature rdf:datatype="&xsd;float">23.0</hasTemperature>
<lightStatus rdf:datatype="&xsd;string">ON</lightStatus>
<hasHumidity rdf:datatype="&xsd;float">78.0</hasHumidity>
<hasRoomNumber rdf:datatype="&xsd;string">A2</hasRoomNumber>
<hasRoomType rdf:datatype="&xsd;string">Office</hasRoomType>
</Room>
<Printer rdf:ID="Printer1">
<objectID rdf:datatype="&xsd;string">A2_1</objectID>
<hasStatus rdf:datatype="&xsd;string">READY</hasStatus>
<numberOfPages rdf:datatype="&xsd;int">703</numberOfPages>
<inkLevel rdf:datatype="&xsd;float">65.0</inkLevel>
</Printer>
<Printer rdf:ID="Printer2">
<objectID rdf:datatype="&xsd;string">A2_2</objectID>
<hasStatus rdf:datatype="&xsd;string">BUSY</hasStatus>
<numberOfPages rdf:datatype="&xsd;int">34</numberOfPages>
<inkLevel rdf:datatype="&xsd;float">36.0</inkLevel>
</Printer>
```

Figure 3-15. Instantaneous GCO

Once the domain-specific ontology has been defined, the next step requires instantiating the ontology. Two instances of the Printer class are needed in this case - one for each printer – as well as a single instance of the Room class, for the one office room being used. Once all instances have been created, the context Provider starts updating the values of each instance's properties based on context information arriving from sensors. Since property values are dynamic, figure 3-15 includes a sample instantiated GCO at a particular point in

time. The figure displays the context Provider's updated knowledge about the office's environment.

3.4. Client Views (CV)

3.4.1. Description of Views

Context Clients differ from each other in terms of the context information they have interest in acquiring. Differences between Clients are not limited to context information they wish to receive, but could include limitations within their context access rights. Dissolving sensors and computing devices into the environment was a key goal of ubiquitous computing. The large amount of context information collected through these sensors may include private data that certain context Clients may not wish to share with other Clients due to privacy and security concerns. Therefore, Clients' access to context information stored within context Providers should not be treated equally. Each Client should only be permitted to access a limited set of context information.

Client Views are our solution to solving this problem in context-aware systems. These Views provide Clients with a description of the context information Providers can serve them, based on defined criteria that each Client should meet prior to being supplied with a View. Each View can be restricted to a single Client or it may be shared by a set of Clients bundled into groups.

Both Global Skeletal Ontology and Global Concrete Ontology are integrated to the formation of Client Views. The Views permit context Clients to gain knowledge about their accessible set of context information stored within the Provider and form their context requests when negotiations between the two parties commence. To derive Client Views, two steps are required by the context Provider. First using the privately-formed domain-specific ontology retrieved from the Global Skeletal Ontology, the context provider retrieves another ontology subset limited to the classes, relationships, and properties the considered context Client is allowed to access. The concepts included within this ontology subset are only high-level classes and properties shared by all class instances the Client is permitted to access. The second step is to retrieve the details of these classes and properties.

Once the initial ontology model has been retrieved, all present classes whose *individuals* are accessible by the client are instantiated. These instances are used by the Clients to perceive the set of individuals whose context information the Clients could access. All context information accessible to each individual of the same class is equal. That is, if two instances of *Device* class were present, one for device X and one for Y, then context classes and properties accessible from X would also be accessible from Y. This first level of accessible classes, properties, and individuals is only a general high-level representation of the Client's accessible context. This initial set of ontology classes, properties, and individual instances is grouped into a *Root View* OWL-based document.

It is often the case that accessible context information for individual instances from the same ontology class is not equivalent. Thus, the second step in Client Views' composition is to extend each individual provided in the *Root View* whose accessible context information was not completely listed. For instance, the Root View may have indicated that the Client was permitted access to activity statuses and battery levels for devices' X and Y, but if the Client could also access information about X's current user but not Y's, then this information would not have been indicated in the Root View. The necessary information is instead shown in a second document detailing all accessible ontology classes, relationships, and properties particular to the individual at hand in this example, X. Like the Root View document, the newly extended document contains a list of accessible individuals particular to X's accessible context information.

For each individual listed in the Root View's document and for whom only partial accessible classes, properties and individuals were included, a separate document is created. The process is repeated for each individual within the new documents and their respective sub-documents until all accessible context information has been clearly defined for the particular Client View.

Client Views do not include any context values like the exact temperature value of an accessible room, or the number of sheets remaining in a printer tray. Like any un-instantiated ontology, Views only contain the definitions for particular accessible context information, along with a list of accessible individuals.

3.4.2. View Example

Following is an example that clarifies our concepts of Client Views. Sensors are embedded in two rooms of a house that collect context information. The sensed context is acquired by a context Provider located within the home's domain which provides this domain's context information to interested Clients. For practical purposes we will assume that all Clients are treated equally and have access to the same context information - Hence all Clients are provided with the same View for their negotiation. The Root View document shown in figure 3-16 contains all the context classes and properties the Client has access to, such as *NonComparativeLocation*, *IndoorSpace*, and *Room*.

The document also lists the rooms (individuals) that are accessible: *Room_1* and *Room_2*. Since those rooms belong to the same class, then all accessible properties in the Root View and related to the Room class are also accessible by both individuals. Thus, the room number, room type, and room temperature constitute information that the Client could access for both of those rooms.

Context information accessible by the Client for Room_1 but not Room_2 would need to be included within a separate document that is particular to that individual. Figure

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.owl-
ontologies.com/GlobalOntology.owl#"
   xml:base="http://www.owl-
ontologies.com/GlobalOntology.owl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="GlobalOntology"/>
<owl:Class rdf:ID="Client_View">
  <rdfs:subClassOf rdf:resource="#GlobalOntology"/>
</owl:Class>
<owl:Class rdf:ID="NonComparativeLocation">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>                                          Accessible
<owl:Class rdf:ID="IndoorSpace">                      classes
  <rdfs:subClassOf rdf:resource="#NonComparativeLocation"/>
</owl:Class>
<owl:Class rdf:ID="Room">
  <rdfs:subClassOf rdf:resource="#IndoorSpace"/>
</owl:Class>
<owl:DatatypeProperty rdf:ID="hasRoomNumber">
  <rdfs:domain rdf:resource="#Room"/>
  <rdfs:range rdf:resource="&xsd;string"/>          Accessible
</owl:DatatypeProperty>                             properties
<owl:DatatypeProperty rdf:ID="hasRoomType">
  <rdfs:domain rdf:resource="#Room"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasTemperature">
  <rdfs:domain rdf:resource="#Room"/>
  <rdfs:range rdf:resource="&xsd:float"/>
</owl:DatatypeProperty>
-------------------------------------------------------------
<Room rdf:ID="Room_1"> </Room>                    Accessible
<Room rdf:ID="Room_2"></Room>                     individuals
</rdf:RDF>
```

Figure 3-16. Root View document for Client

3-17 illustrates the second document within the Client View by listing context information accessible to the Client for Room_1.

Light levels and pressure levels are context information the Client could access only for Room_1, and since the accessible context for Room_2 has already been listed in the Root View document, there is no need for other documents to list Room_2's accessible context. Clients interested in Room_1's context information are also given access to a television device with ID="Television_A". In keeping with the steps taken to access extra context within Room_1, Television_A also needs a View document that details all the accessible classes, properties and relationships particular to that television set.

Once a Client receives its complete set of View documents, this Client knows what context information it may or may not request from context Providers, allowing negotiations to proceed smoothly via Context-Level Agreements.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.owl-
ontologies.com/GlobalOntology.owl#"
   xml:base="http://www.owl-
...
<ViewDocument rdf:ID="Room_1"></ViewDocument>
<owl:Class rdf:ID="GlobalOntology"/>
  <owl:DatatypeProperty rdf:ID="hasLightLevel">
    <rdfs:domain rdf:resource="#NonComparativeLocation"/>
    <rdfs:range rdf:resource="&xsd;float"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasPressureLevel">
    <rdfs:domain rdf:resource="#NonComparativeLocation"/>
    <rdfs:range rdf:resource="&xsd;float"/>
  </owl:DatatypeProperty>
  <owl:Class rdf:ID="Device">
    <rdfs:subClassOf rdf:resource="#Physical_Object"/>
  </owl:Class>..
  <owl:ObjectProperty rdf:ID="hasDevice">
    <rdfs:domain rdf:resource="#IndoorSpace"/>
    <rdfs:range rdf:resource="#Device"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="Object">
    <rdfs:subClassOf rdf:resource="#GlobalOntology"/>
  </owl:Class>
  <owl:Class rdf:ID="Physical_Object">
    <rdfs:subClassOf rdf:resource="#Object"/>
  </owl:Class>
  <owl:Class rdf:ID="Device">
    <rdfs:subClassOf rdf:resource="#Physical_Object"/>
  </owl:Class>
  <owl:Class rdf:ID="Television">
    <rdfs:subClassOf rdf:resource="#Device"/>
  </owl:Class>
</rdf:RDF>
-------------------------------------------------------------------------
<Television rdf:ID="Television_A"> </Television>
</rdf:RDF>
```

Figure 3-17. Second View document referring to accessible context for Room_1

3.5. Client Profiles

Views provided by context Providers to Clients reveal large amounts of private information concerning the context Providers' abilities and the Clients' access rights. Delivery of Views should therefore be limited to those Clients who posses the necessary rights to receive them.

However since access rights are system-dependent, there is no strict definition of what constitutes the "rights" of Clients. The definition of what Clients should possess in order to gain access to their respective Client Views is up to context-aware system designers.

In general, access to Client Views relies on two conditions:

1. *Availability of a fast and efficient user authentication method*: The number of malicious attempts to break into a system is directly proportional to the number of the system's users. Since context-aware systems contain large numbers of context sources and context consumers, the frequency of break-in attempts to the systems would be expectedly high. An efficient user authentication method would control access to context information and guarantee the identity of the accessing Client.
2. *A clear description of the Client*: Context information provided to Clients not only depends on the Client's identity or access specifications, but also on the type of Client requesting the information and its capabilities. For instance, in context-aware applications, providing users with a graphical map of a city's nearby restaurants within a city only a limited set of context information is needed to meet the user's needs. This information could be the user's current location, a list of available restaurants within the city, the restaurants' locations vis-à-vis the user's location, prices for different menu items from each restaurant, and so on. Yet the user (here, the owner of the device on which the application is running) could be permitted access to a much wider range of context information than is necessary by the application itself. There should therefore be a method for describing the type and abilities of the context-requesting Client other than those of the owner of the Client.

Authentication is used to protect sensitive information, including private context information collected within a context-aware system. Any authentication method used should have a high level of reliability and low false acceptance and rejection rates. The false acceptance rate is the percentage of successful authentication attempts by a person other than the correct individual, while the false rejection rate is the percentage of rejected authentication attempts performed by the correct individual. A number of approaches exist for authentication, including usernames and passwords, cryptographic smart cards, USB tokens, digital

certificates, and biometric techniques. However, privacy and security through authentication in context-aware systems is an ongoing research topic that is beyond the scope of this thesis.

Authentication alone does not meet the identification requirements of context-aware systems. A single individual in a Ubi-Comp environment is usually the owner of several context consumers, such as smart applications, mobile devices, and services. Thus providing the context Provider with the necessary information to clearly describe the type and capabilities of the negotiating Client is highly important in order for Providers to make the correct decision as to which Client View should be delivered.

We suggest adapting the use of what we call *Client Profiles* in order to provide the necessary Client descriptions. Client Profiles that describe the type of Client needing context would inform context Providers of the Client's type and capabilities before making a decision as to what Client View should be delivered. There are no strict guidelines of what Profiles should or should not include but we suggest the following guidelines for context-aware systems in order to define their own domain-specific Client Profile structures:

- Must be derived from the Global Skeletal Ontology: as was indicated earlier, the GSO constitutes a publicly-accessible ontology that defines the way knowledge is modeled within any context-aware environment. Since Clients also have access to the GSO, it would be useful to include the definition of Profiles within the GSO.
- Should be OWL-based: Profiles are meant to be delivered by Clients to the context Providers within whom context-level negotiations are performed. Since the amount of data contained within Profiles could vary with the type of Client,

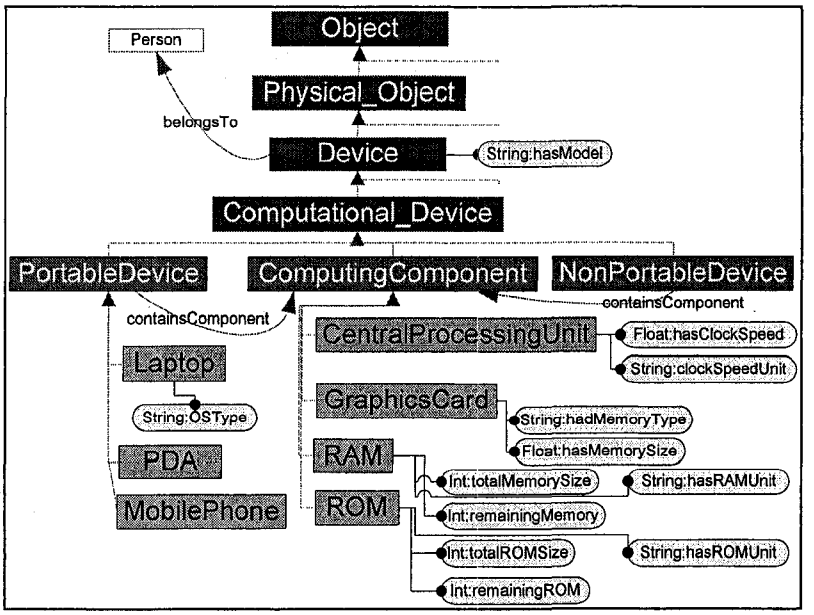


Figure 3-18. Computational Devices Ontology for both portable and non-portable devices

the format used must be light, must have syntax and semantics understood by computing entities (Providers), and must be easily shared. OWL-based ontologies were shown to possess all these attributes; hence, Client Profiles should also be based on the OWL language.

- Should protect the Clients' privacy: the amount of information Clients must submit within their Profiles should not risk their privacy. Profiles should only contain enough Client description as needed by the context Provider so the latter can make a decision as to what context information the Client may be interested in acquiring and is capable of handling.

A sample Client Profile could belong to a device owned by a user interested in negotiating its context needs with a context provider. Due to the wide range of devices currently existing we did not provide a formal model for devices within our Global Skeletal ontology. In this example, however, we provide here a simplified Device ontology restricted to laptops.

Devices exist in different forms such as personal computers, PDAs, televisions, DVD players, etc. In general, though, we divided them into two categories: Computing devices (*Computational_Device*) and non-computing devices (*NonComputational_Device*). Computing devices include any device capable of performing computations such as laptops, personal computers and PDAs, while televisions, VCR and similar devices were grouped under the NonComputational_Device category. The ontology displayed in figure 3-18 was restricted to computing devices since these are related to the example we are providing. Portable devices, such as laptops and PDAs include numerous components such as CPUs, hard disks, and sound cards. The same applies to non portable devices like personal computers - hence the division of computing devices into the PortableDevice, ComputingComponent, and NonPortableDevice classes.

The sample Profile in figure 3-19 belongs to a laptop computer owned by the same user individual, *Yousif*, described earlier in section 3.2.3.1. Each Profile belongs to a Client who is in turn represented by an Object. The object at hand is a laptop and the details pertaining to its capabilities are related to various factors such as the processor and operating system used, the amount of memory and disk space available and other internal computing capacities. These personal details related to the Client's identity and capabilities help the

context Provider deduce which Views are most fit for the Client with whom it is currently negotiating.

The Provider's View choice depends on the Client's identity, abilities, and expected interest. All decisions should be based on internal policies on the Provider's end.

```
<Profile rdf:ID="yousif_laptop_profile">
    <belongToClient rdf:resource="#Yousif_Client"/>
</Profile>
<Context_Client rdf:ID="Yousif_Client">
    <playedByObject rdf:resource="#Laptop_Yousif"/>
</Context_Client>

<Laptop rdf:ID="Laptop_Yousif">
    <hasDeviceName rdf:datatype="&xsd;string">Yousif_PC</hasDeviceName>
    <OSType rdf:datatype="&xsd;string">Windows Vista Home Premium SP2</OSType>
    <hasModel rdf:datatype="&xsd;string">HP Pavilion dv6500</hasModel>
    <belongsTo rdf:resource="#Yousif"/>
    <containsComponent rdf:resource="#yousif_CPU"/>
    <containsComponent rdf:resource="#Yousif_Graphic"/>
    <containsComponent rdf:resource="#Yousif_RAM"/>
    <containsComponent rdf:resource="#Yousif_ROM"/>
</Laptop>

<CentralProcessingUnit rdf:ID="yousif_CPU">
    <hasClockSpeed rdf:datatype="&xsd;float">1.9</hasClockSpeed>
    <hasModel rdf:datatype="&xsd;string">AMD Turion64 X2</hasModel>
    <clockSpeedUnit rdf:datatype="&xsd;string">GHz</clockSpeedUnit>
</CentralProcessingUnit>

<GraphicsCard rdf:ID="Yousif_Graphic">
    <hasMemoryType rdf:datatype="&xsd;string">Shared</hasMemoryType>
    <hasModel rdf:datatype="&xsd;string">NVIDIA GeForce 7150M</hasModel>
</GraphicsCard>

<RAM rdf:ID="Yousif_RAM">
    <totalMemorySize rdf:datatype="&xsd;float">2.0</totalMemorySize>
    <remainingMemory rdf:datatype="&xsd;float">1.2</remainingMemory>
    <hasRAMUnit rdf:datatype="&xsd;string">Gigabyte</hasRAMUnit>
</RAM>

<ROM rdf:ID="Yousif_ROM">
    <totalROMSize rdf:datatype="&xsd;float">160.0</totalROMSize>
    <remainingROM rdf:datatype="&xsd;float">107.0</remainingROM>
    <hasROMUnit rdf:datatype="&xsd;string">Gigabyte</hasROMUnit>
</ROM>
```

Figure 3-19. Client Profile example belonging to a laptop device with owner being Yousif

3.6. Chapter Summary

Given the importance of ontologies for modeling knowledge within context-aware systems as well as the relative ease of sharing ontologies between different entities within such systems, and the limitations found within existing ontologies, we have described in this chapter our own improved and extended ontology for modeling context information. Although this Global Skeletal Ontology is not quite complete, it solves many of the problems found with most existing models. Our general ontology is not as abstract as those top-level ontologies presented by some researchers, and is not too specific - only usable within a small set of predefined domains. Consequently, our ontology expands on upper-level ontologies provided in earlier models, and generalizes the domain-specific ontologies seen in others.

We also illustrate the use of the GSO to create domain-specific ontologies by dissecting the GSO into only a subset of the needed concepts and then extending that ontology subset in order to model the specific concepts within the domain under consideration.

We have introduced two concepts that are specific to our context-aware architecture and negotiation protocol: Client Profiles and Client Views. The former supplied context Providers with the information necessary for determining the identity and capabilities of the negotiating Client. Meanwhile the latter supplied Clients with the ontology information necessary to forming well-informed context requests during its negotiation, based on context information revealed within its Views.

In chapter 4, we will present our context-aware system architecture which utilizes our GSO, GCO, and Client Profiles and Views. Our architecture overcomes the limitations of those presented in chapter 2, by including the ability to establish Context-Level Agreements (CLA) with context Clients through context-level negotiations. This gives Providers the ability to adapt their resources based on established agreements, and to meet complex context request from Clients.

Chapter 4

Context-Aware System Architecture

4.1. Chapter Objectives

In chapter 2, we provided background information on existing system architectures and middleware in context-aware environments: the Context Managing Framework (CMF), the Service Oriented Context-Aware Middleware (SOCAM), the Context-Awareness Sub-Structure (CASS), Hydrogen, and Gaia in sections 2.2.1 through 2.2.5 inclusive.

CMF lacked a strong context model capable of representing complex relationships between various context concepts. CMF also restricted context delivery to a request/response and subscription/notification method that was inflexible to changes in the surrounding environment. SOCAM created a new problem by permitting Clients to freely query and pull the context information they are interested in without regard for privacy or security issues. Hence SOCAM was mainly designed to meet the context needs of enclosed domain-specific systems where no external intrusion attempts occur - an assumption that is unacceptable in ubiquitous environments where context consumers (Clients) are in constant motion in and out of the system as they search for their needed context information. Further problems existed in CASS, where no ontology was used at all in modeling acquired context information. An SQL database was used to store the system's acquired context, and context consumers openly accessed the database which did not differentiate between Clients and did not have any regard for data privacy. Hydrogen faced not only the same problems as the earlier systems, but also the problem of system designers' inability to share context knowledge between servers based on their current system design. Finally, Gaia's directory-based access to context information provided an obscure method for context retrieval by consumers. It was unclear how Clients would be made aware of changes to the directory structure, or whether or not their needed context information existed within the directory in the first place.

Clearly, most existing context-aware middleware and architectures have various disadvantages when it comes to a method for modeling a context knowledgebase or an approach for sharing that knowledge with entities interested in their stored context information. We have already suggested how ontologies are the optimal solution to modeling context, given their ability to represent complex relationships between context concepts, the relative easiness of sharing ontologies, and ontologies' high level of formality. Based on these conclusions, we saw the need for a global ontology that would be able to represent knowledge in context-aware environments without being too abstract (as was suggested in [17] and [18]) or so specific such that it is restricted to certain domains. Our Global Skeletal Ontology (GSO) in chapter 2 supplies system designers with a wider range of ontology classes and properties. These in turn are capable of modeling an increased number of concepts of context information. Any context-aware system architecture should therefore be designed with an ontology similar to the GSO for modeling its acquired context information.

Another problem with earlier system architectures was the lack of a common method through which context consumers could access context information *independently* from the context providers' designs and database structures. In some designs, Clients were expected to have knowledge of the context Provider's storage structure including knowledge of any changes that might happen to the model itself. Those systems that did not make the former assumption restricted context Clients to a *subscription-notification* method of context dissemination, which prevents Clients from making complex context requests based on the structure of ontologies used and the dynamic nature of the Clients' and Providers' environment. Establishing Context-Level Agreements (CLA) through context-level negotiations is our solution to this obstacle. Our context-level negotiation protocol is derived from the natural structure of OWL-based ontologies; it is therefore intuitive to apply and allows Clients to compose complex context requests according to their needs. Context-level negotiations should thus be part of any context-aware system architecture, so that Clients can freely express their context requirements and so Providers can express their context delivery abilities.

To overcome the above limitations in earlier context-aware system architectures, and to take advantage of the benefits of ontology use and the adaptability of Context-Level Agreements

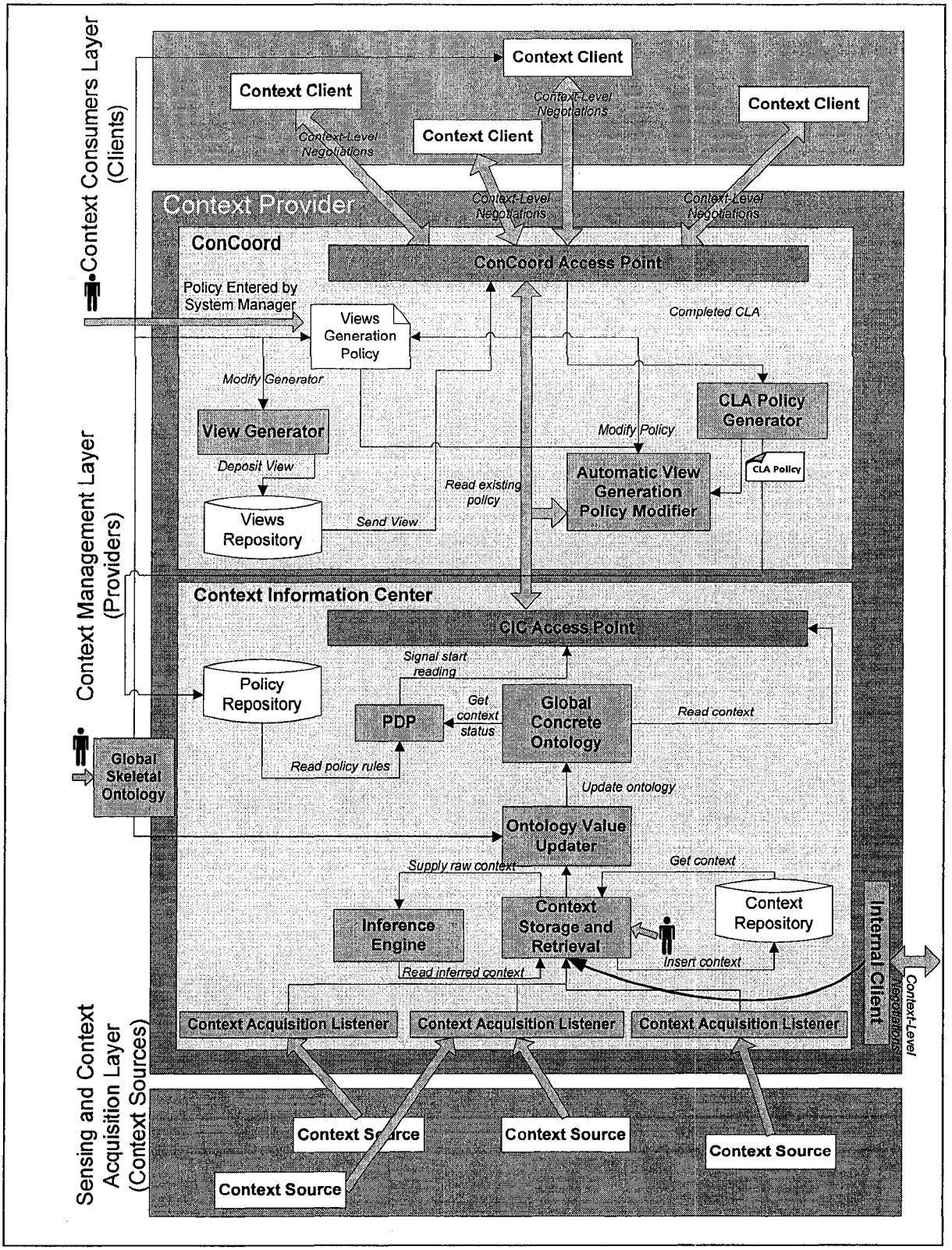


Figure 4-1. Context-aware system architecture with three levels: Clients, Providers, and sources

through negotiations, we introduced a new context-aware system architecture. Figure 4-1 provides a visual representation of our three-layered system architecture.

The general three-layer hierarchical composition used by most context-aware system architectures was adopted. These three levels include the sensing and context acquisition layer, the context management layer, and the context consumer layer, and each layer can be developed separately from the layers above or below it.

4.2. Sensing and Context Acquisition Layer

Sensors are a crucial component in context-aware environments. Their ability to sense context at different levels of abstraction provide context management systems with the context that is necessary for manipulating and inferring context information from higher levels. This context information can be modeled and stored for dissemination to interested Clients. Sensors take on different forms based on their sensing capabilities. Some sensors are restricted to sensing low-level contexts, such as motion detection and light levels, while others are more complex and exploit a wide variety of sensed context in order to compose, assemble and infer a set of higher-level context information. The preference, however, is to maintain simplicity at the sensor layer by moving complex processes related to inference, composition, and manipulation of context to the context management layer. Sensors should therefore remain small, inexpensive, and easy to distribute within the environment.

A variety of sensors exist. Motion detectors such as accelerometers and angle of inclination detectors are widely used in today's technology; these include angle of inclination detectors found on devices where the user's graphical interface is rotated in accordance with the angle at which the device is held. Audio sensors capable of measuring volume level, background noise and base frequency are also in use, along with optical sensors such as photo-diodes, IRs, and color sensors that measure light intensity, wavelengths and densities.

The vast majority of context-aware systems rely on location as the sole type of context sensed within the environment. Location-awareness allows context-aware systems to deduce new context, including current activities, actions, or services needed. Location detection could be achieved through GPS systems, or through the signal strength of mobile devices carried by users.

Other specialized sensors also exist for measuring temperatures, air pressure, humidity, and gas concentrations, as well as to detect touch, pulse, blood pressure, emotion (through stress levels, and facial expressions), skin resistance, and so on.

The sensing and context acquisition layer in our architecture is not restricted to low-level sensors such as those mentioned above. Our definition of a context source includes any entity capable of providing any type of context information, be it fine-grained or course-grained. High-level context sources can employ one or more low-level context to produce more useful high-level context through inference, composition and manipulation.

A third source of context can arrive from other context providers within the environment. This is achieved through an *Internal Client* component, present within each context Provider that is capable of engaging into context-level negotiations with other context Providers to establish Context-Level Agreements. These agreements can list the low-level and high-level context information the Provider is interested in acquiring.

4.3. Context Consumer Layer

Entities interested in obtaining context information are referred to as Clients. Clients can be mobile devices, applications, or services. Even with the increased processing powers available today, most mobile devices are incapable of sensing the entire context they need in order to meet the user's requirements due to limitations in size, lack of available resources, and short battery life. Our suggestion for overcoming these limitations is the development of Clients that are capable of negotiating their context information needs with available context providers. Negotiations are performed using our proposed Context-Level Negotiation Protocol, which will be presented in further detail later on in chapter 5.

Context information that Clients are permitted to access from a context Provider can be determined in a number of ways. Access to some context information may be granted to Clients holding passwords, or private or public keys, or through biometric human recognition methods such as fingerprinting, retinal scans, voice recognition or even DNA sampling. These methods of recognizing Clients are well established for enclosed environments where all Clients' context needs and interests are based on identity. Security is of utmost importance in context-aware systems; moreover, limiting access protects the privacy of all Clients in the environment, by preventing the leak of any private context information

hardcoded in the system, collected from sensors, or inferred from various context information.

Nonetheless, access to context information is not confined to the identity or security clearance a Client may possess. In chapter 3 we have seen how our proposed use of ontology-based *Client Profiles* can be used to determine what 'type' of context information Clients may be interested in receiving from the Providers. Profiles supply context Providers with information on the type of Client performing the negotiations – that is, on whether the Client is an application, a device, or a human user. Profiles are also able to express the identity and role played by the Client within the system. For instance, Profiles could depict the role of a human Client within a company, such as CEO, engineer, secretary, etc. Most importantly, Profiles also supply context Providers with a clear description of the Client's capabilities and capacities. Consequently, the most appropriate Views are handed over to the Client.

In general, any entity interested in acquiring context information from context Providers who is also able to perform context-level negotiations is considered a context Client. Thus even context Providers can play the role of a Client with other Providers in order to receive context information which they cannot acquire directly through sensors or infer from various contexts at hand. The precise steps involved in negotiations are presented in detail in chapter-5.

4.4. Context Management Layer

The constraints present within the majority of context-aware architectures (including the ones presented earlier), required a new architecture that would not allow Clients open access to private context information on the Provider's side, as in [20] or [23]. Other architectures lacked a well-structured model for their context knowledge, such as [21], while others limited context dissemination to subscription/notification methods based on first-order logic. Client's context requirements depend on many dynamic factors which are in turn influenced by their environment.

These limitations justified the introduction of a new architecture for context Providers. Providers exist in an intermediate layer between the context sources' layer and context Clients' layer. The Provider's purpose is to acquire context information arriving from

various context sources, model, infer and store that context information, negotiate the context needs of incoming context Clients, and to deliver that context information to the Clients based on their established Context-Level Agreements (CLA).

Providers consist of two interconnected units, each representing a unique set of functions: the Context Information Center, and the Context Coordinator. The Context Information Center, or CIC, is responsible for acquiring context information from available context sources, for inferring new context information from the acquired context, and for storing its context knowledge within its context database. The CIC is also responsible for monitoring changes that occur within its context knowledge so as to meet the requirements of Context-Level Agreements that have been established with external Clients.

The Context Coordinator, or ConCoord, handles context-level negotiations with incoming Clients. The ConCoord also retains a repository for Client Views that determines what context information Clients can access (based on the Clients' Profiles submitted by the Clients themselves during negotiations). All established CLAs are utilized by the context Provider to modify existing Views according to the new requirements formed as a result of these CLAs.

In the following sections, we present a description and purpose for each component within the context Provider's architecture.

4.4.1. Context Information Center

As stated above, the Context Information Center (CIC) is responsible for acquiring context information from nearby sources to model the acquired context, to use to infer new context information, to store and monitor the state of the stored context in order to determine whether any CLA conditions established with Clients are triggered as a result.

4.4.1.1. Acquiring Context Information

Sensed context values arriving to context Providers from various sources are gathered through various sensors. These sensors can differ in ability, type, model, sensing frequency, make and model. In addition, these sensed values are expressed through different protocols and message formats. Since converting these direct sensor values into the general context

model used by the context provider is a resource-intensive task, little-to-no manipulation of the sensed data is applied at the sensor level to maintain the sensor's design simplicity.

Context Providers must therefore possess components that are capable of translating the sensor readings into understandable context. The 'Context Acquisition Listener' component is designed to decode incoming sensor readings into a format recognizable by the context Provider – for instance by decoding values arriving from a temperature sensor into their respective Celsius degree values. With the appearance of new types of sensors that are unrecognizable by Providers, Context Acquisition Listeners must be manually updated by system managers so that context Providers can recognize the sensors' messages. Sensor readings that have been decoded at the *Context Acquisition Listener* are eventually delivered to the *Context Storage and Retrieval* unit.

4.4.1.2. Internal Client

Context information obtained by context Providers is not limited to direct sensor readings. Providers should be capable of negotiating their context needs with other nearby context Providers in order to enlarge their context knowledge base, to increase the accuracy of context information it receives from its own set of sensors in the Sensing Layer, and to serve incoming Clients with a wider set of context information. Therefore, for a Provider to play the role of Clients, the 'Internal Client' component is needed. The *Internal Client* is equivalent to the Clients within the Client layer discussed earlier in section 4.3. Internal Clients proceed in context-level negotiations with other context Providers. These negotiations result in Context-Level Agreements that list the context information these Providers will supply to the negotiating Client - who is also a context Provider in this case.

In a method similar to that used by Context Acquisition Listeners, the Internal Client forwards its received context information to the 'Context Storage and Retrieval' unit.

4.4.1.3. Context Storage and Retrieval Unit

Context information arriving to the *Context Storage and Retrieval* unit (CSR) is used by the Provider to update its knowledge base and to serve Clients with which it has established Context-Level Agreements (CLAs). Any context that arrives at the CSR must pass through three stages. At the first stage context is saved within a *Context Repository* if it arrives from sensors through the Context Acquisition Listeners, or from other context Providers based on

CLAs established by the Internal Clients. The repository is a large storage area where all incoming context information is stored. Context-aware systems require access to historical context information. The historical context is used to infer new context information through composition or pattern recognition and prediction.

The same context information that was sent to the context repository in the first stage is also delivered to an *Inference Engine* in the second stage. Inference engines utilize the newly received context information, as well as context that had been stored within the repository to infer new context. Inference engines contain a set of rules that depend on current context contents. The set of rules whose contents are satisfied are selected for execution. Context information produced by the execution of some rules may result in the execution of other rules within the engine.

System designers can build their own inference engines, which would be capable of meeting their user defined need. One such inference engine is the Ruby-based Simple Inference Engine (SIE) [26]. SIE is based on the ESIE system with an XML knowledgebase that have elements for goals, rules, and questions. Rules within SIE consist of a list of attribute/value pairs that must evaluate to "true" before passing. Rules that pass trigger a list of actions composed of attribute/value pairs.

Jena [27], a Java framework for building semantic web applications, includes a rule-based inference engine. Jena was designed so that the composition of several inference engines and reasoners could be plugged into the framework, deriving RDF assertions from the base RDF document. Jena contains several reasoners, including a transitive reasoner that supports storage and traversal of class and property lattices, an RDFS rule reasoner, an OWL reasoner that provides an implementation of the OWL/Lite subset, a DAML micro reasoner, and a generic rule reasoner that supports user-defined rules, forwarding chaining, tabled backward chaining and hybrid execution strategies.

Context information resulting from the inference engine is also stored into the Context Repository. In the third stage, context information stored within the repository is continuously extracted and delivered to the next component, the Ontology Value Updater.

4.4.1.4. Updating the Ontology and CLA Enforcement

We have already explained in section 3.2 that all context providers have their own Concrete Ontologies. These ontologies are domain-specific derivations from the general, publicly available, and globally sharable Global Skeletal Ontology. We also showed that Concrete ontologies contained all the updated values of the latest context received by the context Provider. New context information that was stored within the context repository, whether arriving from sensors or inferred by the inference engine should be used to update the Concrete Ontology. Thus, the Context Storage and Retrieval unit has the responsibility of continuously retrieving new context information from the context repository and sending it to the Ontology Value Updater.

The Ontology Value Updater must have knowledge about the structure of the Global Concrete Ontology, and thus be able to easily update values stored within the GSO. These updates are necessary for triggering the delivery of context information according to conditions present within established Context-Level Agreements. All CLAs are converted into policies that enforce the agreements that Providers had made with Clients.

All policies derived from Context-Level Agreements are stored within a *Policy Repository* whose conditions are continuously read by a *Policy Decision Point (PDP)* to trigger delivery of context information to affected Clients. The PDP reads the GCO to see if any changes that have taken place to the stored values will trigger any of the conditions present within the policies. Triggered conditions can affect the activation/deactivation of CLAs, or activation/deactivation of specific context information delivery.

When changes in context values turn some CLA policy rules to true and a CLA condition is triggered, the Policy Decision Point signals the Context Information Center Access Point (CIC Access Point) to retrieve the required context information from the Global Concrete Ontology. The retrieved context information is context that must be delivered to the affected Clients. As was indicated earlier and shown in figure 4-1, the CIC does not communicate directly with the Clients; this remains ConCoord's responsibility. Context information retrieved by the CIC Access Point from the Global Concrete Ontology is instead delivered to ConCoord for delivery to the respective Clients.

The Global Skeletal Ontology and Domain-specific ontologies (Global Concrete Ontologies) undergo constant updates. Changes to the former affect all Clients and Providers present within the context-aware system, while the latter changes in accordance with the context acquisition abilities or domains of Providers.

4.4.2. Context Coordinator

We have seen how the CIC of the context Provider is responsible for communication with context sources and other context Providers in order to receive the context information it needs so it can update and expand its knowledgebase, and meet the requirements of CLAs that have been established with negotiating Clients. Still, the CIC is not concerned with direct communication with Clients. Rather, the Context Coordinator (ConCoord) is the component responsible for performing context-level negotiations with Clients.

In section 3.4 it was clearly noted that Clients who engage in context-level negotiations with context Providers require a clear description of the context information they are permitted to access. This information included ontology classes, datatype properties, relationships and individuals the Client had the right to access from the Provider's Global Concrete Ontology. We referred to such descriptions as *Client Views*, or CV.

Client Views are formed by a set of user-defined policies entered by the system managers prior to the system's runtime. A *View Generator* component within ConCoord utilizes these policies to create an initial set of *Client Views*. These Views are stored within a *Views Repository* so they can be retrieved whenever they are needed for delivery to the Clients.

The ConCoord Access Point is the component responsible for engaging in context-level negotiations with incoming Clients. Negotiations are performed with multiple Clients in parallel; hence, context Providers must be able to handle concurrent communication with multiple Clients. Every successful negotiation performed between a Client and the context Provider results in a Context-Level Agreement (CLA) that is forwarded by the ConCoord Access Point to the *CLA Policy Generator component.*

The *CLA Policy Generator* has the task of translating established CLAs into policies. These generated policies are then delivered to two different components: the *Policy Repository* and the *Automatic View Generation Policy Modifier.* As explained in the previous section, policies delivered to the Policy Repository are used to monitor changes in context that is

stored within the Global Concrete Ontology. By triggering rules in these policies, the context information needed by Clients is delivered according to conditions set within their CLAs.

Policies delivered to the Automatic View Generation Policy Modifier (AVGPM) serve a different purpose. The initial policies used to create the Clients' Views did not take into consideration potential changes situation surrounding the context-aware system. Context-Level Agreements established with Clients may result in changes to the context information other Clients are permitted to access. The Provider's ability to serve certain types of context information may also be affected by the increasing number of Clients that have requested a given type of context. The quality with which those contexts were guaranteed to be served may also change. As the Provider's ability to serve certain types of context increases or decreases, changes to the original Views generation policies are needed. Since manual updates to policies within context-aware systems are inefficient, there is a need for automatic updates to View generating policies.

The AVGPM modifies the stored View generation policies to meet the new requirements and capabilities of the context Provider. As new Clients arrive for context negotiation with the context Provider, they are served with the updated Views, and any CLAs they establish with the Provider may also be used to update the recently modified policies. Clients that have already established CLAs with the provider may have changes occur to Views they received during their negotiation phase. Context requests affected by these changes are automatically cancelled by the Provider, due to conflicts with their respective Views. Whether a Client is notified of changes to its Views dependent on the Context-Level Agreement previously established with the Provider. In general, Client who had requested notification of changes to their Views are supplied with the updated Views, depending on the negotiated conditions. Further details of View updates are discussed in Chapter 5.

4.5. Chapter Summary

The context-aware system architecture presented within this chapter described an alternative solution to current architectures. The context modeling problems seen with some of the available middleware were solved by the use of ontologies. Almost all systems presented have adopted a method of subscription-notification supplying context information to consumers. Many problems, including those related to privacy and security, as well as the

inability to personalize delivered context information, have been solved by the use of context-level negotiations in our proposed architecture.

The agreements reached through these negotiations are internally converted into policies that can meet the Clients' requirements and adapt and dedicate resources in the context Provider to supply Clients with the context information required in accordance with their own personalized conditions.

In chapter 5, we will present the details of our context-level negotiation protocol. All the steps necessary for performing the negotiations, from the initial exchange of Profiles and Views, to the final stage of enforcing these agreements, will be clearly presented.

Chapter 5

Context-Level Negotiation Protocol and Context-Level Agreements

5.1. Chapter Objectives

The limitations previously seen in other context-aware architectures stemmed from two problems these architectures shared. First, the architectures used an inferior context modeling approach that was unable to model all complex context information, properties, and relationships. We solved this problem by suggesting the use of OWL-based ontologies separated into four categories: Global Skeletal Ontology, Global Concrete Ontology, Client Profiles and Client Views.

The second problem was related to the method and extent of context information access permitted to Clients from the Providers' context repositories. To solve this problem we suggest the establishment of Context-Level Agreements between context Clients and context Providers. This creates a list of personalized context information for delivery by Providers to the requesting Clients, in keeping with the conditions and quality of context levels requested. These agreements would also allow Providers to reserve services and resources such that context requests and conditions in existing agreements (CLAs) are not negatively affected by future agreements with other Clients.

Context Views that belong to Clients (Client Views) and are stored within the Views repository are usually also affected by agreements made with Clients. Therefore CLAs established with Clients through context-level negotiations are used so that Client Views in the View Repository can adapt to changes resulting from newly established CLAs.

Establishing Context-Level Agreements secures the context privacy of other Clients by not permitting other Clients access to personal context information. CLAs also prevent Clients from freely accessing the context Providers' knowledge repositories. Finally, knowledge of the syntax and semantics employed by Providers in their internal context representation

becomes irrelevant to Clients, as the latter use ontologies to express their context information requests.

To establish CLAs, Clients and Providers need to negotiate both their context interests and their context delivery abilities respectively. These negotiations must be performed through a dedicated context-level negotiation protocol expressing the nature of context-aware systems. This protocol must be fast, well-ordered and easily implemented by Clients and Providers. Any protocol used must allow an exchange of Client Profiles (section 3.5) that expresses the identities and capabilities of the negotiating Clients.

Since Clients delivering their Profiles have done so with the expectation of receiving their respective Views in return, it is important that the negotiation protocol permit the transfer of Client Views by Providers.

Most importantly, the protocol used must allow Clients to prepare a list of conditions under which the finalized CLA is activated and de-activated. Clients are usually interested in a variety of context information and not just a single context; thus, the protocol should also give Clients the ability to make multiple context requests, each having its own list of activation and de-activation conditions separate from other requests made.

Since context acquired by context Providers is not equally or accurately measured, and the sensing frequency differs from one sensor to another. It is that Clients be allowed to express the Quality of Context levels with which they wish to receive their context information. This permission should be part of the negotiation protocol.

Lastly, Views composed by a Provider for a negotiating Client undergo numerous updates and changes during the Provider's lifetime. Changes in the Views directly affect established Context-Level Agreements with some of the existing Clients. It would therefore be in the interest of the Client to be able to express the action that must be taken by the Provider when changes occur within their Views. Thus, the context-level negotiation protocol should also allow Clients to express these actions freely.

All successful context-level negotiations between context Providers and context Clients result in a Context-Level Agreement (CLA) that details the Clients' context information needs and the Providers' expected level of context delivery in terms of all the conditions expressed within the CLA and all the expected quality levels. Clients are expected to

establish Context-Level Agreements with all context Providers from whom they expect to receive context information.

In the remainder of this chapter, we will present our context-level negotiation protocol, the structure of the protocol's messages, and a number of examples to clarify these concepts.

5.2. Context-Level Negotiation Protocol

Context-level negotiations between Clients and Providers are divided into eight steps, as provided in figure 5-1. In the first step, Clients supply Providers with their Profiles. In exchange, the second step is for Providers to supply Clients with their respective Views. Once Providers have been familiarized with the Clients, and Clients have gained knowledge about the context classes, properties, relationships, and individuals they are permitted to access, negotiations pertaining to the CLA activation and to the Clients' specific context information requests.

In the third step Clients and Providers negotiate the conditions related to the *Activity* of the CLA as a whole. These conditions regulate when a CLA is activated and when it is deactivated. Once the CLA conditions have been agreed on, Clients make their context requests, which detail the context information they are interested in receiving from the Provider. Each request has context conditions and quality levels that must be met prior to delivery to the Client; both are negotiated in steps four and five, respectively. Since Clients are usually interested in receiving more than one type of context information, the steps concerned with negotiating the needed context information, the conditions under which this context is delivered to the Client, and the quality levels that context must meet prior to delivery are all repeated for each context request made by the Client.

Once all context requests have been included in the CLA, Clients perform the sixth step in negotiating the method used to update the Client's context Views in the event changes do occur. Once a View update method has been agreed on, Clients and Providers finalize the contract by exchanging the CLA and expressing their satisfaction with it.

In figure 5-1 a summary of the eight steps of negotiations is shown using a transition diagram. The diagram illustrates the high-level cycle of messages exchanged during each step, where Clients and Providers exchange offers and counter-offers until both parties reach

an agreement on acceptable levels of context to be supplied. Line 20 from the transition diagram shows to the repetition of negotiation steps 3, 4 and 5 for each context request the Client makes.

During negotiations, fifteen types of messages can be exchanged between Clients and Providers. Some of these messages are used during the negotiation phase, while others are used after the CLA has been established. Table 1 provides a list of all 15 types of exchangeable messages.

	Message Type	**4-Bit Header**
1	Start Negotiation (Client → Provider)	0000
2	Request Profile (Provider → Client)	0001
3	Send Profile (Client → Provider)	0010
4	Send View (Provider → Client)	0011
5	CLA Validity Condition	0100
6	Context Information Request	0101
7	Context Request Applicability	0110
8	Quality of Context Request	0111
9	Client View Update	1000
10	End Negotiations	1001
11	Send Context	1010
12	Modify Established CLA	1011
13	Delete CLA	1100
14	Modify Context Request	1101
15	Delete Context Request	1110

Table 1. The 4-bit header Message Type values for negotiation messages

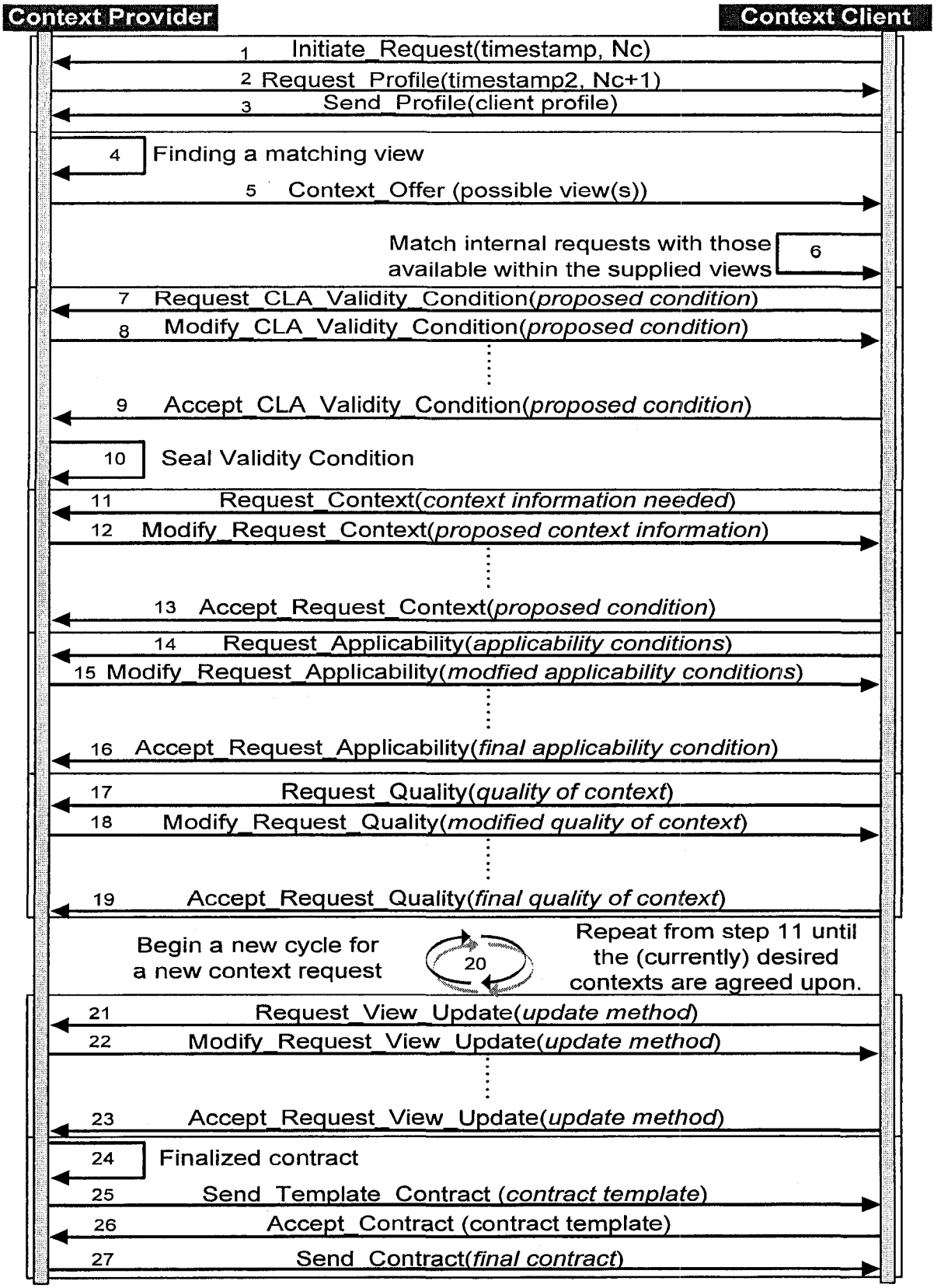


Figure 5-1. Context-Level Negotiation Protocol Transition Diagram

5.2.1. Exchange of Clients' Profiles and Views

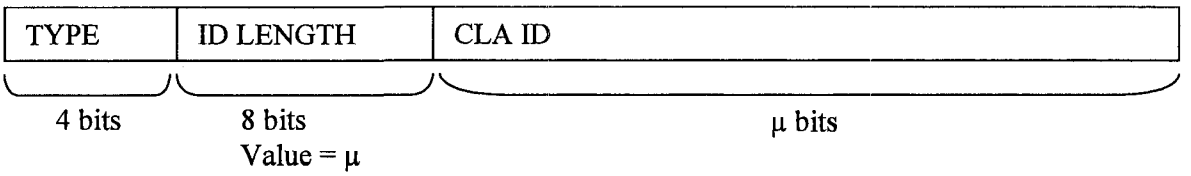
The first two steps in our context-level negotiation protocol are concerned with identifying the Client to the context Provider and familiarizing the Client with context information it has the right to access. We have seen some of the details of these two steps earlier, when we discussed the structure of Client context Profiles and Client context Views.

Context Clients begin the negotiation session by sending a '*Start_Negotiation (ID)*' message to the context Provider. This initial message does two things. First, it makes the context Provider aware that a new Client within the system is interested in acquiring some context information. Secondly, it allows the Client to supply the Provider with an 'ID' variable. This variable is a unique identifier provided by the Client in order to identify the Context-Level Agreement (CLA) is to be established during the current negotiation session. The ID also has the second purpose of indicating the CLA to which the context information belongs. This ID will later be attached to context messages sent by the Provider to the Client.

IDs and other strings within the protocol can vary in length. Our protocol therefore precedes all strings with an 8-bit field, representing the length of the string that follows. The use of eight bits allows protocols to use strings with lengths anywhere between 0 and 255 bits. However, since no strings of length 0 are used, 00000000 is used to represent strings of a length equivalent to 256 bits. Thus, a CLA ID provided by a Client can be any string with a length in the range of 1 to 256 bits. Since ASCII hexadecimal characters require 8 bits per character, then an ID could be any string with 1 to 32 characters.

For instance, if a Client sends a *Start_Negotiation* request message to a context Provider with an ID composed of 6 ASCII characters, such as "A53B46", then the 8 bits preceding the ID characters will be evaluated to 48 because (6 characters) X (8 bits/character) =48 bits. Representing 6 ASCII characters therefore requires 48 bits of space.

The general structure of a *Start_Negotiation(ID)* message follows the format shown below:



The TYPE is a 4-bit field used to indicate the type of message being sent. Since the message is of type Start_Negotiation, thus using the header value provided in line 1 in table 1 allows us to determine that the header is equivalent to '0000'.

The ID LENGTH is an 8-bits field used to represents the number of bits to be taken by the ID of the CLA being negotiated. We have already explained the process used to find the ID's length.

Finally, the CLA ID field contains the bit representation of the ASCI characters for the CLA ID. Some of the common conventions used for naming variables in programming languages are utilized in choosing an ID value. ID names are case-sensitive with a length of 1 to 256 characters. The characters can be a combination of letters and digits excluding characters such as +, -, =, &, (,), *, ^, %, $, #, @, !, ~, |, \, and a number of others. Thus, characters can be any alphabetical letter, digits, or the '_' symbol (as long as these characters are not located at the beginning or end of the ID).

When the context Provider represented by ConCoord (the entity capable of performing negotiations with Clients) receives the *Start_Negotiation* message, a new negotiation session for a new CLA begins between the Client and Provider. Since the context Provider is able to perform context negotiations with the Client, an acceptance of the negotiation is delivered to the Client through a *Request_Profile* message. Context Providers require access to the Client's OWL-based profile documents, which contain information on the Client's identity, capabilities and any other information needed to determine the View that must be supplied to the Client. *Request_Profile* messages are simple messages that have the exact same structure as that belonging to the *Start_Negotiation* message. The 4-bit header field defines the type of message being sent by the Provider: in this case, '0001' represents a Request_Profile message. The next two fields are the 8-bit CLA ID length, and the CLA ID fields respectively. The CLA ID used in this situation is the same ID provided by the client in its Start_Negotiation message. The CLA ID is included in case the Client is engaged in multiple context-level negotiations with other context Providers in the system. Utilizing the CLA ID thus gives Clients the ability to determine which Profile request is arriving for which negotiation session.

Clients receiving Profile-Request messages must supply their Profiles to the Providers in order for negotiations to commence. Delivery of the Profile commences with a Send_Profile message. Since Profiles can consist of large OWL-based documents, there is no fixed length for Profiles that Clients must follow. Thus, each time a Send_Profile message is sent to the context Provider, the message carries with it as many or as little sections of the Client's Profile as needed until the complete Profile document has been delivered to the Provider. Since our context-level negotiation protocol is designed to utilize other existing protocols for delivery, the message's maximum possible length depends on the underlying protocols used.

Our context-level negotiation protocol is located at the application layer. The choice for the protocols used over the data link, network, and transport layers are determined by the system designers. Therefore the maximum length of permitted messages used by our context-level negotiation protocol is directly related to the protocols used at the lower layers. If a choice is made (as in our implementation) to use IP and TCP at the network and transport layers respectively, the maximum length of the TCP's data field is variable. Two choices are possible in this case. The first is to strictly place a pre-defined size at the application layer, and hence within our negotiation protocol. The second solution is for Clients and Providers to allow the message size to expand or shrink dynamically according to the underlying network and transport layer protocols.

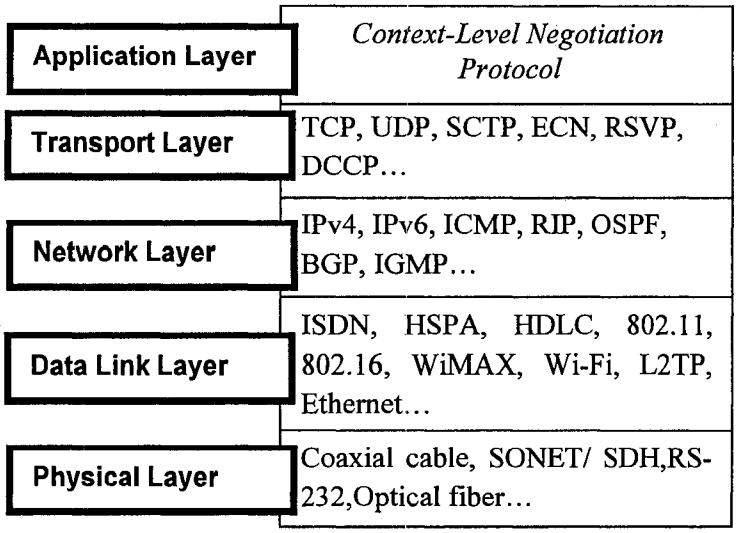


Figure 5-2. Protocol hierarchical levels

The Data field in TCP is variable; therefore, if TCP was the protocol choice for the Transport layer then a decision must be made as to how many bytes each message should be allowed to carry. The sliding window mechanism employed by TCP is one of the main determinants of how much data a segment could send based on an examination of the Window field on each side of the communication. Devices using TCP also dictate a maximum segment size (MSS)

value for TCP size that cannot be exceeded, no matter how large the window size is. Thus, message lengths are chosen based on the current window size, ensuring that the MSS of each device is not exceeded.

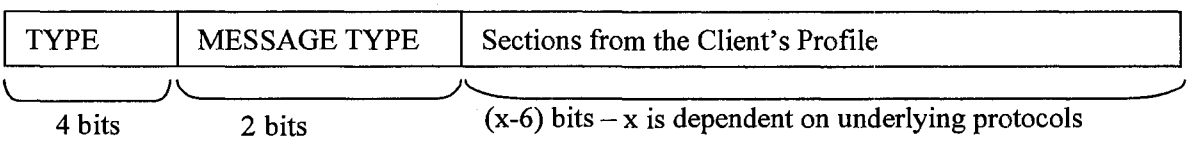
In general, all networks are required to handle an IP datagram with a size of 576 bytes. The standard MSS for TCP is calculated by subtracting 20 bytes for the TCP header and 20 bytes for IP header, leaving 536 byte for the MSS. Using the options available for both IP and TCP headers would cause defragmentation once the 576 bytes are exceeded.

Given the wide range of data field sizes in the various protocols available, we have left it to the Clients and Providers to add as little or as much content to their exchanged messages as the underlying protocols permit. Consequently, delivery of a Client's Profile commences with a Send_Profile message, where each message carries with it as many sections of the Client's Profile as possible until the complete Profile has been delivered.

The Profile delivery proceeds in three stages. In the first stage, called the *Start* stage, the first Send_Profile message is delivered from the Client to the Provider. The message informs the context Provider that the Client's Profile delivery has started and that Profile delivery of the remaining sections is underway. The Start message also carries with it as many Profile sections that have been converted to their binary formats as possible.

Remaining sections of the Profile are delivered in the second stage, called the *Body* stage. Unlike the *Start* stage, the body stage is composed of repeated Send_Profile messages, each carrying sections of the Client Profile until the last part of the profile remains undelivered. Using TCP guarantees that messages delivered are ordered once they arrive at their destinations. *Send_Profile Body* messages are delivered sequentially until the second-to-last section of the Profile is delivered.

In the last stage, the End Send_Profile message is delivered to the Provider carrying with it the last remaining section of the Client's Profile. The message indicates to the Provider that Profile delivery has reached its end and that the complete Profile is now available to it.



Start and End Send_Profile messages share the structural format shown below. The 4-bit TYPE field has a purpose similar to that of the Start_Negotiation and Send_Profile messages. Using table 1, the value for the TYPE field of a Send_Profile message is equivalent to '0010'.

Since sending Clients' Profiles proceeds in three stages - *Start*, *Body*, and *End* - a 2-bit MESSAGE TYPE field must indicate which of these three delivery choices the current message belongs to. '00' indicates the Start stage, '01' indicates the Body, and '10', for the End message. These are shown in Table 2. There are no restrictions at the context-level negotiation protocol levels regarding the number of bits from the Profile each message is permitted to send; Clients can therefore deliver as many bits as possible as long as the number does not surpass the limitations imposed by the underlying protocols. Consequently, if the transport layer protocol allows 'x' amounts of bits to be delivered, Clients can therefore deliver up to 'x-6' bits from their Profiles for each message. Six bits were subtracted for the 4-bit header and 2-bit message type fields respectively.

MESSAGE TYPE	*Value*
Start	00
Body	01
End	10

Table 2. Send_Profile message types

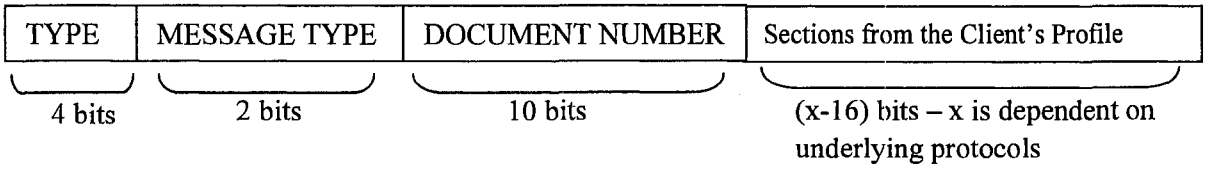
Sending the Profile's Body messages utilizes the same type of message structure shown above with start and end messages. In a manner similar to that of the Start and End messages, the Body message includes a 4-bit Type header with a value of '0010', indicating that the message sent carries a section of the Client's Profile. The two bits '01' are placed within the Message Type field marking the message as one of many possible Profile Body messages.

Once the Client's Profile has been received by the Provider, the context Provider initiates a search for a matching View. As seen earlier in section 3.4, Views provide Clients with a description of the context classes, properties, relationships and individuals they have the right to access from the context Provider's Global Concrete Ontology (GCO). Access to certain Views can be restricted to a single Client or to a group of Clients. Views can be supplied to Clients on the basis of their identities, roles, abilities, and so on. The question of which Views belong to which Client depends on the design of the context Provider. A set of

policies provided by the system manager governs the decision context Providers make as to which Views are to be provided to the Clients.

Unlike Profiles that are limited to a single OWL-based document, Views can span over multiple documents. A possible Client context Views was shown in section 3.4. Views are composed of at least the *Root View* document. Any additional context classes, properties, or individuals an individual within the Root View may have, and that are not shared by other individuals of the same class type, are listed in a separate document. If necessary, this is done for each individual within the Root View. Regardless of the contents or syntax used, the difference between Client Profiles and Client Views lies in the former being composed of a single document (instantiation of an ontology), while the later is composed of multiple documents.

Given these differences, context Providers delivering Views to Clients utilize the message structure displayed below. In addition to the 2-bit Message Type field present in the Send_Profile Body messages, Send_View messages have a 10-bit field that allows the Provider to indicate the number of the document currently being delivered. Document numbers begin at '0000000000' for the Root View document, with an increment of one for each subsequent document.



The 4-bit Type header is retrieved from table 1, where the *Send_View* of line 4 indicates that the Type header should be '0011'. Just as the *Send_Profile* protocol Body messages' structure had three possible values for the Message Type 2-bit field, Send_View also utilizes the values in table 2 to represent the *Start*, *Body*, and *End* of View delivery. Unlike *Send_Profile* messages, no differentiation is present in the message formats for *Start*, *Body*, and *End Send_View* messages.

Since a Client's View may be composed of more than a single document. Therefore a 10-bit Document Number field is included. This allows Providers to inform Clients of which document the current View message belongs to. Each document may also require more than a single message for delivery to be completed.

Finalizing the exchange of Profiles and Views means that the Provider has already clearly identified the Client with whom it is currently engaged in negotiations. It also means that the Client is now able to construct all its context requests by using information contained within the View it has received to form its Context-Level Agreement. CLA establishment begins with negotiating the conditions under which the CLA should be activated and de-activated. This process is described in the next section.

5.2.2. CLA Validity Conditions

The step after Profile and View exchange mark the start of a new phase in CLA establishment. The former Profile and View exchanges clearly distinguished which entity (Client or Provider) was responsible for the delivery of messages: Clients delivered Profiles and Providers delivered Views. It was the sender's responsibility to guarantee complete delivery of the Profile/View, while the receiver listened for all incoming messages until the 'End' message was received. The negotiation steps following Profile and View exchanges are not as restrictive, as both Clients and Providers may become involved at some point in delivering the same type of context-level negotiation messages. Permitting negotiations mean allowing entities to exchange offers and counter-offers until an agreement is reached. Consequently, each type of message exchanged between Clients and Providers after Profile and View delivery can be expressed using the state diagrams shown in figure 5-3.

The state diagrams apply to all types of negotiation messages exchanged after Profile and View deliveries. Clients begin by sending their 'offer' to context Providers, and then enter a 'wait' state until the Provider's reply is received. Any received offer transfers the Provider into the 'evaluation' state. The Client's received offer is evaluated based on information within the Client's Views, the current and future abilities of the provider, and the effects an acceptance may have on established CLAs with other Clients. If the offer is accepted, the Provider sends an 'accept' message to the Client, thus transferring both back to their initial state for a new round of offer exchanges.

A Provider rejecting an offer must send a 'Reject' message to the Client. Upon receiving the rejection, the Client enters another waiting state until the Provider sends its counter-offer. The context Provider creates a counter-offer that does not compromise its performance or existing CLAs. The Client evaluates the Provider's offer and makes a similar decision to

accept or reject the offer. An acceptance concludes the negotiation cycle and both Client and Provider move on to the next cycle of context negotiations. Alternatively, a rejection by the Client requires that it makes its own counter-offer to the Provider. The cycle of offers and counter-offers continues for every step in the context-level negotiations shown in figure 5-3, until all conditions and context requests have been agreed on.

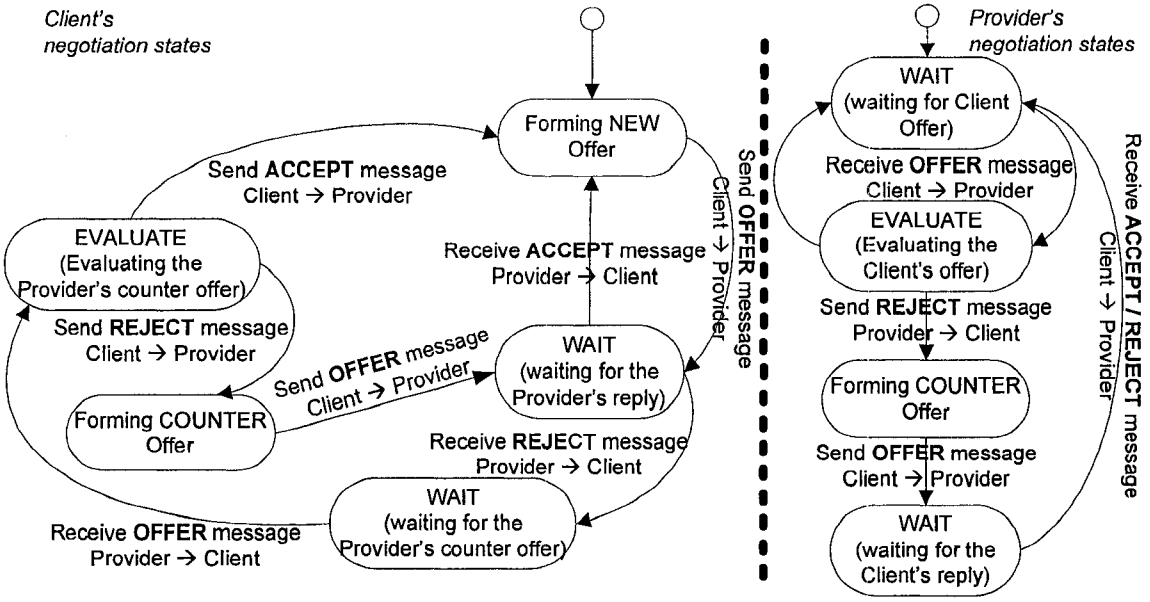


Figure 5-3. Negotiation state diagrams

Clients negotiating with context Providers to establish Context-Level Agreements are often not interested in the immediate activation of their CLAs. CLA activation is dependent on events and conditions that Clients may require to occur prior to activating or deactivating a CLA. These conditions can depend on time or on changes in context information acquired by the context Provider. Thus, Clients should have the ability to describe complex CLA activation conditions.

To cover all expected activation situations, we have broken down CLA validity conditions into three categories:

1. Periodic
2. Logical Conditional
3. Hybrid

Each of these categories is to be further explained in the following sub-sections.

5.2.2.1. Periodic CLA Validity Conditions

Clients whose CLA activation is time-dependent must establish a Periodic CLA validity condition. Periodic validity conditions must include a description of the CLA's activation time (Start time) and de-activation time (End time). Periodic CLAs require clear definitions of both start and end times, using the Time ontology publicly available in the Global Skeletal Ontology.

These Periodic CLA validity conditions can be sub-divided into two distinct groups: *continuous* and *non-continuous* periodic. The former refers to a CLA that is continuously activated and de-activated on a daily basis, while the later describes a CLA that enters the active state and is terminated indefinitely at a defined date and time.

To achieve a non-continuous validity condition Clients must only indicate the 'time' at which the CLA is to be activated and de-activated. There should not be any reference to the dates of activation and deactivation. For example, a CLA can be activated from 6:30 a.m. to 8:45 p.m. Consequently, the CLA is continuously activated and deactivated on a daily basis in accordance with the negotiated start and end times. Start and end dates without reference to specific years are also accepted as continuous periodic CLA validity conditions. In such cases, a reference to the day numbers only activates the CLA every month from the start day until the end day. A reference to both start and end days and months activates the CLA on an annual basis. Additionally, a CLA with continuous periodic CLA validity conditions remains valid until the Client or Provider requests its termination.

On the other hand, non-continuous periodic conditions must clearly define the *time* AND *date* at which the agreement must start and end. Reaching the end date and time terminates the CLA completely. For example, a CLA can be activated from 9:30 a.m. on February 5th, 2008 to 5:00 p.m. on February 27th, 2009. Thus, to create a new CLA after the current CLA's termination, new negotiations must be executed between the Client and Provider. Otherwise, modifications to the CLA validity conditions must be negotiated prior to the CLA's termination.

An example of the previous non-continuous Periodic CLA validity condition is illustrated in figure 5-4. The message shown is a high-level representation of an offer message sent by a

Client to a context Provider and serves for demonstration purposes only. Messages need to be expressed in their formal bit-level formats prior to delivery to the receiver.

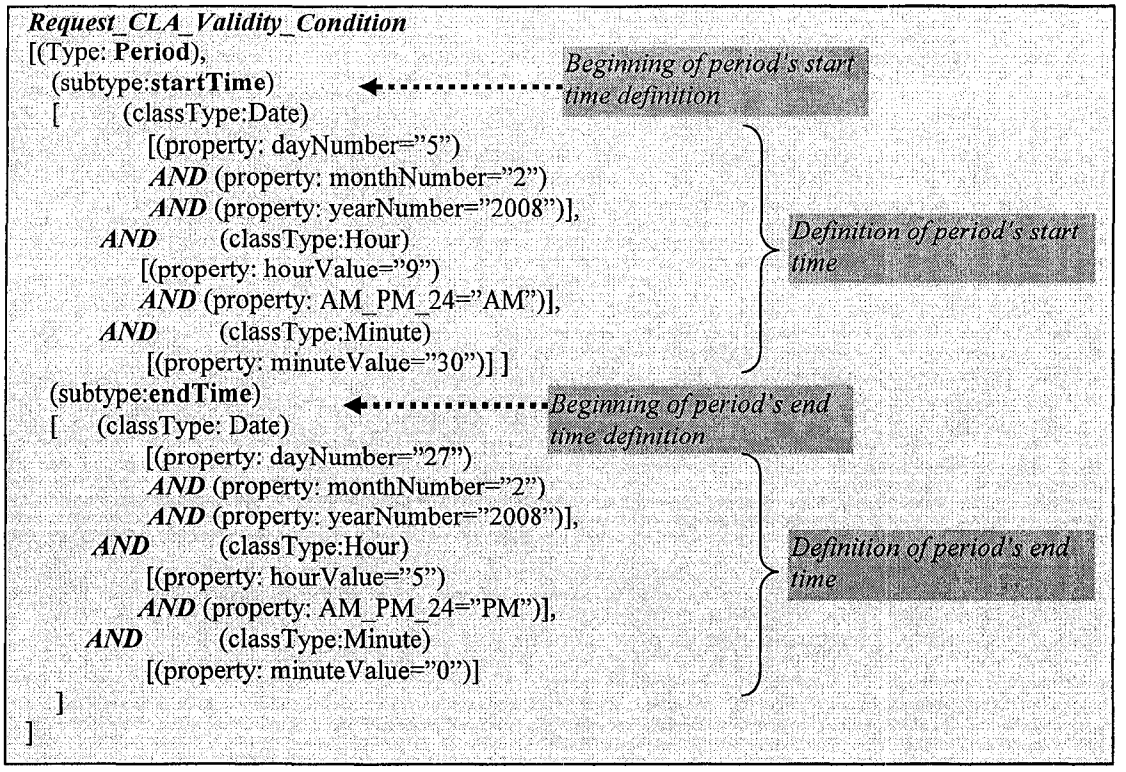


Figure 5-4. CLA Validity Condition example

5.2.2.2. Logical Conditional CLA Validity Conditions

Logical conditional CLA validity conditions are used when Clients are interested in activating or deactivating their CLAs following upon the occurrence of specific events – for instance, when the light status of room B502 switches to the 'ON' state, or when a user's status is set to 'NOT BUSY' and he is located in his office. These requests require Clients to have access to all ontology classes, individuals and properties included within their requests in order for them to negotiate a conditional CLA validity condition request. In the first example, then, the Client must have access to a set of rooms (class), more specifically to room B502 (individual), and finally, to the light status of that room (property). The same requirements apply to the second condition.

Conditional CLA validity conditions requests are divided into two separate groups:

1. WHILE group: Activates the CLA while the condition holds. The Client defines a logical condition using accessible context classes, properties, and individuals presented to the Client within its View documents. The Client's condition must generally refer to the exact individual(s) and property value(s) that need to be met prior to a CLA's activation.

```
Request_CLA_Validity_Condition
[(Type: LOGICAL_CONDITIONAL),
  (subtype:WHILE)
  [ (classType:Person, individual = "Mike")
     [(property: hasCurrentLocation classType: Room)
          [(Property: hasRoomType = "Office")],
          AND
          [(Property: hasOwner classType: Person)
               [(Property: hasFirstName = "Mike")]
          ]
          AND
          [(Property: isRoomWithinBuilding classType: Building)
               [(Property: hasBuildingName = "CBY")]
               AND
               [(Property: hasBuildingNumber = "161")]
               AND
               [(Property: hasAddress classType: GeographicalAddress)
                  [(Property: hasStreet classType: Street)
                       (Property: hasStreetName = "Louis Pasteur")]
                  AND
                  [(Property: hasCity classType: City)
                       (Property: hasCityName = "Ottawa")]
                  AND
                  [(Property: hasProvince classType: Province)
                       (Property: hasProvinceName = "Ontario")]
                       AND
                  [(Property: hasPostalCode = "K1N 6N5")]
               ]
          ]
     ]
  ]
]
```

Figure 5-5. Logical Conditional CLA validity condition

An example could be a CLA that remains active as long as the user "Mike" is located within his office. This is shown in the sample code in figure 5-5. Upon Mike's arrival into his office, the established CLA is activated, and it remains activated until Mike's exit from the room. The request shown in figure 5-5 illustrates some of the information that a Client may need to provide in order to form a clear WHILE logical conditional CLA validity request. It is important to have an indication of the

individual [*(classType: Person, Individual="Mike")*] whose current location governs the CLA activation. To simply specify the type of room (*office*) Mike must be located in, is usually not enough. Mike might be associated with a number of office rooms – hence, the need to specify which office is being considered in the condition. In this example, then, the building within which the office room was located, as well as the building's address have been supplied by the Client in its request.

Clearly, the use of ontologies in constructing negotiation requests gives both context Clients and context Providers the flexibility to express their requests with as much or as little detail as necessary. The example shown in figure 5-6 is only one possible method of expressing the Client's request. Using the Person and Location ontology subsets provided within its View, the Client can create any context request shape that fits its needs.

```
Request_CLA_Validity_Condition
[(Type: START END TRIGGER),
    (subtype: START TRIGGER)
    [(classType: Room)
        [(Property: hasOwner classType: Person)
            [(Property: hasFirstName="Mike")]
        ]
        AND
        [(Property: hasRoomType = "bedroom")]
        AND
        [(Property: hasTemperatureLevel >= "25")]
    ]

    (subtype: END TRIGGER)
    [(classType: Room)
        [(Property: hasOwner classType: Person)
            [(Property: hasFirstName = "Mike")]
        ]
        AND
        [(Property: hasRoomType= "bedroom")]
        AND
        [(Property: hasTemperatureLevel <= "20")]
    ]
]
```

```
Request_CLA_Validity_Condition
[(Type: START END TRIGGER),
    (subtype: START TRIGGER)
    [(classType: Room, individual= "bed236")
        [(Property: hasTemperatureLevel >= "25")]
    ]

    (subtype: END TRIGGER)
    [ (classType: Room, individual = "bed236")
        [ (Property: hasTemperatureLevel <= "20")]
    ]
]
```

Figure 5-6. Trigger CLA Validity Condition

2. START/END TRIGGERS group: Activates the CLA on the occurrence of the START trigger condition. The CLA remains active until the END trigger condition is validated. A sample Trigger Condition CLA Validity request may include a CLA that is activated whenever Mike's bedroom temperature rises above 25°C and de-activates when the temperature is below 20°C. This hypothetical situation is shown in figure 5-6, using a high-level message representation. Two methods of expressing

the Client's request have been provided illustrating the various formats Clients can utilize to express their requests through ontologies. In 5-6 (top example) the Client does not include a direct reference to the Room individual, whose temperature it requires to be monitored as a condition for activating and de-activating its CLA. Thus, the Client provides a high-level description of the room, such as the name of its owner (Mike) and its type (bedroom). Assuming that 'Mike' is the owner of only ONE bedroom, the condition would be sufficient to describe the Client's condition. An easier and more direct method of expressing the Client's request would be to have a reference to the exact Room *individual* on which the condition is dependent. This is expressed in the lower half of Figure 5-6. The only thing needed is the individual's ID (bed236 in this case) and the activation conditions (here the temperature levels).

```
Request_CLA_Validity_Condition
[(Type: Hybrid Periodic Conditional)
  (subtype:startTime)
  [ (classType:Date)
      [(property: dayNumber="5")
       AND (property: monthNumber="2")
       AND (property: yearNumber="2008")],
       AND  (classType:Hour)
       [      (property: hourValue="9")
              AND (property: AM_PM_24="AM")],
              AND (classType:Minute)
              [     (property: minuteValue="30")]
      ]
  ]
  (subtype:endTime)
  [ (classType: Date)
      [(property: dayNumber="27")
      AND (property: monthNumber="2")
      AND (property: yearNumber="2008")],
      AND  (classType:Hour)
      [      (property: hourValue="5")
             AND (property: AM_PM_24="PM")],
             AND (classType:Minute)
             [     (property: minuteValue="0")]
  ]
  (subType: Condition)
  [ (classType:Person, individual = "Mike")
    [(property: hasStatus != "BUSY")
  ]
]
```

Figure 5-7. Hybrid CLA Validity Condition

5.2.2.3 Hybrid CLA Validity Conditions

The third type of CLA Validity Conditions combines the previous periodic and logical conditional methods in order to produce a more expressive CLA validity condition. There are two methods of combining the previous two methods; Periodic Conditional, and Conditional Periodic.

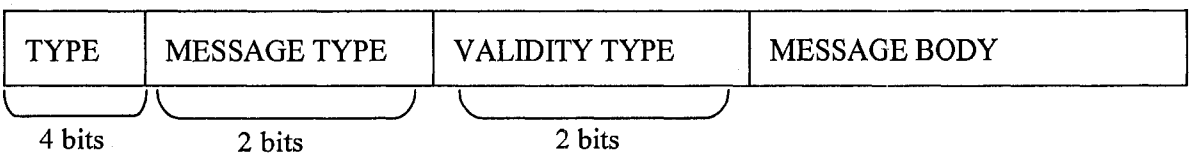
A. *Periodic Conditional Hybrid CLA Validity Condition (P-C)*: Refers to logical conditions embedded within a period. The period is used as a monitor to the embedded condition. Therefore, the logical condition is not evaluated until the current system time is located within the activation period of the CLA. Once the current system time is

within the valid period of activation, the logical condition is evaluated. During that period, the CLA is activated whenever the logical condition evaluates to "true", and it is deactivated otherwise. The CLA thus oscillates between the activated and deactivated states according to the logical condition during the activation period.

B. *Conditional Periodic Hybrid CLA Validity Condition (C*-P): Refers to periodic conditions embedded within logical conditions. The CLA remains inactive until the logical condition is validated. The logical condition is no longer needed Upon validation, and only the periodic condition monitors the activation of the CLA. The CLA remains active until the end of the period after which the CLA is terminated.

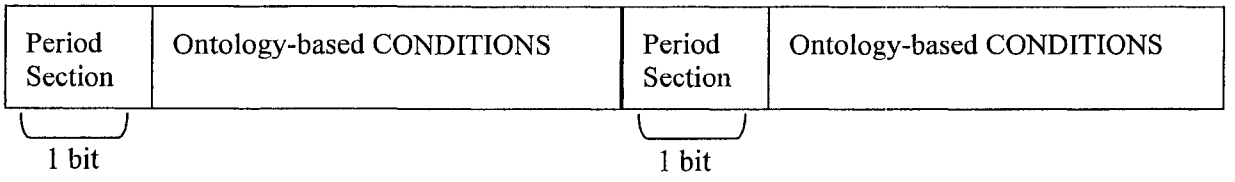
A possible Periodic Conditional (P-C) Hybrid CLA validity condition is provided in Figure 5-7. In the example shown, a Client is requesting its CLA to be activated between 7:25 A.M. on the 3rd of March 2008 and 2:00 P.M. on the 26th of September 2008, but only if the individual 'Mike' has a status of 'not busy'. Therefore both P-C and C-P CLA validity conditions requests require the offer's sender to indicate the period's start and finish times, as well as the logical condition that must be validated.

All three aforementioned CLA Validity Condition request messages (Periodic, Logical Conditional, and Hybrid) can be translated into the message structure shown below:



The message structure is similar to that of previous messages. The 4-bit TYPE field is loaded with '0100' from table 1, informing the receiver that the message is of type CLA Validity Condition. A 2-bit MESSAGE TYPE field follows, announcing whether the message is a new offer, an acceptance of a previous offer, or a rejection of a previously received offer. Since there are three types of CLA validity conditions Clients and Providers can utilize in their negotiations, a 2-bit VALIDITY TYPE field has been included. Clients and Providers can therefore negotiate Periodic (00), Logical Conditional (01), and Hybrid (10) CLA validity conditions.

The message's BODY structure is directly related to the value provided in the previous VALIDITY TYPE field. As we discussed earlier, Periodic validity messages require the sender of the offer to define the *Start* and *End* times of the CLA. Thus a Periodic message BODY can be generalized as follows:



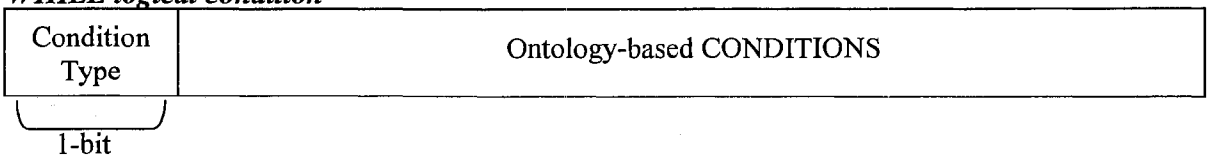
A 1-bit Period Section field is provided so the sender can inform the receiver of the message whether the conditions in the subsequent fields refer to those of the Start of the period (0) or of the End (1).

Unlike Periodic CLA validity conditions, Logical conditions were divided into WHILE and START/END TRIGGERs groups. This necessitates a 1-bit field indicating which of these two choices the message refers to. Choosing the WHILE group simply requires that the sender to follow the 1-bit group choice field with the condition(s) it requires to be evaluated for CLA validity.

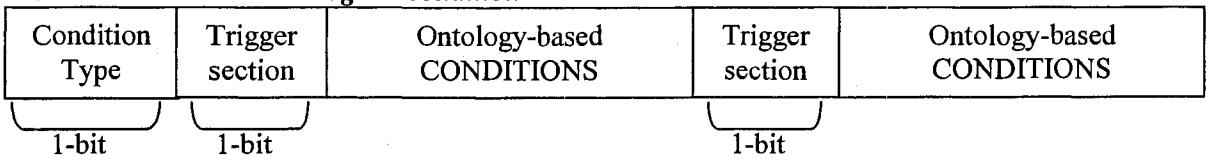
Conversely, choosing the START/END TRIGGER group requires the division of the message body into two sections similar to those of the Periodic CLA validity message: one section for the Start trigger condition, and another for the End trigger condition. Each of these two sections is preceded by a 1-bit field indicating which of these two triggers is being defined - the Start trigger (0) or the End trigger (1).

MESSAGE BODY fields for logical conditional CLA validity conditions are summarized as follows:

WHILE logical condition



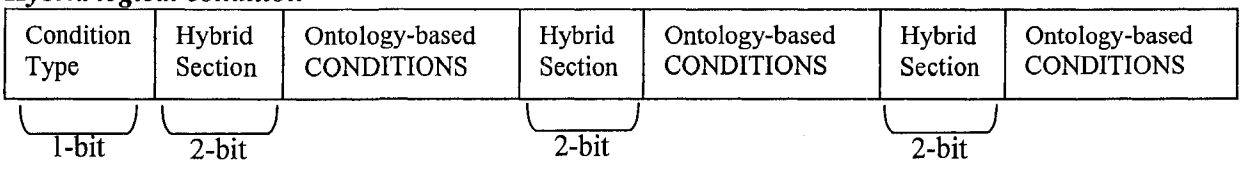
START/END TRIGGER logical condition



The third group of logical conditions, the HYBRID group, was also divided into two categories. Given the choice of PERIODIC-CONDITIONAL and CONDITIONAL-PERIODIC hybrids, a 1-bit Condition Type field (similar to that presented within the previous Logical conditional validity message) is needed in order to indicate the choice of P-C(0) or C-P(1). Both P-C and C-P hybrid conditions require that the sender of the offer define three sections within the message body: the logical condition, the period's start conditions, and the period's end conditions.

Given the above requirements, the general structure of the MESSAGE BODY of a Hybrid CLA validity condition is displayed below:

Hybrid logical condition



Three 2-bit Hybrid Section fields allow the sender to indicate to the receiver which of the three hybrid sections is being defined - the logical condition (00), the period's start condition (01), or the period's end(ing) condition (10).

OWL-based ontologies are RDF graphs can in turn be seen as sets of RDF subject-predicate-object triples. These graphs are written in many syntactic forms such as RDF/XML, as long as they result in the same underlying RDF triples. OWL documents consist of class axioms, property axioms, and facts about individuals. Classes provide an abstraction for groups of resources with similar characteristics. Every OWL class is associated with a set of extensions or instances referred to as individuals.

Properties within OWL are divided into two main categories:

- *Object* Properties: Link class individuals to other class individuals. Object properties are used to represent the relationships that may exist between different OWL classes. A sample object property may be the 'belongsTo' property that links devices to their owners:

```
<owl:ObjectProperty rdf:ID="belongsTo">
    <rdfs:domain rdf:resource="#Device"/>
    <rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>
```

- *Datatype* Properties: Link individuals to data values. Datatype properties may have a range of values in floats, Booleans, strings, integers or other types. An example of a datatype property may be the 'hasStreetName' property that the expression of street names in the form of strings:

```
<owl:DatatypeProperty rdf:ID="hasStreetName">
    <rdfs:domain rdf:resource="#Street"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

Since our context-level negotiation protocol is based primarily on the structure of OWL-based ontologies, this dependency must be reflected in the protocol's body structure. In the CLA Validity Conditions protocol messages provided earlier, all message had one or more fields marked as 'Ontology-based CONDITIONS'. In those fields the message sender, be it the Client or Provider, expresses the conditions that must be evaluated for the CLA to be activated and de-activated. For instance in the Periodic CLA validity conditions, two such fields existed; in one, the sender proposed the period's start time while the other field

Option Number	**General Message Structure**
Option -1	Define the *class* → define the *datatype property* → define the *condition* of the property value → define the property *value*.
Option -2	Define the *class* → define the *object property* → define the *class* → define the *object property* → ... → define the *datatype property* → define the *condition* for the property value → define the property *value*.
Option -3	Define the *class* → define the *individual* → define the *datatype property* → define the *condition* of the property value → define the property *value*.
Option -4	Define the *class* → define the *individual* → define the *object property* → define the *class* → define the object *property* → ... → define the *datatype property* → define the *condition* for the property value → define the property *value*.

Table 3. Message structure condition options

contained the period's end time. Expressing the start and end times requires having access to the Time ontology and forming requests that take the structure of ontologies into account. The same concept applies to 'Ontology-based CONDITIONS' fields for logical conditional and hybrid CLA validity conditions messages.

To simplify the process of forming the aforementioned ontology-based conditions, we have provided four different combinations of classes, individuals, and properties. All four cases are shown in Table 3.

The options illustrated in table 3 provide the senders with options for forming their conditions, based on all major combinations that may exist within ontologies. Each of the four options will be explained in further detail below.

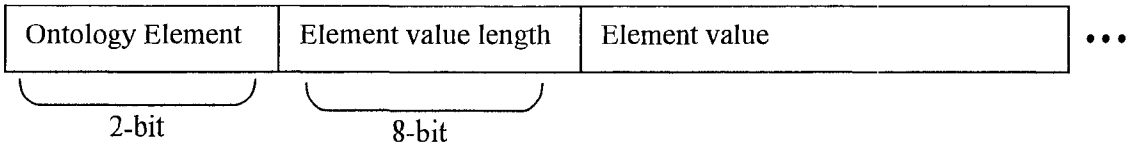
A. OPTION -1:

Conditions that depend on the current value of some context information usually utilize Option -1 provided in table 3. For example, if we assume that a Client is requesting to have its CLA activated between March 21st, 2008 and May 20th, 2008 then Option -1 would be sufficient for forming the Client's requests. The Client would therefore refer back to the *Time* ontology provided within the Global Skeletal Ontology (a sample Time ontology was provided in section 3.2.2). As seen in figure 3-5, *days*, *months*, and *years* were datatype properties of the *Time* subclass of Date. Thus to define the period's start date the Client indicates the class (Date), the property it is interested in (*dayOfMonthNumber*), the condition related to the property (==), and the property value (21). This is repeated for the start month and year.

To summarize, the process of declaring the periodic CLA validity start period condition can be generalized as follows:

*[**Class**(Date), **Datatype Property**(dayOfMonthNumber), **Condition**(==), **Property value**(21)] AND*
*[**Class**(Date) **Datatype Property**(monthName), **Condition**(==), **Property value** (March)] AND*
*[**Class**(Date), **Datatype Property**(yearNumber), **Condition**(==), **Property value**(2008)]*

Translation of the above high-level representation of the condition is intuitive, and follows the repetitive three-field format as illustrated below:



The 2-bit Ontology Element field is used to indicate which of the four possible element categories in an ontology is being defined: class (00), individual (01), condition (10), and property (11). If the choice is a condition, then one of the first six possible condition choices provided in Table 4 is placed. Following the condition field would be the 8-bit value length field, which represents the number of bits occupied by the value stored within the next field.

If the choice is not a condition, the name of the ontology element is provided, preceded by an 8-bit field containing a value that represents the length of the following string. The element value (which could be the class name, the individual's id, or a property name) follows the 'length' field.

1. '0000': == (Equal To)	2. '0001': != (Not Equal To)
3. '0010': > (Greater Than)	4. '0011': > (Less Than)
5. '0100': >= (Greater Than or Equal)	6. '0101': <= (Less Than or Equal)
7. '0110': hasID. sets the request's ID	8. '0111': hasQuality- reference to QoC
9. '1000': AND	10. '1001': OR
11. '1010': (//Open parenthesis	12. '1011':) //Close parenthesis

Table 4. Possible values for condition fields

Each of the two above choices are repeated until the complete request has been defined. Repetitions are separated from each other using one of the two possible 4-bit fields, AND (1000), or OR (1001) shown in table 4. To define the day of month (21st) in our Periodic CLA validity condition example the offer's sender forms a message body similar to the partial example shown in figure 5-8

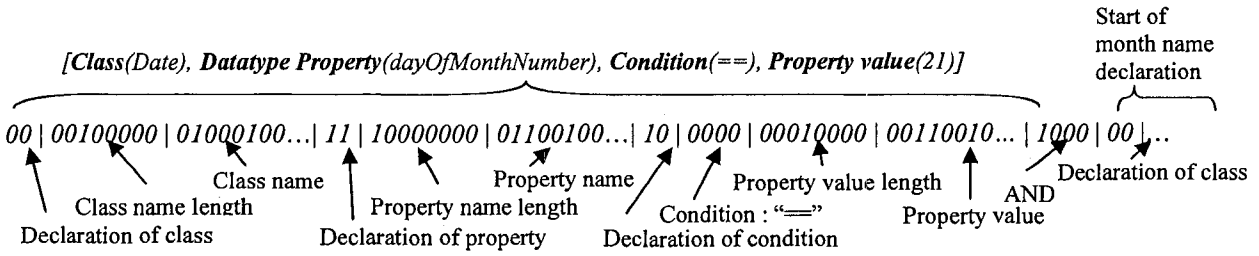


Figure 5-8. Ontology-based option 1 condition structure

B. OPTION -2

During the presentation of the Time ontology in section 3.2.2 we explained that users of the ontology must be given the ability to loosely express time in relation to certain events or to specific times. This ability was provided through the *RelativeTime* subclass of *ComparativeTime*. If the Client sends a Periodic CLA validity condition request with start or end times expressed in relation to a specific start time without requiring a concrete value, Clients could utilize the section condition option to express their requirements.

One example is a Client interested in activating its CLA *AFTER* 9:00 A.M. In this case, the exact time of CLA activation is unimportant as long as it is not activated before that time. Although this example may not be very likely, it is used to show the diverse ways in which Clients and Providers can express their condition requirements and abilities. In an approach similar to that in the previous example, a high level representation of the sender's message body can be expressed as follows:

[Class(Before), Object Property(inRelationToExactTime), Class(Hour), Datatype Property (hasHourValue), Condition(==), Property value(9)] AND
[Class(Before), ObjectDatatype Property(am_pm_24), Condition(==), Property value(A.M.)] AND
[Class(Before), Object Property(inRelationToExactTime), Class(Minute), Datatype Property (hasMinuteValue), Condition(==), Property value(0)]

The sender declares its interest in using the *Before* subclass of *RelativeTime* and its object property '*inRelationToExactTime*', which were shown earlier in figure 3-5. The object property links the relative time to an exact time whose hour and minute are declared using the *Hour* and *Minute* subclasses respectively, along with their needed datatype properties such as *hasHourValue* and *hasMinuteValue*.

In keeping with to the steps used to illustrate Option-1 in section A, the message structure of the Option-2 example in this section is shown in figure 5-9 with the partial message body.

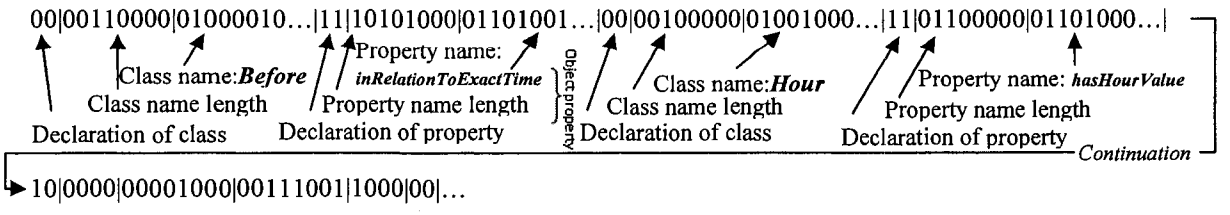


Figure 5-9. Ontology-based option 2 condition structure.

C. OPTION -3

Conditions that depend on context values of specific class instances (individuals) must use the format presented as Option-3 in table 3. Periodic CLA validity conditions did not involve the need to reference a certain user within the system. They also did not involve context values of specific rooms within the context Provider's environment. Unfortunately, a large percentage of ontology-based conditions that are needed to express the sender's requirements must refer to a specific individual present in the Client's View whose context values affect the condition's outcome.

An example of this is a logical conditional CLA validity condition request message. The START/END Trigger CLA validity request message presented in figure 5-6 provided a hypothetical situation where the Client was requesting to trigger the activation of its CLA once the temperature value within a room was equal to or greater than 25 °C. The room the Client was interested in was a specific instantiation of the *Room* class with an ID of '*bed236*'. This instantiation is referred to as an *individual*. Permitting Clients to freely express this type of condition requires using Option-3 from table 3, using the high-level message body representation:

*[**Class**(Room), **Individual**('bed236'), **Datatype Property**(hasTemperatureLevel), **Condition**(>=), **Property value**(25)]*

The above condition is translated into the bit-level formats of our context-level negotiation protocol in a manner similar to that of the earlier examples with the addition of a field for information about the Room individual. This is shown in figure 5-10.

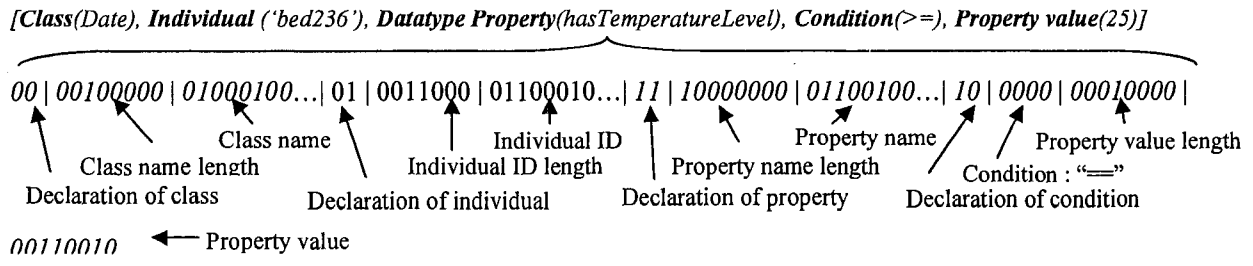


Figure 5-10. Ontology-based option 3 condition structure

D. OPTION -4

The fourth option in table 3 shows the final ontology-based condition format that permits Clients and Providers to form their context conditions. This option is similar to that provided in option-3, with the addition of direct reference to the individual whose object-type property (relating ontology classes to each other) is required in expressing the sender's ontology-based condition.

Since we did not provide an example that may be categorized as an Option-4 type of context condition, we provide here a sample logical conditional CLA validity request message of the WHILE group. The sample message in figure 5-6 proposes a situation where a Client is interested in activating its CLA while the current activity of the user 'Mike' has a privacy status of 'public'. This means that Mike's current activity may be attended to or viewed by any person. A high-level representation of the delivered message is provided in figure 5-11. In this example, the individual 'Mike' is needed who is an instance of the *Person* class. An object type property provides a relationship between the *Person* class and the *Activity* class, and the datatype property of the latter must have a value equivalent to 'public'. Therefore as with the previous three options, we provide the bit-level sequential message representation in figure 5-12 for option-4.

```
Request_CLA_Validity_Condition
[(Type: WHILE)
    [(classType: Person, individual = "Mike")
        [(Property: hasCurrentActivity classType:Activity)
            [(Property: hasPrivacyStatus == "public")]
        ]
    ]
]
```

Figure 5-11. WHILE CLA Validity Condition

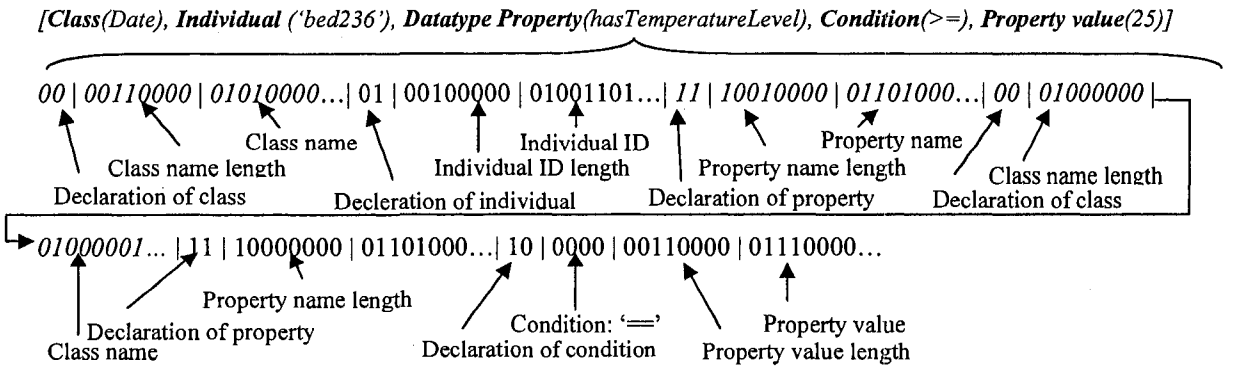


Figure 5-12. Ontology-based option 4 condition structure

All examples of ontology-based conditions used thus far to explain the four condition options in table 4 only involve the use of simple single conditions, such as the current activity of a user or the current temperature status of a room. In contrast, it is more than likely that situations might arise where the CLA validity condition is dependent on two or more contexts within the environment. An example would be a case where a Client is interested in activating its CLA while both of the following conditions hold true:

1. *An individual 'Mike' is located outside room 'B502'*, and
2. *When the door status of room 'B502' is set to 'LOCKED' OR the light status of that room is set to 'OFF'.*

As this example suggests, extending the concepts presented thus far to include the ability to compose ontology-based conditions involving multiple sub-conditions raises a new requirement for the protocol: it must inform receivers of the order and priority in which conditions should be evaluated while also giving message senders the flexibility to express complex conditions.

Referring to the above example, we make the following assumptions:

'A' represents the condition: individual 'Mike' is located outside room 'B502'.
'B' represents the condition: room B502's door status is == 'LOCKED'.
'C' represents the condition: room B502's light status is == 'OFF'.

Therefore, expressing the CLA validity condition request the Client sends to the context Provider according to the protocol information we have provided thus far, will inform the Provider that the Client is requesting the CLA's activity to be bound to the following condition: *While(A and B or C).*

However, this is not the condition the Client is requesting. The Provider's/Receiver's interpretation of the validity condition activates the CLA while either:

1. Mike is outside room B502 *and* the room's door is locked, or
2. When the light status of room B502 is set to OFF.

In mathematics and Boolean algebra the use of brackets, "(" and ")" indicates the logical order in which mathematical operations should be executed. If all operations present within

a condition are of equal priority, then they are performed starting with those present in the innermost brackets and expanding outwards until all operations have been performed.

We have adopted the same algebraic concept in our context-level negotiation protocol. Clients and providers can compose complex ontology-based conditions and requests, by employing brackets as in entries 11 and 12 of Table 4 to provide message receivers with the proper method for interpreting the received conditions. Consequently, the Client would have been able to send a logical conditional CLA validity request message requesting the CLA's activation *while (A and (B or C))*. Unlike the earlier example, this condition matches the Client's request.

The following high-level representation of the condition shows the steps taken to form the above condition:

*[**Condition**("("), **Class**(Person), **Individual**("Mike"), **Property**("hasCurrentLocation"), **Class**(Outside), **Property**("inComparisonTo"), **Class**(Room), **Individual**("B502"), **Condition**("AND"), **Condition**("("), Class(Room), **Property**("hasDoorStatus"), **Condition**("=="), **Property Value**("LOCKED"), **Property**("OR"), **Class**(Room), **Property**("hasLightStatus"), **Condition**("=="), **Property Value** ("OFF"), **Condition**(")"), **Condition**(")").]*

Moreover, it is the responsibility of the sender of the condition to ensure that all brackets balance out – that is, that each open bracket has a corresponding closing bracket.

5.2.3. Context Requests

Once the CLA Validity Conditions have been negotiated, Clients begin negotiating their core context needs with the Providers. This level of negotiation refers to the context information Clients wish to receive upon activation of their CLAs; according to the conditions negotiated during the CLA validity conditions step.

As expected, the context information Clients request must not exceed the context access rights provided within their context Views. Consequently, to compose their respective context requests, all context request messages utilize the View that Clients had received during the initial stages of negotiations.

Context requests are divided into two classes according to the data types returned to the Client:

1. ***Notification Requests***: A notification is delivered to the Client when the agreed upon conditions occur. An example is a notification sent whenever the door status of room 'B502' switches to the 'OPEN' state, or whenever the current traveling speed of a vehicle exceeds 100 km/h. As with CLA validity conditions, complex context requests can be composed of multiple conditions in different orders and logic operations. The high-level sample message displayed in figure 5-13 shows a notification request indicating the Client's interest in receiving a notification if the current research topic of the individual 'Yousif' is equivalent to 'Ontology-based negotiations and context-level agreements'.

2. ***Value-delivery Requests***: unlike notification requests, value-delivery requests require the return of specific values to the Client. The current position of a person, their current activities, the status of devices in a room, the load on network devices, and many other characteristics are all specific values which a Client may have interest in receiving.

```
Request_Context
[(Type: Notification, ID="2897")
    [(Class_Type: HomeResident individual="Yousif")
        [(Property: engagedIn Class_Type: Research)
          [(Property: researchTopic == "Context-Level Agreements")]
        ]
    ]
]
```

Figure 5-13. Notification context request

As discussed in chapter 3, OWL-based ontologies can be viewed as sets of classes, properties, and individuals. Individuals (or instances) are instantiations of certain ontology classes. Thus each individual within an ontology belongs to a specific class. Additionally, ontology classes have properties that model interclass relationships (object properties), and properties representing characteristics/attributes present within each class, in terms of values such as strings, integers, floats, and other datatype properties. Extending these concepts has led to two conclusions. First, all context requests must include a direct reference to the class individual whose context information the Client is trying to access. Second, Value-delivery Requests can be broken down into two types; Property Value Requests and Class Value Requests.

```
Request_Context
[(Type: Property_Value, ID="2897")
    [(Class_Type: HomeResident individual="Mary")
        [(Property: engagedIn Class_Type: Research)
          [(Property: researchTopic)]
        ]
    ]
]
```

Figure 5-14. Property value context request

*A. **Property Value Delivery Requests***: These are context requests that require the Provider to return the specific values of context datatype properties. The values returned may be in the form of strings, integers, Booleans, floats, etc. Figure 5-14 shows an example of a high-level representation of a context request pertaining to the type of activity the individual 'Mary' is currently engaged in. In this request, the Client is interested in receiving Mary's research topic, assuming that the activity class to which Mary's current activity belongs is 'Research'. To form such a request, the Client informs the context Provider that the current message is a context request message (Request_Context) of type (Property_Value). Thus, the Client must supply the Provider with enough details about the exact information it is interested in receiving. To do so, the class type and instantiation ID (individual name) whose property values the Client is requesting must be indicated within the request message. In addition, the name of the property whose value the Client is interested in receiving must also be included (researchTopic).

If the above context request is accepted by the context Provider, the name of the research topic 'Mary' is currently engaged in and would be delivered to the Client.

B. *Class Value Delivery Requests*: Property values are not the only type of context information Client may need. Context information modeled using ontologies gives Clients the ability to represent knowledge with varying degrees of detail. This concept can be seen in all ontologies presented in chapter 3, where educational organizations were broken down into universities, and universities were divided into faculties. The faculties were in turn composed of departments. Similarly, the Activity class was divided into many activity types such as Meetings, Research, Events, Courses, and so on. Each was modeled as an extension of the Activity class. Outdoor spaces were divided into classes representing countries, provinces, cities and streets, and indoor spaces were categorized as buildings, floors, and rooms.

```
Request_Context
[(Type: Class_Value, ID="465634")
    [(Class_Type: HomeResident individual="Mary")
        [(Property: engagedIn Class_Type: Activity)
        ]
    ]
]
```

Figure 5-15. Class value context request

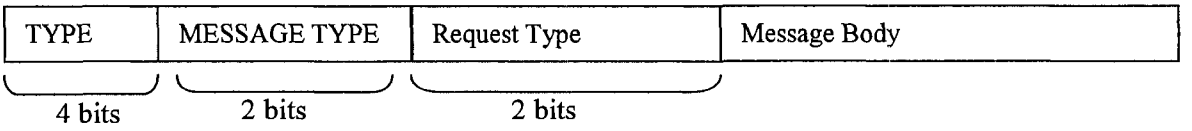
This university example shows that there is a hierarchy in ontology-based context models and this hierarchy can be used to form Clients' context requests. Such situations might arise

if a Client is only interested in knowing whether the location of an item is indoors (IndoorSpace class) or outdoors (OutdoorSpace), or knowing the class of activity an individual is engaged in without reference to any other properties (such as the activity name or the activity's start and end times). A Client may also be interested in receiving information about the type of student an individual is classified as, such as undergraduate or graduate. If an individual is a graduate student, then one might want information on which of the three subclasses (s) he belongs to; Masters, PhD, or Postdoctoral.

Such scenarios do not require the context Provider to supply Clients with information about the specific values of datatype or object properties. Alternatively, the Provider supplies Clients with the name of the ontology class in which the Client is interested in receiving and which the requested individual currently belongs to. This type of context request is referred to as a *Class Value Delivery Request*. A sample high-level Class Value context request is provided in figure 5-15. Unlike the previous example shown for requesting the datatype property, (such as the research topic for the current research activity of the individual Mary), the current example shows a Client's request for the current Activity subclass to which Mary belongs.

Notification requests required Clients to specify the conditions and property values that needed to be met prior to the notifications' delivery by the Provider (for example, room temperature > 20°C). On the other hand, value-delivery requests only required Clients to identify the name of the property whose value is to be supplied to the Client (in Property Value requests), or the name of the class whose respective subclass is to be delivered by the Provider (Class Value requests).

Using the concepts we have introduced in this section, we arrive at the following structure for Context Request messages:



As with all previous context-level negotiation protocol messages, context request messages start with a 4-bit field used to indicate the type of message being exchanged. Referring back to table 1, the value '0101' is used in this case to indicate that the message is of type Context

Request. The Message Type field contains the 2-bit value that is used to indicate whether the current message is a new/counter-offer, a rejection of a previously received offer, or an acceptance of the previous offer. Finally, since we have divided context requests into two main types (notification requests and value requests), and since value requests were further divided into property value requests and class value requests, hence the need for a 2-bit Request Type field is necessary. Three possible values can be used within the Request Type field:

00- Indicates the context request message delivered is of type Notification request.

01- Indicates the delivered message is of type Property Value request.

10- Indicates the delivered message is of type Class Value request.

The remainder of the Message Body can be broken down into three main parts, two of which are used for identification and one for forming the Client's request. Every CLA negotiated between Clients and Providers may contain one or more context requests. Each of these context requests lists the type of context information that needs to be delivered to the Client categorizing this list according to the types of requests discussed above. It is therefore important to distinguish context requests from each other so that when context Providers deliver the context information to Clients, they can indicate which request the delivered context belongs. In addition, context Clients and Providers are required to negotiate the conditions that activate and deactivate delivery of each context request as well as the quality of context levels that must be met prior to delivery of the context information. These two requirements will be discussed further in sections 5.2.4 and 5.2.5, respectively. In both cases, Clients and Providers must clearly include a reference to the context request these validity conditions and quality levels belong to - hence the need for an identifier attached to each context request message.

The first four bits of the message body is filled with the value '0110' found in table 1, indicating that the request's ID is being defined. Since the request's ID can be any string the Client defines, then according to our earlier explanations, any string of unfixed lengths must be preceded by an 8-bit field indicating the length of the following string. The request's ID is therefore preceded by an 8-bit field for the ID's length, followed by the request's ID, and then followed by the request conditions. This process is summarized as follows:

0110	Request ID Length	Request ID	Context Request

4 bits 8 bits

Finally, the Client forms its Context Request according to information provided within its ontology-based Views and according to the format presented in table 3 in section 5.2.2. Consequently, the steps to forming the Context Request within the Message Body of the Notification context request from figure 5-13 can be summarized as follows:

*[**Class**(HomeResident), **Individual**("Yousif"), **Object Property**(engagedIn), **Class**(Research), **Datatype Property**(researchTopic), **Condition**("=="), **Property Value**("Context-Level Agreements")]*

Finally, translating the context request message and the above context condition into our context-level negotiation protocol produces the following message:

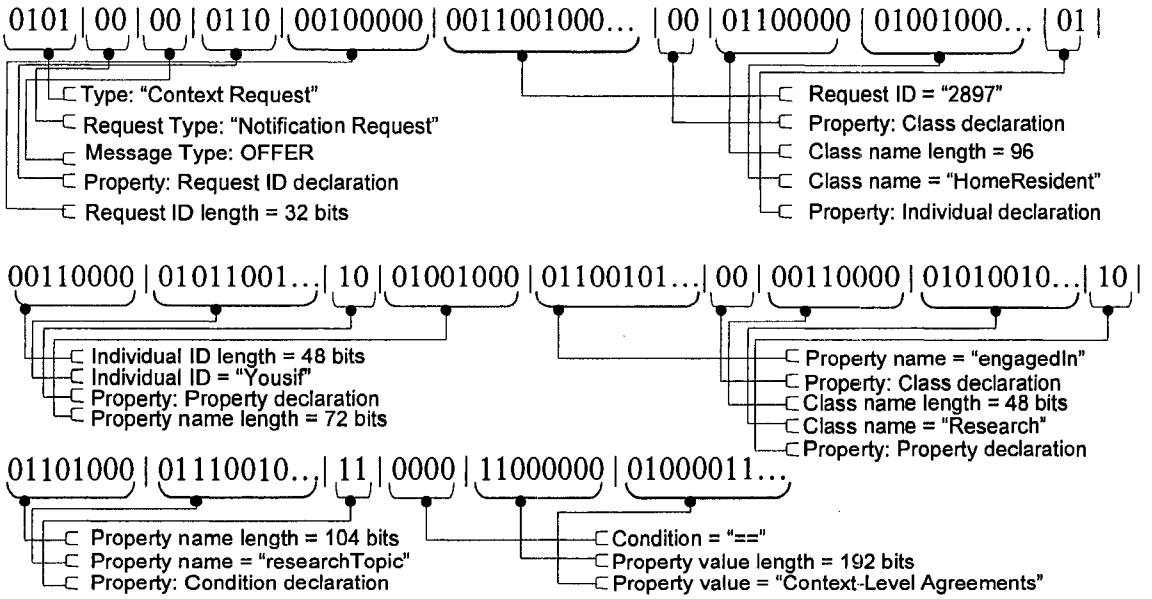


Figure 5-16. Encoded Notification context request message

In the event the context Provider accepts the Client's context request, negotiations proceed to the next step; where both parties exchange offers and counter-offers for conditions under which the requested context will be delivered. These conditions are referred to as *Context Validity* or *Context Applicability* conditions and will be discussed in further detail within section 5.2.4. If the Client's context request is rejected, the Provider will supply the Client with its counter-offer. This exchange of counter-offers is continued until both parties reach

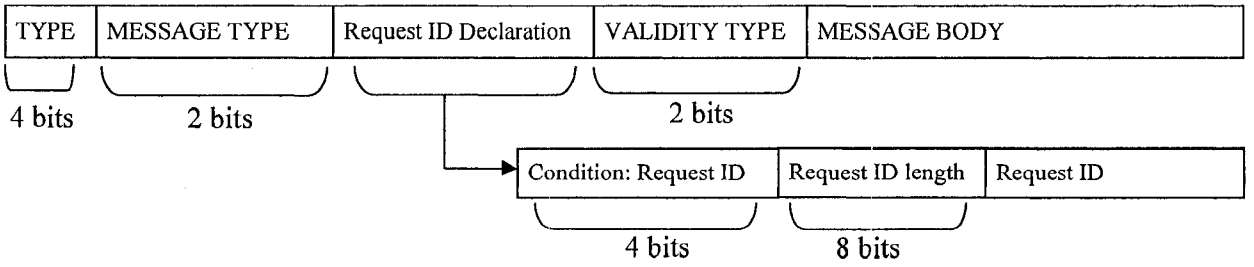
an acceptable agreement on the context information that needs to be delivered to the Client, and whether it is a notification, a property value, or a class value type of request.

5.2.4. Context Request Applicability Conditions

Every Context-Level Agreements (CLA) established between Providers and Clients can include within it one or more requests for context information. As was discussed in section 5.2.2, the activation and de-activation of every CLA is regulated through Client-specified conditions that have been negotiated with the context Providers (CLA Validity Conditions). Yet it is important to keep in mind that an active CLA does not necessarily mean that all context requests listed within the CLA will also be activated and thus delivered to the Client. Clients may be interested in restricting the delivery of context information from the Provider according to lists of conditions that are separate from the CLA's validity conditions. These context-request-specific conditions are referred to as *Request Applicability Conditions*.

For instance, if we assume that a Client and Provider have agreed that their established CLA is to be activated according to a periodic condition from April 20th, 2008 to September 2nd, 2008, then it is not necessary for the Client to be interested in receiving the context information it had continuously negotiated within the CLA during this whole period. Instead, the Client may wish to receive its context information under separate conditions such as its location being within a specific room, or a scheduled meeting taking place-keeping in mind that all conditions must not exceed the permitted context accessible by the Client in its Views received at the beginning of the negotiation session.

Each context request within a CLA whose delivery conditions are not equivalent to those of the CLA Validity condition(s) requires that the Client negotiate its Request Applicability conditions with the Provider. These applicability condition negotiations have the same steps and procedure presented in section 5.2.2 for the CLA Validity Conditions, as well as a reference to the context request whose applicability conditions as being negotiated. This reference is achieved by including the context request's ID within the exchanged messages; this results in the inclusion of request identifiers in context request messages (see section 5.2.3). Therefore, the general structure for Context Request Applicability Conditions messages can be summarized as follows:



The structure of context applicability conditions messages is clearly similar to that of CLA Validity messages, with the addition of the context request's ID. Indication of the request's ID starts with a 4-bit field containing the value '0110' retrieved from table 4, followed by an 8-bit field used to indicate the length of the string that follows (which is the Request ID in this case), and finally, by the Request ID itself.

The MESSAGE BODY field contains the conditions guarding the activation of context delivery. Delivery of context information listed within that context request commences upon validation of these conditions, since the CLA Validity conditions have also been validated (in other words, meaning that the CLA is currently active). Requests composed of two or more conditions utilize the AND (1000) and OR (1001) options presented in table 4.

Clients and Providers continue negotiating the context requests' applicability conditions until an agreement is reached by both sides. Once the two sides have reached an agreement, then the Client would have successfully negotiated its needed context information and the conditions under which these contexts would be delivered by the Provider. The final step for that context request is to negotiate the quality of context levels that the context information must meet prior to delivery by the Provider.

5.2.5. Quality of Context Requests

Context information acquired by Providers from different context sources contains many inconsistencies given sensor imperfections and the dynamicity of pervasive environments. Sensors differ in their abilities and context information provided to Clients is therefore directly affected by the context sources. As shown in section 3.2, the Global Skeletal Ontology must give Clients and Providers the ability to express the quality levels of modeled context information. As indicated, the ontology provided in [17] was adopted in modeling all quality of context information within our GSO. QoC was modeled as a set of quality parameters, such as Accuracy, Certainty, Error Margin, and so on. Every quality parameter

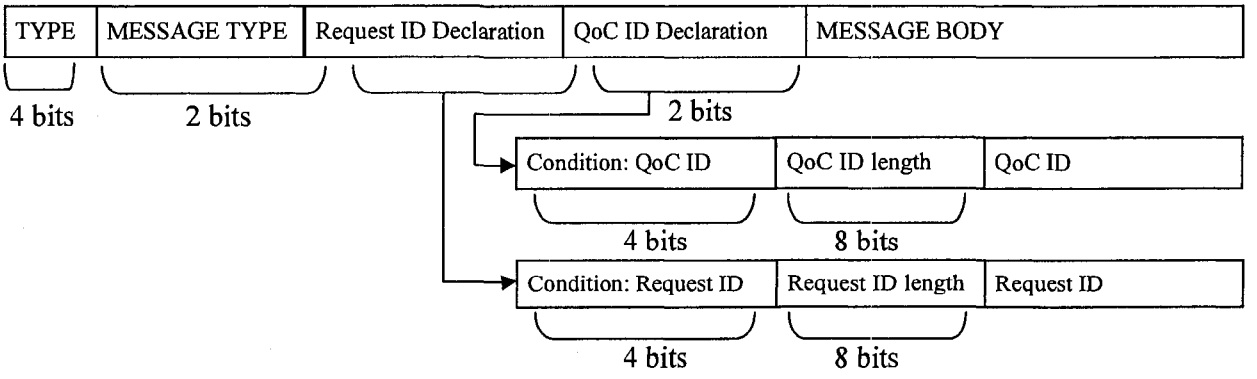
permitted the existence of relationship with a quality metric class that described the units, types, and values of the parameters. Freshness was another quality parameter whose values were described using the Time ontology to provide information about the total time elapsed since the last update has been applied to the stored context information values.

In general, Clients are also able to associate Quality of Context levels for context information they expect to receive from Providers. Clients should therefore be able to express QoC requirements for every context request established with Providers. This can be accomplished through Quality of Context Requests messages.

CLAs may contain one or more context requests. Each request lists a single type of context information the Client expects to receive from the Provider. Each request may also have its own applicability conditions that need to be met prior to delivery, and its own quality of context levels to be guaranteed before delivery by the Providers. An example may arise in a situation where a Client requests the current location of the individual 'Mary'. As explained earlier, each request message has a unique ID used as a reference. We will therefore assume that the reference given to this context request message was 's324'.

We will also assume that the Client is only interested in receiving Mary's location if the Provider can guarantee that the delivered values are at least 94% accurate, and have a freshness of less than 30 seconds. In other words, the information provided to the Client must be updated on the Provider's side at a rate faster than every 30 seconds.

Quality of context request messages follow the general message structure provided below:



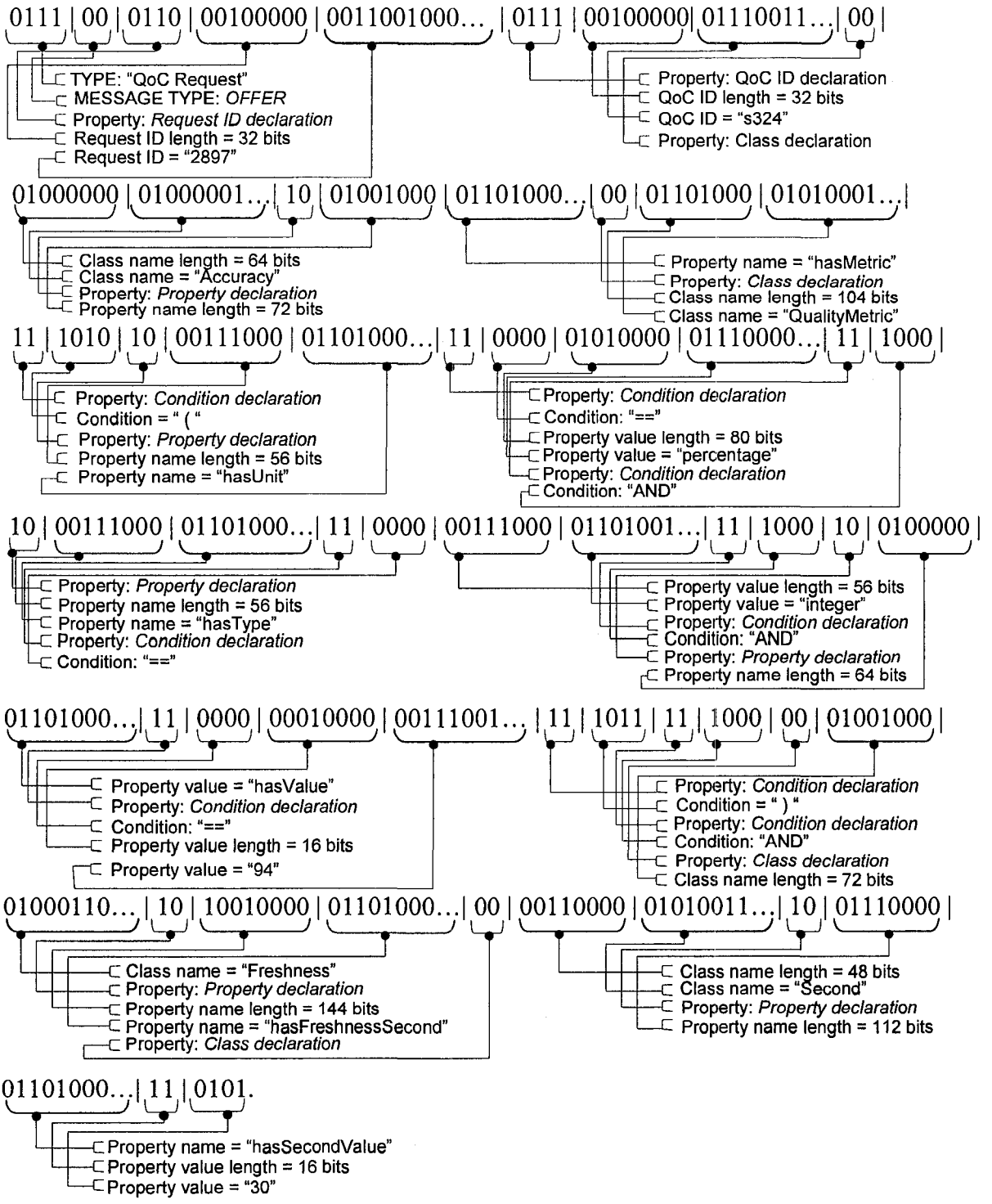


Figure 5-17. Encoded Quality of Context request message.

The message begins with the usual 4-bit header declaring the type of message being delivered - *Quality of Context* in this case. The 4-bit value of '0111' is here retrieved from table 1. The 2-bit message type is an indicator of whether the message is a new offer, a

rejection of a previous offer, or an acceptance of the previous offer. The request ID is used to inform the context Provider that the QoC request message refers to the context request who's ID is the same as the supplied ID. Just as each context request message was given a unique ID for future reference, each QoC request message is also given an ID for future reference if changes or cancelations would be later requested by the Client. Finally, formation of the Message Body follows the same ontology-based structure as that of CLA Validity Conditions.

The bit-level messages provided in figure 5-17 summarize the steps in forming the above sample QoC request's message. Strings requiring large numbers of bits have only been partially displayed, due to the lack of available space.

The context Provider receiving the QoC offer from the Client compares its internal abilities, actual quality levels its stored context information meet, and effects that accepting such request would have on CLAs established with other Clients. If the Provider agrees to the offer conditions, an acceptance is sent back to the Client. Otherwise, the context Provider sends the Client a rejection message followed by a counter-offer. The exchange of offers from both sides, continues until an agreement is reached.

Once an agreement has been reached, a Client completes a single context request within its CLA. This request includes the actual context information it expects to receive from the context Provider, the conditions under which the context information is to be delivered to the Client apart from the CLA validity conditions (Request Applicability Conditions), and the quality of context levels that must be met (Quality of Context Requests) by that context information prior to the information's delivery to the Client.

If the Client is interested in receiving further context information beyond that included within the established request, then the above steps must be repeated for each context the Client requires. Thus, steps 3, 4, and 5 from the message transition diagram of figure 5-1 (Context Request messages, Request Applicability message, and Quality of Context messages) must be repeated for all context information the Client requires within the context of the current CLA being established.

Once all Client-requested context information, applicability conditions, and quality levels have been agreed upon for all context information requested by the Client, then the Client would negotiate the final section of the CLA: the View update methods.

5.2.6. View Update Requests

Context information acquired by context Providers is highly dynamic. Context values adapt to changes in the environment, the reliability of the acquired context increases and decreases continuously, and the measurement accuracy of gained context also directly adapts to changes in context sources. Context sources supplying Providers with their context information may enter and leave the Provider's environment (assuming that they are mobile), and context sensors may stop functioning as a result of power failure or lack of a power supply. Failure of sensors may halt the Provider's ability to acquire a certain set of context information, or decrease the quality of acquired context information in terms of its accuracy, reliability, and freshness. These variations in the Provider's context acquisition abilities may have positive or negative effects on Client Views stored within the View Repository.

Context Providers establishing CLAs with incoming Clients direct a percentage of the Provider's resources and capabilities towards serving the requirements listed within these agreements. In addition, CLAs being established with high-priority Clients may require reduction in the quality of context information provided to Clients of lower priority who have already established CLAs with the Provider, or even stop delivery of certain context information to the former Clients altogether.

Any changes to the Provider's capabilities or to the context information accessible by Clients are reflected onto the Client's Context Views. As was discussed in chapter 3, these Views are stored within the View Repository, and Clients who have either established CLAs with Providers or are currently involved in negotiations with Providers obtain their respective Views as part of their negotiations. These Views are used by Clients to form intelligent context requests based on their context accessibility rights. Therefore, any changes to these Views directly influence established CLAs and CLAs under negotiations.

Consequently, there remains a need for a method of informing Clients of any changes to Views they had received earlier. Updated Views should be supplied to Clients according to the conditions each Client sees as necessary. This requirement is also achieved through

negotiations between Clients and Providers as part of the CLA establishment. Clients have a choice from four possible View update methods:

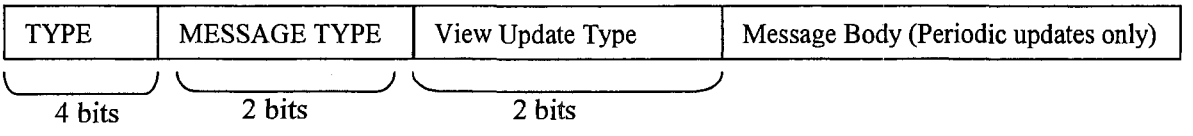
- *No_Update*: the Client is not interested in receiving any updates to the View it had received during the initial negotiation steps. Any changes to the View made within the View Repository within ConCoord on the Provider's side are not reflected back to the Client. Clients that do not wishing to receive updates to their Views usually fall within the category of those who establish short-term agreements unaffected by changes within Views that may not occur frequently.

- *Periodic*: the Client and context Provider agree on the frequency at which updated Views are delivered to the Clients - for instance, every 15 minutes. The Time ontology is used by the Client to form its periodic update request to the Provider. If the periodic update method is chosen by the Client, the latest View stored within the View Repository is delivered by the context Provider upon time expiration even if no changes have occurred since the last update.

- *On_Request*: the latest and most up-to-date views are delivered to the Client upon request. This approach reduces the number of times Views are delivered to the Client, when compared to the Periodic approach.

- *On_Change*: the Client is only supplied with its View when changes occur within the original Views stored in the View Repository. Increases or decreases in the amount of accessible context information a View are considered valid changes. Therefore, a Client who has chosen the *On_Change* View update option will always have the most updated View when referring to future requests or to modifications to its established CLAs.

View Update request messages are some of the simplest types of request messages in our context-level negotiation protocol. The general message structure is displayed below:



In light of previous negotiation messages discussed, the 4-bit Type header contains the value '1000' from table 1. The 2-bit Message Type field is used to indicate whether the message is

a new offer (00), a rejection (01) of a previous order, or acceptance (10) of a previous offer. If the Provider had agreed to supply the Client with its view using the *Upon_Request* option, then a Message Type field with the value (11) is used by the Client to indicate its interest in receiving the latest View available within the Provider's View Repository. During negotiations, the Client indicates its preferred View update method through the 2-bit View Update Type field, where 00 is used for the '*No_Update*' choice, 01 for the 'Periodic' update choice, 10 for the '*On_Request*' update choice, and 11 for the '*On_Change*' update choice.

All View update message types, with the exception of Periodic, do not have a Message Body. Rather, Periodic View update messages require that the Client (or Provider) to utilize the Time ontology to clearly indicate the time period between each View update the Client will receive from the context Provider.

5.2.7. Finalizing Negotiations

Once Client and Provider have agreed on the method used to inform the Client of its updated Views, then negotiations through our proposed context-level negotiation protocol are complete. The Client would have successfully established its Context-Level Agreement (CLA) with the context Provider, and it becomes the Provider's responsibility to guarantee delivery of the requested context information based on the agreed upon conditions and quality levels.

To do so, the CLA Policy Generator within ConCoord converts every established CLA into a set of policies. The generated policies serve two purposes in our context-aware system architecture as presented in chapter 4: first, the policies are used to modify Views stored within the View Repository. Secondly, they are utilized by the context Provider to monitor changes within the Global Concrete Ontology (GCO) that may trigger delivery of context information to the Clients.

The initial View generation steps performed during the Provider's startup relied on a set of policies entered manually by system management. These policies were ultimately used by the *View Generator*, and through access to the publicly accessible Global Skeletal Ontology (GSO) to generate Client context Views. Using ontologies, these Views listed all context classes, properties, relationships, and individuals accessible by a Client or groups of Clients. The generated Views were stored within a View Repository and supplied to incoming Clients

as they performed context-level negotiations with the Providers. Unfortunately, CLAs established by the Provider with Clients can reduce or increase the amount of context information Clients may be permitted to access, as well as the quality levels at which context is supplied. This may from a reduction in the Provider's capacity as it serves a greater number of Clients.

Consequently, Views stored within the View Repository need to be updated. To achieve this, an Automatic View Generation Policy Modifier first receives the newly-generated policies from the CLA Policy Generator, and then applies all necessary modifications to the initial set of policies stored in the Views Generation Policies. These policy changes trigger the View Generator to apply the necessary changes to these policies. Once the stored Views have been modified and updated, it might be necessary for ConCoord to notify some of the Clients with whom it had already established CLAs. The way in which these Clients are notified depends on the View Update methods the Clients had requested during their negotiations.

The second use of CLA-based policies concern the enforcement of established CLAs to meet the Clients' context requirements. Generated CLA-based policies are delivered to a Policy Repository within the Context Information Center in the Provider. This repository serves as a storage representing all rules generated by the established CLAs to meet the Clients' requirements. A PDP (Policy Decision Point) component is responsible for monitoring not only CLA-based policies within the repository, but also changes in context information and context values stored in the Global Skeletal Ontology. Any changes in context in the GCO that affect CLA conditions - such as the activation/deactivation of a CLA, or the activation/deactivation of context delivery - force the PDP to signal the CIC Access Point component to retrieve the necessary context information for delivery to the respective Clients.

Since the CIC (Context Information Center) does not communicate directly with Clients, the CIC Access Point retrieves the necessary context information from the GCO and sends this information to the ConCoord Access Point. The ConCoord Access Point sends in turn this context information to the respective Client, based on the Client's established CLA.

Since our context-aware system architecture and its context-level negotiation protocol constitute a new proposal, CLA-enforcement through policies remains a topic for future research and development. Currently, we have adopted a different approach to enforcing CLAs established between Clients and Providers. Each established CLA is converted into separate Java-based classes whose conditions and context requests are monitored by separate programs running their separate threads. The details of our implementation process will be discussed in further details in chapter 6. Nevertheless, our CLA-enforcement process can be summarized as follows:

As explained in this chapter, every Context-Level Agreement requires the negotiation of CLA Validity Conditions, which indicate when a CLA would be activated and deactivated. These conditions are converted into equivalent Java-based conditions, such as IF and WHILE statements that run within separate threads. These conditions are continuously evaluated until they are validated, at which time they instantiate and trigger new threads for every context request the Client has made within its CLA. Each newly-instantiated class monitors the Context Request Applicability conditions in a separate thread. As long as the applicability conditions hold, context information is retrieved from the Global Concrete Ontology and delivered back to ConCoord, the Provider's component responsible for communication with the Clients. Context information is then forwarded from ConCoord to the Client. Therefore a single thread runs for the CLA Validity Condition for each established CLA. Once the condition is validated, a thread for every context request is initiated, monitoring the context request's applicability conditions and retrieving the required context information.

5.3. Chapter Summary

In this chapter, we have presented the context-level negotiation protocol that completes our proposed context-aware system architecture. Our protocol allows context consumers to form complex context requests by utilizing our proposed ontologies. Context Clients were given the ability to negotiate the type of context information they require from the context Providers, the conditions under which those contexts should be received, and the quality that received context must meet prior to delivery by the Provider.

The negotiations presented resulted in context-level agreements (CLA) that formally defined the responsibilities of context Providers, thus giving them the ability to adapt and dedicate their resources to meeting Clients' needs.

To illustrate the functionality of our proposed context-aware system architecture and negotiation protocol, we have built a prototype system that will be illustrated in chapter 6.

Chapter 6

System Prototype Implementation

6.1. Chapter Objectives

This chapter presents a prototype implementation of our proposed context-aware system architecture, which utilizes context-level negotiations to establish Context-Level Agreements (CLAs). We have managed to implement all components and features of our architecture, with the exception of CLA-based policy derivation. This feature has been replaced with an equivalent implementation approach consisting of Java classes, conditions and threads.

The system implementation is based on a three-stage process. In the first stage, the Global Skeletal Ontology was developed using OWL [16]. The ontology, as presented in chapter 3, represented a general model through which context knowledge within all domains can be modeled directly or by extending the GSO. The second step involved installing and programming a set of Crossbow [28] sensors that would be capable of collecting environmental context information, including temperatures, humidity levels, pressure levels, and so on. Sensor readings were delivered through a wireless connection to a base station, the latter being connected to a workstation onto which the context Provider was built.

During the third stage, both Client and Provider were designed using the Java programming language. We implemented a simple inference engine that converted low-level sensor readings (e.g. temperature in degrees) into coarse context (e.g. cold, hot, warm, etc). An SQL database was used as a context repository for historical context. The sensors' latest readings, which were retrieved from the repository, were updated into an instantiated OWL document used as the GCO.

Our negotiation protocol was built on top of the TCP/IP protocols, and a sample Client was designed as a dynamic graphical user interface that adapted its content according to exchanged Views. The Client's Profiles presented the Provider with information about the Clients' identities, and through this information a View selection was made, providing Clients with rooms and context information to which they were permitted access. The

Client's GUI graphically presented context-level negotiations to users taking them through all necessary steps for a CLA to be established.

The Provider used established CLAs to dynamically build and compile Java-based classes for each established CLA and all context requests. The Provider triggered a thread that would monitor the CLA activation condition from context information that is continuously updated in the GCO. Once a CLA was activated, threads to monitor each context request's activation conditions were triggered. This in turn retrieved the necessary context information from the GCO as the conditions were triggered. All retrieved context was directed to ConCoord for delivery to the respective Clients.

In the next sections we present in detail the process of implementing each layer in our architecture.

6.2. Sensing and Context Acquisition

Numerous platforms exist for the creation of wireless sensor networks. MoteWorks [29] offers an end-to-end platform with optimized processing hardware, mesh networking software, and client monitoring and management tools for wireless sensing solutions. MoteWorks' deployment is composed of three software tiers; the Mote tier, the Server tier, and the Client tier.

The Mote tier encompasses the XMesh software running on sensor nodes which form a mesh network. XMesh is a networking protocol developed by Crossbow for wireless networks supporting such features as multi-hop, ad-hoc, mesh networking. Sensor nodes running the XMesh software compose an XMesh network. These nodes continuously hop radio messages to a base station from which they are passed on to a PC and thus extend the radio communication range while reducing power consumption. XMesh's message hopping also increases radio coverage and improves reliability. Through one or more in-between nodes, messages are delivered from the source node to its base station destination. XMesh also provides a networking service that is self-organizing and self-healing.

Motes within XMesh can be configured into one of three power modes: HP (high power), LP (low power), and ELP (extended low power). In addition, Quality of Service of the motes is

provided by either best effort or guaranteed delivery. The former uses link-level acknowledgement, while the later used end-to-end acknowledgement.

The Server tier (XServe) provides services to route, parse, transform and process data. XServe also serves as a gateway between wireless mesh networks and applications interacting with the mesh.

The third tier, the MoteView Client tier, is the client's user interface that displays information from the network to end users. MoteView displays the entire network or individual nodes for analysis through graphical charts or textual formats, historical views to network status, and sensor readings. MoteView also permits automatic emails if links between sensors are rerouted due to environmental changes or if user-defined conditions are met (for instance if sensor readings exceed some thresholds). The Client tier also allows users to reconfigure the network layout, to interactively analyze gathered information, and to remotely configure Motes in the wireless network.

MoteWorks is modeled on TinyOS [30], a component-based, event-driven and open-source operating system. TinyOS is capable of supporting microprocessors ranging from 8-bit architectures to 32-bit processors, providing a set of APIs to access the capabilities of sensor nodes so data can be preprocessed on the node prior to delivery to the base station. The component library provided with TinyOS includes network protocols, distributed services, sensor drivers, and data acquisition tools. TinyOS does not have a file system, supports only static memory allocation, and is a programming framework for embedded systems. As such, TinyOS permits building application-specific operating systems into applications.

Software components in TinyOS are organized into layers. The lower the layer, the closer it is to the hardware, whiles the higher the layer, the closer it is to the application. Each component in TinyOS has three computational concepts: commands, events, and tasks. Commands are requests made to a component to perform a service, and events signal the completion of a service. Meanwhile, tasks are functions executed by the TinyOS scheduler at a later time, such that by posting tasks, commands and events may return immediately after execution instead of waiting for a reply and letting the task complete the necessary computations separately.

The MoteWorks development environment supports a variety of programming tools, and the one we have used is the MIB520CA USB port programming board. The MIB520 uses the USB port as a virtual COM port through the FTDI FT2232C VCP drivers. MIB520 has two virtual serial ports. The first, com_x, is used for Mote programming, and the second, com_{x+1}, is used for Mote communication. The following general command is used to program Motes using a programming board:

make *<platform>* ***re|install,*** *<n> <programmer>, <port>*

Here, *<platform>* refers to the type of processor and radio hardware used for the Mote being programmed, *<n>* is an optional number used to the node ID or address, *<programmer>* is the type of board used in programming the Mote, and *<port>* is the port address or number of the host PC to which the programming board is attached. Node IDs are used to distinguish between sensor readings received by the base station according to the reading's source node. Therefore, if a MICAz Mote with an ID of '4' is to be programmed from a MIB 520 connected to a PC's serial port COM5, the following command is used:

make *micaz* ***install,*** *4 mib520, com5*

The use of *install* compiles the application for the target Mote, sets the node's ID, and programs the sensor. Using *reinstall* on the hand sets the node's ID and downloads the previously compiled program into the mote without a recompilation.

In the next section, we present the overall procedure of designing and organizing our sensing layer using the XMesh multi-hop protocol.

6.2.1. Sensing Layer Configuration

Our testing environment was distributed over four rooms on the 5th floor of the Colonel By building at the University of

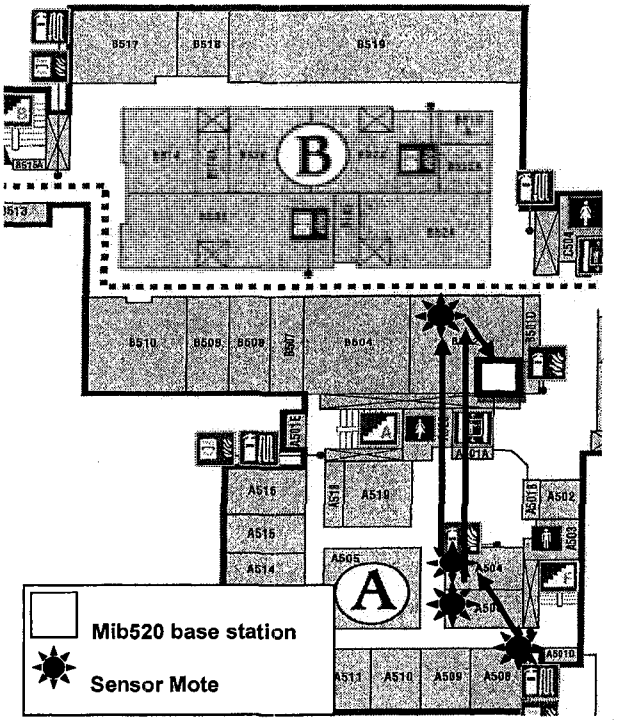


Figure 6-1. Map of prototype testing domain

Ottawa. These rooms were A508, A506, A504, and B502. In each room was placed a single Mote with a standard edition of MICAz (MPR2400) / MICA2 (MPR 4x0). Each Mote was supplied with an MTS400 basic environmental sensor board capable of sensing the ambient light levels, the acceleration in the x and y axis, the barometric pressure and temperature levels, and the relative humidity and temperature levels of its surroundings. A picture of a sample mote and the MTS400 sensing board is provided in figure 6-2.

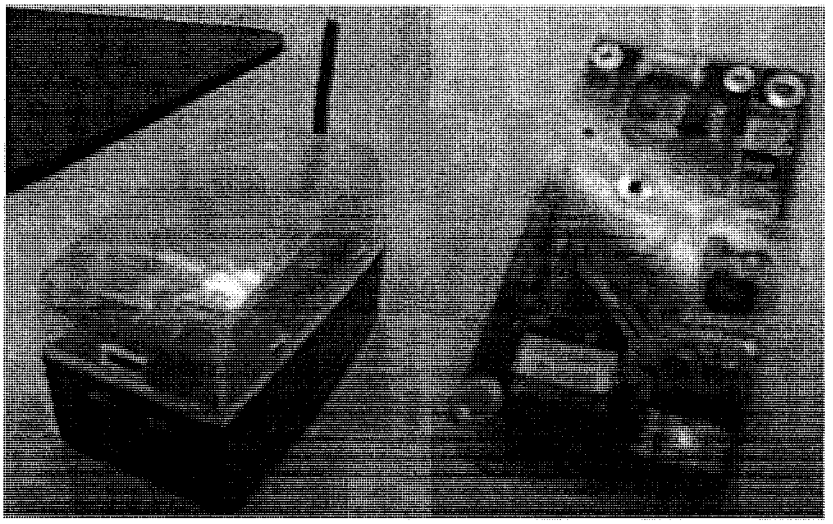


Figure 6-2. Wireless Mote and sensing board

A single gateway MIB520 programming board with a MICAz/MICA2 processor and radio module was connected via USB port to a Window's XP PC with 2.00 GB of RAM, 3.6GHz Intel Pentium D processor, and 232 GB of total disk space onto which MoteWorks was installed.

Each Mote was programmed to sequentially sample all environmental sensors provided on its MICAz board. The readings were then packetized and the data was sent back to the base station through the XMesh multi-hop networking service. To program each Mote, the first step is to type in the *Makefile*, as shown below:

```
Include     Makefile.component
include     ../../MakeXbowlocal
GOALS       += binlink
include     $(MAKERULES)
```

Adding the 'include' lines causes the compiler to add the indicated files. The MakeXbowlocal file allows users to change the local group ID, the RX/TX frequency, and

the RF transmission power. Makefile is therefore a file containing the dependencies the application uses during the compilation step.

The next step is to create the *Makefile.component* file. This file describes the top-level application component, SENSING400 in our case, and the name of the sensor board being used (MTS400). Indicating the name of the sensor board informs the compiler to use the proper pre-built nesC components (drivers) in order to access the sensor devices on that board. The *Makefile.component* file used in our case is provided below:

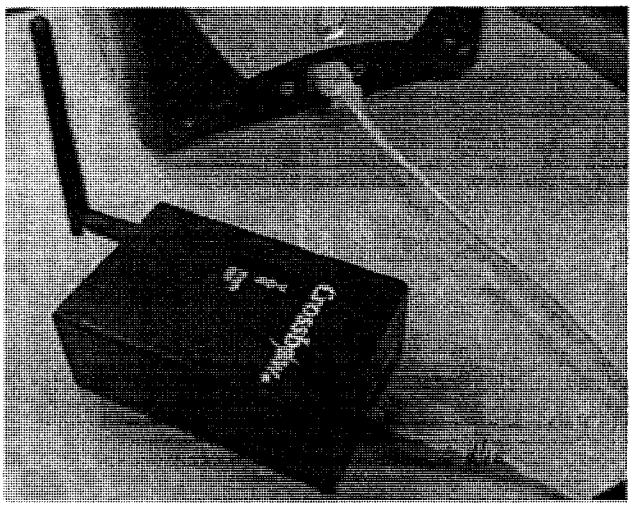


Figure 6-3. MIB 520 base station

```
COMPONENT     =XMTS400
SENSORBOARD   =mts400
```

Once both *Makefile* and *Makefile.component* files have been created, the next step is to create the top-level configuration file. The configuration (SENSING400.nc) file includes references to such interfaces as the StdControl interface, which provides TinyOS applications with the basic functionalities to initialize, start and stop. The configuration file also includes information about wiring board devices to functions. Wiring connects providers and users together; prior to wiring, a component is completely decoupled from the components that it calls. For one module to be able to call another module, a set of names in one component (interface) must be mapped to a set of names in another component. Our configuration file wired devices to functions for all light sensors, accelerometers, humidity and temperature sensors, as well as Timer and LED devices. The sample code displayed below partially shows some of the wiring done in our SENSING400.nc file for humidity, temperature, and pressure sensors:

```
Main.StdControl                 →XMTS400M;
Main.StdControl                 →QueuedSend.StdControl;
Main.StdControl                 →MULTIHOPROUTER.StdControl;
Main.StdControl                 →Comm;
Main.StdControl                 →TimerC;
XMTS400M.TempHumControl         →  SensirionHumidity;
XMTS400M.Humidity               → SensirionHumidity.Humidity;
XMTS400M.Temperature            → SensirionHumidity.Temperature;
XMTS400M.HumidityError          → SensirionHumidity.HumidityError;
XMTS400M.TemperatureError       → SensirionHumidity.TemperatureError;
XMTS400M.IntersemaCal           → IntersemaPressure;
XMTS400M.PressureControl        → IntersemaPressure;
XMTS400M.IntersemaPressure      → IntersemaPressure.Pressure;
XMTS400M.IntersemaTemp          → IntersemaPressure.Temperature;
```

The MULTIHOPROUTER seen in the above code component implements the XMesh protocol that can exchange messages between Motes until they arrive at the base station. Messages are sent via the MhopSend interface, which adds more parameters to the delivered messages. This specifies the XMesh communication transport mode that delivers messages upstream in the direction of the base station.

Executable files are generated by a nesC compiler when both a configuration file and the source file are present. Our source file, the SENSING400M.nc, provides the functionality of the SENSING400 application that is located the application's module. The SENSING400.nc configuration file is used to wire the SENSING400M.nce module to other components that are required by the SENSING400 application. This SENSING400M.nc file contains the application's programming code used to read sensor data, start the Timer and toggle the LEDs located on each device.

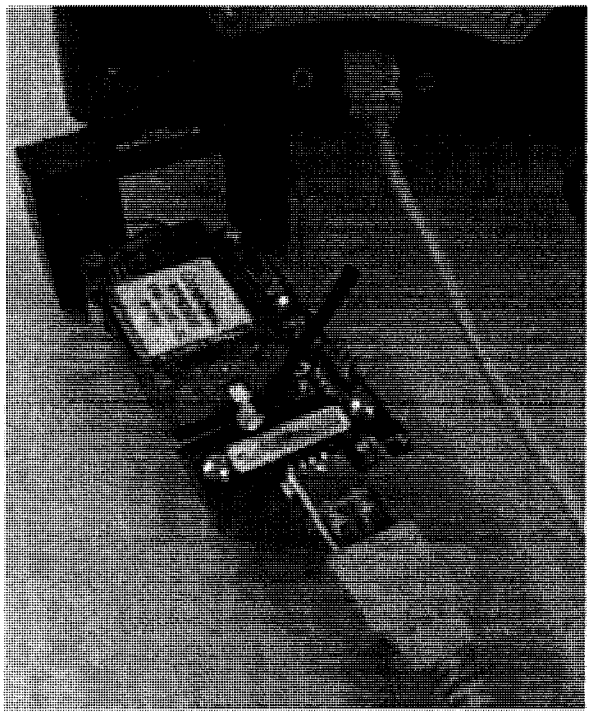


Figure 6-4. Programming board

To program the sensing devices, the Mote attached to the sensor board is connected to a

programming board that is in turn connected to the PC via a USB port (see figure 6-4). The SENSING400 program was compiled and the program was uploaded to each Mote with a unique ID for each Mote. Each Mote was programmed to read its sensor data periodically in 1-second intervals. The read data was then passed on from one sensor to another using XMesh until the data arrived at the base station.

For the base station to be able to receive the messages arriving from the distributed sensor Motes, it had to be programmed with a special application named XMeshBase, provided with the CrossBow installation kit. The XMeshBase application program acted as a sniffer and allowed the base station to listen to, and interpret, all wireless sensor messages arriving from the distributed Motes. The base station connected to the PC is displayed in figure 6-3.

To verify the received messages and the functionality of the wireless sensors and base station, we ran the XServe application on the PC to which the base station was connected. XServe is a program that runs within a Cygwin command prompt window, which displays the sensor message packets arriving over the serial port to which the base station is connected. Figure 6-5 provides a snapshot of messages received from Mote with a node ID = 04. The reading shows remaining battery life, as well as humidity, temperature, pressure, light and acceleration levels.

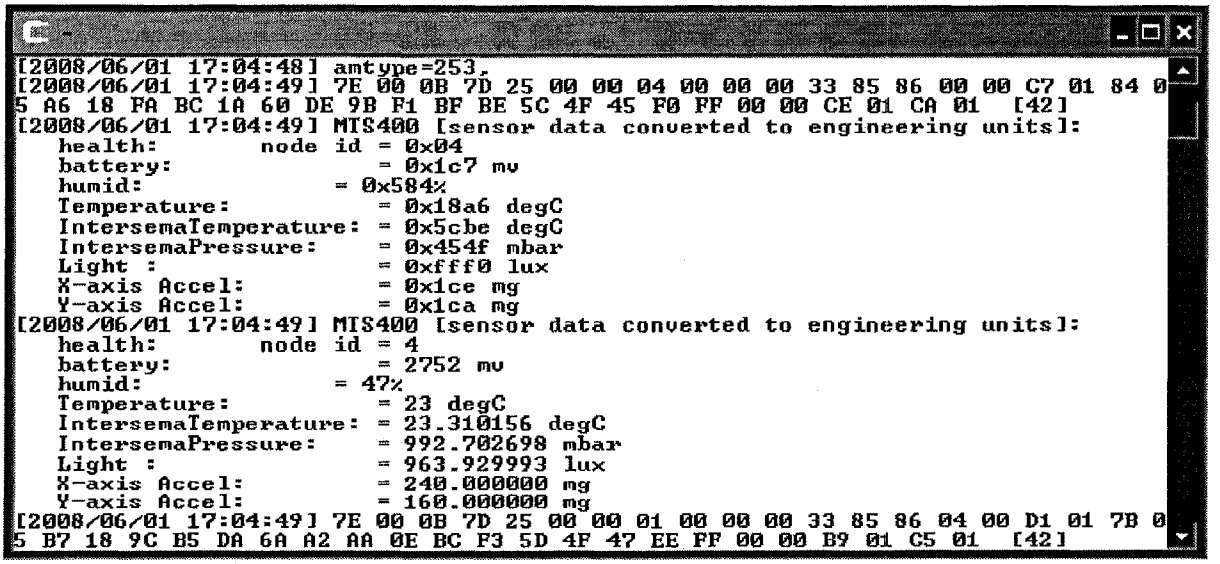


Figure 6-5. Received sensor readings

In our experiment we have replaced the XServe application with a Java-based program that imitates the role of the Context Acquisition Listener unit responsible for acquiring the context information arriving through the serial port from the base station. The details concerning the context Provider's implementation will be further explored in the next section.

6.3. Context Provider Prototype

The majority of the context provider's components in the architecture presented in chapter 4 have been implemented here. We will here present a bottom up description of the implementation process, showing the functionality at each stage. Since the MIB520 base station in section 6.2 was directly connected to a PC through a USB serial port, packets arriving through that port needed to be interpreted in a format that could be recognized by the context Provider. The Context Acquisition Listener unit is responsible for listening to any information arriving through the USB serial port. As we are currently dealing with only one type of sensor, though, the Context Acquisition Listener unit was only responsible for interpreting one type of message packets and we only needed a single Context Acquisition Listener unit to meet the requirements. Adding further context sources may have required modifying the Listener unit so it could listen for the new packet formats or build multiple Context Acquisition Listeners. Building Context Acquisition Listeners would allow the Provider to interpret all incoming sensor message formats. For instance, Figure 6-6 provides a sample of the code used in the Context Acquisition Listener to interpret the humidity level within the packet arriving from the base station.

```
//Humidity
int hum2 = (readBuffer[23] & 0xff)<<8;
int hum1 = (readBuffer[22] & 0xff);
int hum=hum1+hum2;
int tem2 = (readBuffer[25] & 0xff)<<8;
int tem1 = (readBuffer[24] & 0xff);
int TempData2=tem1+tem2;
float fTemp=(float)(-38.4+0.0098*(float)TempData2);
float fHumidity=(float)(-4.0+0.0405*hum-0.0000028*hum*hum);
fHumidity=(float)((fTemp-25.0)*(0.01+0.00008*hum)+fHumidity);
```

Figure 6-6. Partial decoding method of sensor readings.

Context information retrieved by the Listener is either sent for storage within the Context Repository, or (if possible) to an inference engine to generate new high-level context

information from the low context. Information arriving and interpreted by the Context Acquisition Listener is handled by a *Context Storage and Retrieval* unit that sends and retrieves information from the Context Repository and Inference Engine.

We have provided a mySQL-based database into which all acquired and inferred context information is saved. The database provides an ontology-independent storage for the context Provider's knowledge. External entities, like context Clients, will therefore not be familiar with the table structures used by the context Provider for its Context Repository. The Context Repository serves two purposes. First, historical information that is usually not saved within the Global Concrete Ontology due to limitations in GCO's size, and the fact that only the latest context values are stored within the GCO, can be accessed by the Provider from the Repository and then served to Clients. Secondly, system failures and unexpected loss of information in the GCO may be overcome by retrieving the latest context information from the Context Repository. Figure 6-7 presents an instantaneous snapshot of our Context Repository, which stores all interpreted sensor readings acquired from the Context Acquisition Listener module.

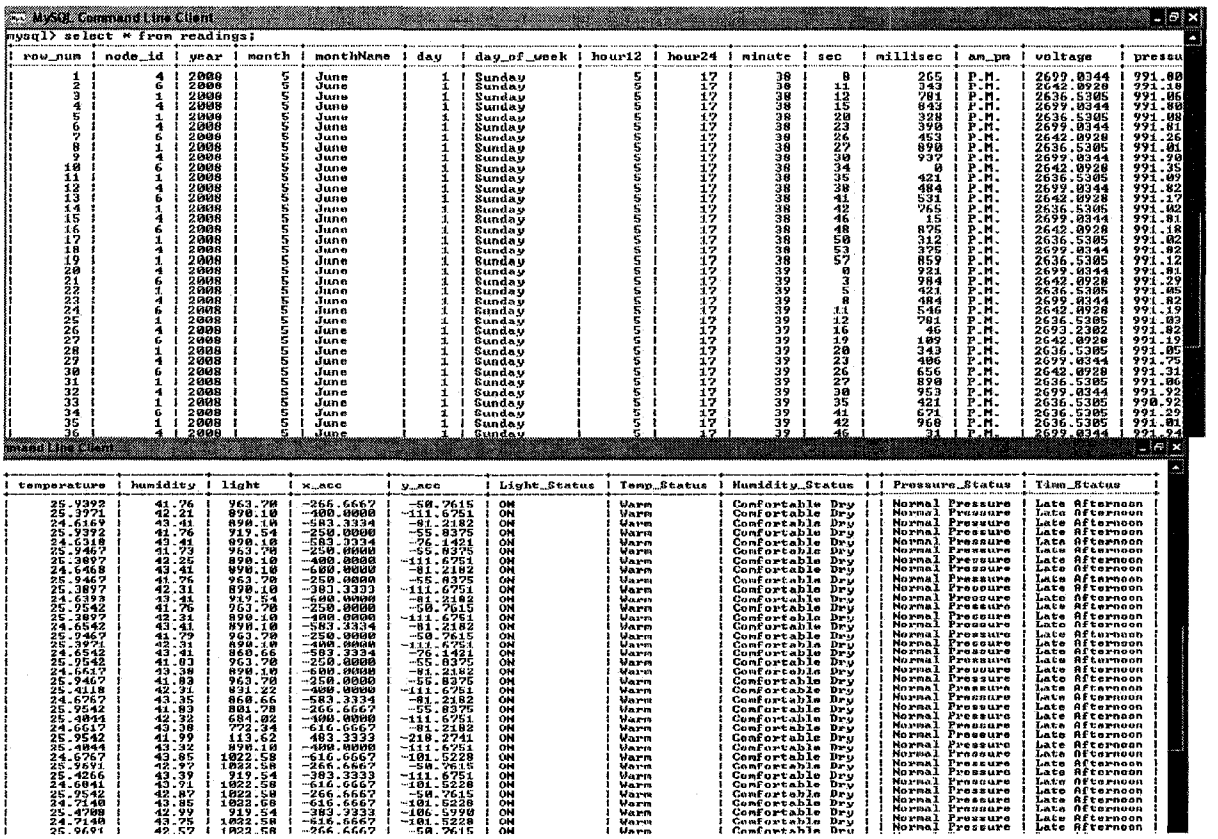


Figure 6-7. Sensor context values stored within repository

The Context Repository also stored inferred context information generated by our inference engine. The current Java-based implementation of our inference engine is based on our first-order logic, which converts the acquired raw sensor data (temperature, humidity, and pressure levels, for instance) into more abstract and high-level contexts. For example, IF(temperature levels < 10°C) are given a status of "Cold", while those IF(temperature level > =10°C and <15°C) are considered "Cool", and the same principal is followed for "Warm", "Hot", or "Very Cold" temperature statuses. The same approach has been applied to represent humidity levels ranging from "Very Dry" to "Very Humid".

Similarly, light levels were given a high-level representation. Since our sensor Motes were located indoors, only two states of light status were necessary: ON and OFF. Tests showed that sensor readings below 10 watts represented rooms with the lights turned off, while readings above 10 watts were seen for rooms with lights turned on. Pressure readings were also converted to low, normal, and high pressures. Given the simplicity of the rule engine used for inferring context, we have begun the process of expanding our inference engine to include more powerful rule engines, such as Jess [31], which uses an enhanced version of the Rete algorithm to process rules.

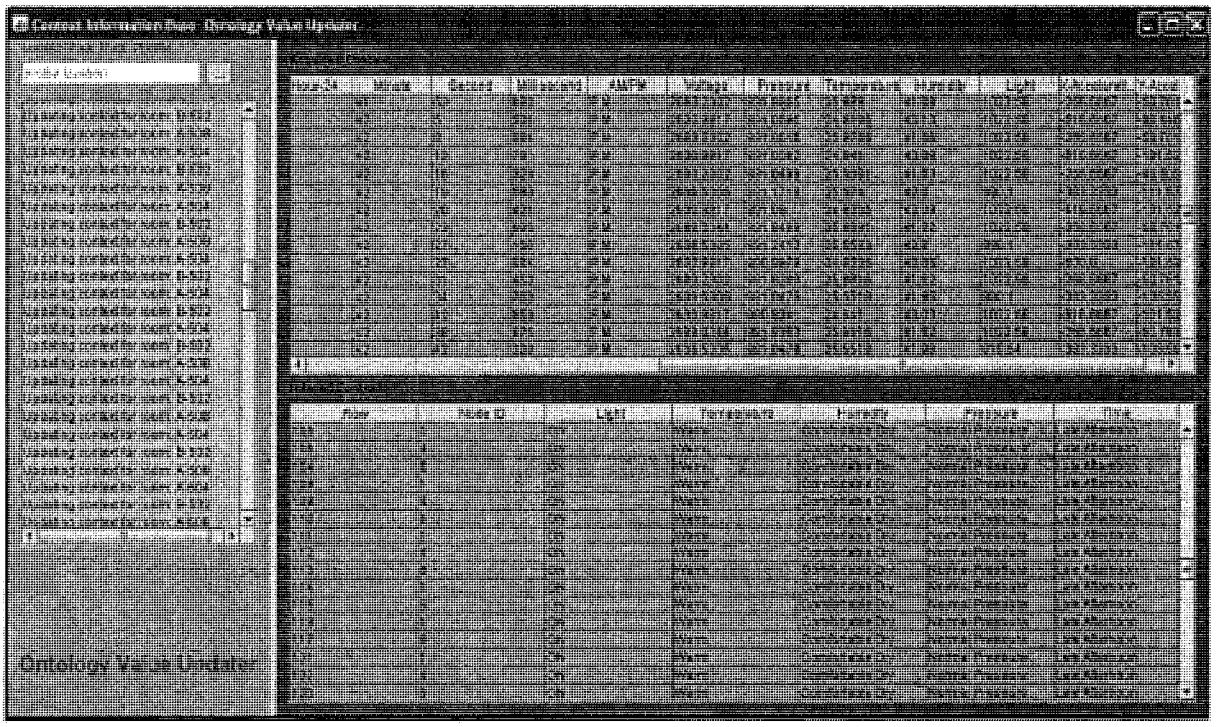


Figure 6-8. Values retrieved from context repository by Ontology Value Updater.

Each sensor reading was time-stamped to indicate the day, month, year, hour, minute, second and millisecond the sensor reading arrived at the *Ontology Value Updater* unit. The sensor readings were also tagged with the ID of the source Mote that delivered the sensor readings to the base station. We have provided a graphical user interface to illustrate the Ontology Value Updater unit. This GUI emphasizes two aspects of the Ontology Value Updater's functionality: its ability to store context data arriving from the Context Acquisition Listener unit into the Context Repository, and its ability to send that data to the Inference Engine to generate higher level context information. This is shown in the two right tables ("Acquired Context" and "Inferred Context") of figure 6-8.

Raw and inferred context information is not only stored within the Context Repository. The latest context information available within the repository is used by the Context Value Updater to update the Global Concrete Ontology. To perform this update, the Context Storage and Retrieval unit class instantiates the Ontology Value Updater class and utilizes a set of APIs we have provided. These APIs abstract the process of accessing the OWL-based GCO from the Storage and Retrieval unit, continuously calling methods provided by the Ontology Value Updater class to update specific context values, such as temperature levels or pressure statuses.

Importantly, the Global Concrete Ontology - OWL document - is concurrently accessed by three components within the context Provider. These components are:

- Ontology Value Updater: utilizes the latest context information stored within the Context Repository to update the GCO.
- CIC Access Point: retrieves needed context information from the GCO for delivery to the ConCoord Access Point and ultimately to the Clients that have requested the context information.
- Policy Decision Point (PDP): the policy decision point that constantly monitors and listens for changes in context information occurring within the GCO. These changes may trigger the activation/deactivation of the Clients' CLAs, and the activation/deactivation of context information delivery to the Clients.

The existence of multiple units trying to access the OWL-based GCO document means that any attempt to read or write to the file from any entity would block access to the other entities. This access blockage results in the existence of an unbalanced race between the Ontology Value Updater, the CIC Access Point, and the PDP to access the GCO.

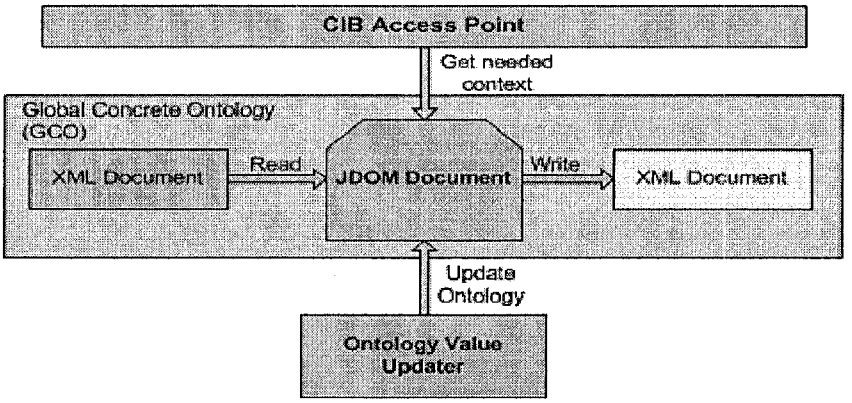


Figure 6-9. Parallel GCO access through JDOM.

Consequently, we have used the JDOM [32] Java-based API, which is based on the Jakarta Ant originally developed for the Jakarta Tomcat project. The latter manipulated XML files through noncomplex, low-cost, and lightweight means to reading and writing XML data without having to follow memory-consumptive options.

JDOM operates by reading the OWL-based GCO and creating a java.util.List of the complete document. This list is called a JDOM document. During the course of the Provider's life, entities interested in accessing the ontology simply access the JDOM document to update and read the current instances of context information. Any updates to the ontology are done to the List without needing to read or write to any file. Updating the actual physical Concrete Ontology represented by the OWL-based file could be done periodically at specified intervals, or whenever the context provider is about to be shutdown. This reduces the delay the three entities may have faced if the GCO was accessed directly each time. A key advantage of using the periodic write method is having a restore point in case of a system failure during the system's runtime

Because each Mote was placed in a different room, the Ontology Value Updater provided methods for updating specific context information for specific rooms within the GCO. One method included the *setRoomTemperature(String roomID,int tempValue)* method, which

updated the room temperature value of any room given the correct room ID. Figure 6-10 provides the sample code for the *setRoomTemperature* method, illustrating the use of JDOM's methods to accessing the GCO-derived Lists to update the room's temperature values.

Since our prototype context-aware system was designed for use within a university domain, our Global Concrete Ontology utilized only a subset of the Global Skeletal Ontology. The limited amount of context information gained from the wireless Crossbow sensors meant that our Global Concrete Ontology required only a partial representation of the Location Ontology that was concerned with representing rooms, floors and buildings, some of the relationships between them, and attributes like temperature levels, humidity, light, and so on. Other information contained in our GCO referred to the users within the University domain, such as professors and different types of student classes. The GCO also contained information about Quality of Context levels and Time representation.

Clients were differentiated according to their identities and roles. Identities in this case referred to names, dates of birth, and student numbers. Roles referred to a given Client's position within the university's role hierarchy (for instance, the hierarchy composed of professors, PhD students, Master's students, undergraduate students and so on). The initial set of Client Views present within the context Provider were designed in accordance with these guidelines, supplying Clients with a list of context information they were permitted to (for instance, as a list of the rooms whose context information the Clients would be allowed to access). Other ontology classes, properties, and individuals were also included, providing Clients with a complete description of their accessible context information.

All context Views were stored within a directory located on the same PC station the context Provider was running from, and this directory represented the *Views Repository* provided within our architecture. During negotiations, the unit responsible for performing context-level negotiations with Clients (ConCoord Access Point) retrieved the proper Views for delivery in accordance with their identities and roles. The ConCoord Access Point also used the Views within the Repository, along with context information available within the GCO through the CIC Access Point to determine whether the Client's context requests were feasible.

```
public static void setRoomTemperature(String roomID,String tempValue)
   {// <editor-fold defaultstate="collapsed" desc=" Generated Code ">
      String room=roomID;
      String temperature=tempValue;
      List roomChildren=root.getChildren("Room",owl);
      for (int i=0; i<roomChildren.size(); i++){//search all existing rooms for the correct one.
         Element currentRoom=(Element)roomChildren.get(i);
         String RoomID=currentRoom.getAttribute("ID",rdf).getValue();
         if(RoomID.equals(room)){//room with correct ID has been found.
            Element roomTemp=currentRoom.getChild("hasTemperature",owl);
            String tempID=roomTemp.getAttributeValue("resource", rdf);
            List temperatureElementList=root.getChildren("Temperature", owl);
            for(int j=0; j<temperatureElementList.size(); j++){
               Element temperatureElement=(Element)temperatureElementList.get(j);

if(temperatureElement.getAttributeValue("ID",rdf).toString().equalsIgnoreCase(tempID.substring(1))){
//Update the rooms temperature.
                  temperatureElement.getChild("hasTemperatureLevel",owl).setText(temperature);
               }
            }
            break;
         }
      }//</editor-fold>
   }
```

Figure 6-10. Method using JDOM to update room temperature within GCO

6.4. Client Prototype

Communication between Clients and Providers was based on a client-server channel that exchanged context negotiation messages over the TCP/IP protocols. Clients were represented by a two-tier implementation. At the top was a graphical user interface that presented human users with a simple and intuitive method for performing context-level negotiations with Providers. At this level, users were provided with the information they needed to successfully perform all the negotiation steps; connection to the Provider, negotiation of the CLA Validity Condition, negotiation of multiple context requests with each having its own Applicability Conditions and Quality of Context criteria, and the View Update methods. Information displayed to the user within the user's graphical interface is retrieved from the Client View the user had received. This information thus populates components such as combo boxes, dropdown menus, and similar components with context information such as the Clients' accessible rooms (ontology individuals) and their respective context information (ontology properties).

The Message Interpreter (MI) was located at the lower level. The MI was responsible for translating all high-level actions the user made through the high level graphical interface. The result would be low-level messages such as those in the context-level negotiation protocol we presented in chapter 5. All converted messages were packetized and delivered to the context Provider over the network. The MI was also responsible for translating and displaying messages arriving from the context Provider into a higher-level format that could be understood by human users. These messages could be replies to offers made by the Client or context messages carrying context information delivered in accordance to established CLAs.

The following is a step-by-step analysis of a hypothetical context-level negotiation between a Client and a context Provider, where a Context-Level Agreement (CLA) is established through our proposed negotiation protocol.

6.5. Negotiation Walkthrough

To illustrate the functionalities of our prototype system and to prove the concept of our context-level negotiation protocol, this section describes a sample negotiation session between a context provider and a context Client. This negotiation establishes a CLA through which the Client will be supplied with a set of context information acquired by the context Provider through the environmental sensors described in section 6.2.

The first step involves triggering the context Provider. Two graphical windows are displayed: the first, presented earlier in figure 6-8, belongs to the Ontology Value Updater unit. This unit is responsible for retrieving the latest raw and inferred context information from the Context Repository in order to update the Global Concrete Ontology. The second window belongs to the ConCoord Access Point (figure 6-11). This window displays all messages arriving from the Client and sent by the Provider. This negotiation monitoring window is usually not needed in reality, but we have included it here to monitor the current state of the context Provider.

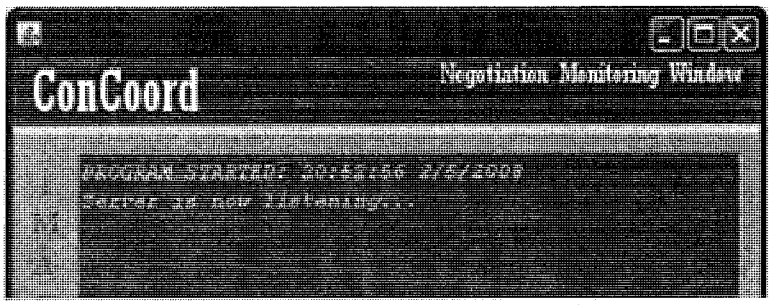


Figure 6-11. ConCoord negotiation monitoring window

Initially, the context Provider remains in the listening state waiting for Clients to begin their negotiations.

We have built our Java-based Client application on a separate computer that is connected to the network. The Client is not connected to the base station that supplied the context Provider with raw sensor readings. Thus, for the Client to acquire its needed context information, it must establish its CLA by engaging in context-level negotiations with the context Provider. The initialized Client application gives the human user on the Client's side a graphical interface that guides him/her through the steps needed to establish an agreement.

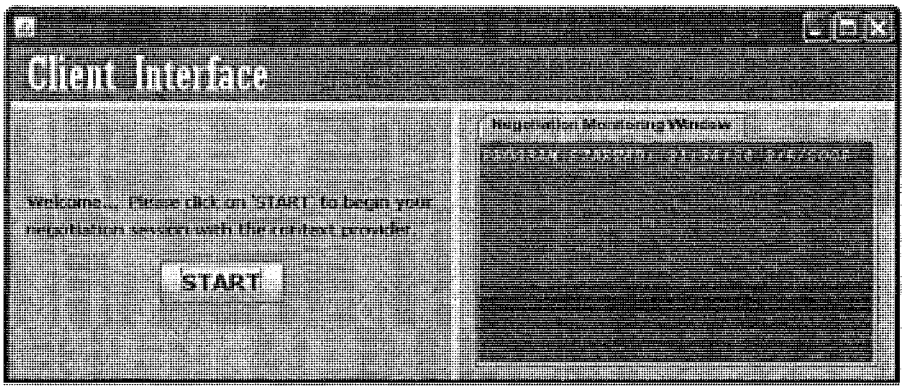


Figure 6-12. Client negotiation monitoring window

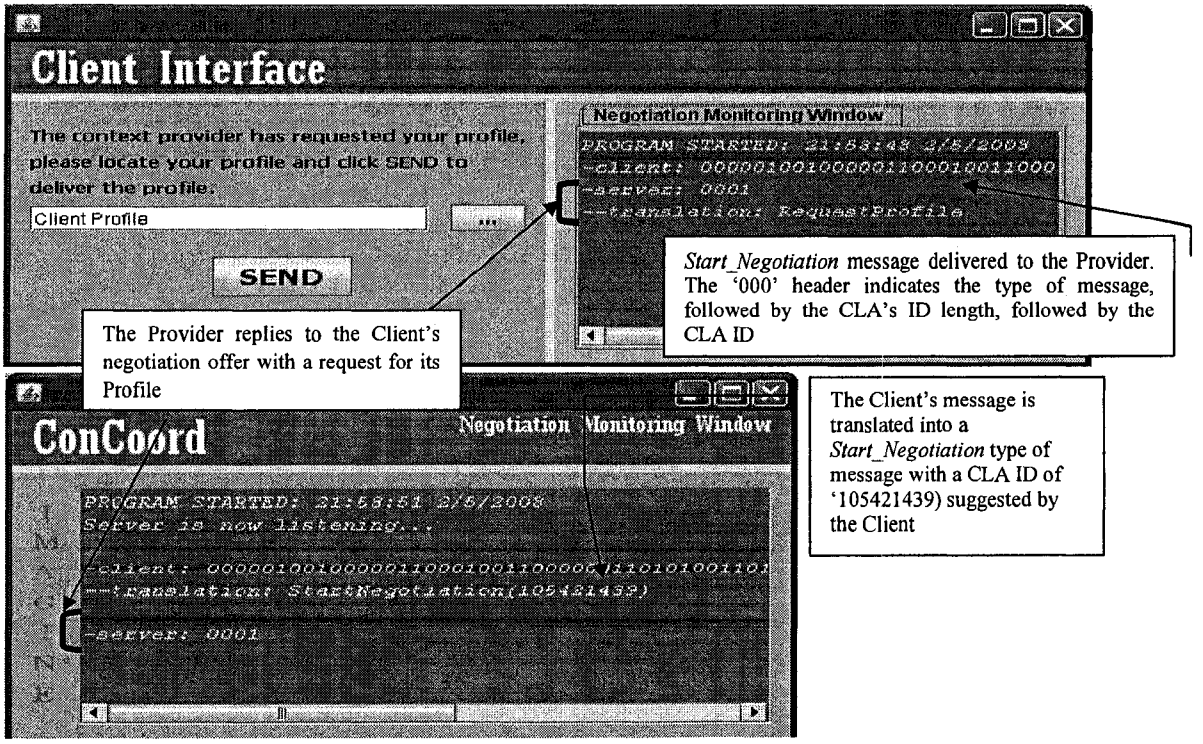
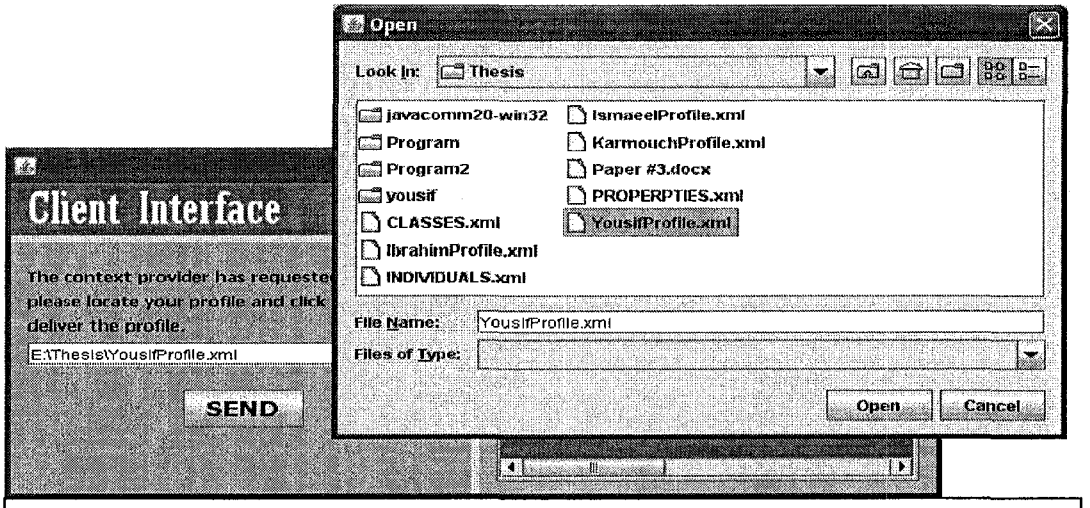


Figure 6-13. Initializing negotiations

To begin the negotiation session, the user must click the 'START' button. This abstract high-level action is in turn translated by the MI into the first message required in the context-level negotiation protocol discussed in chapter 5: *Start_Negotiations(CLA_ ID)*. Both Client and Provider use the provided CLA ID for future modification to the CLA. Assuming the Provider's acceptance of the negotiation session, the Provider replies with a *Request_Profile* message requiring the Client to supply its Profile to the Provider. These steps are displayed in figure 6-13. Worth noting is the change in the Client's user interface, as the Client now needs to search for the correct Profile for it to deliver to the Provider.




```
<Profile rdf:ID="Yousif_Profile">
      <hasProfileID rdf:datatype="&xsd;string">YProfile1</hasProfileID>
      <has_Profile_Owner rdf:resource="#Yousif"/>
</Profile>
<MastersStudent rdf:ID="Yousif">
      <hasMiddleName rdf:datatype="&xsd;string">Kadhim</hasMiddleName>
      <hasFirstName rdf:datatype="&xsd;string">Yousif</hasFirstName>
      <isStudentWithinUniversity rdf:datatype="&xsd;string">University of
Ottawa</isStudentWithinUniversity>
      <hasStudentNumber rdf:datatype="&xsd;int">2984293</hasStudentNumber>
      <hasSupervisor rdf:datatype="&xsd;string">Dr. Ahmed Karmouch</hasSupervisor>
      <hasLastName rdf:datatype="&xsd;string">Al Ridhawi</hasLastName>
      <hasTitle rdf:datatype="&xsd;string">Mr.</hasTitle>
      <ownerOfRoom rdf:datatype="&xsd;string">A-504</ownerOfRoom>
      <yearOfStudy rdf:datatype="&xsd;int">2</yearOfStudy>
      <isWithinFaculty rdf:datatype="&xsd;string">Graduate and Post
Graduate</isWithinFaculty>
      <isMemberWithinGroup rdf:datatype="&xsd;string">IMAGINE</isMemberWithinGroup>
</MastersStudent>
```

Figure 6-14. Searching for Client Profile and partial Profile representation

It is assumed that each Client is capable of generating its own personal Profile according to the concepts provided within the Global Skeletal Ontology. These Profiles contain information about the identity and abilities of Clients. Because of these Profiles, context Providers are able of making informed decisions as to which Views should be delivered to a given Client.

Figure 6-14 shows a Client searching for its Profile – the Client in this case being a user named 'Yousif'. A partial representation of some of the information provided with Yousif's Profile is displayed in figure 6-14. As shown below, the Profile's owner (*has_Profile_Owner*) including his full name, student number, supervisor, and office are revealed.

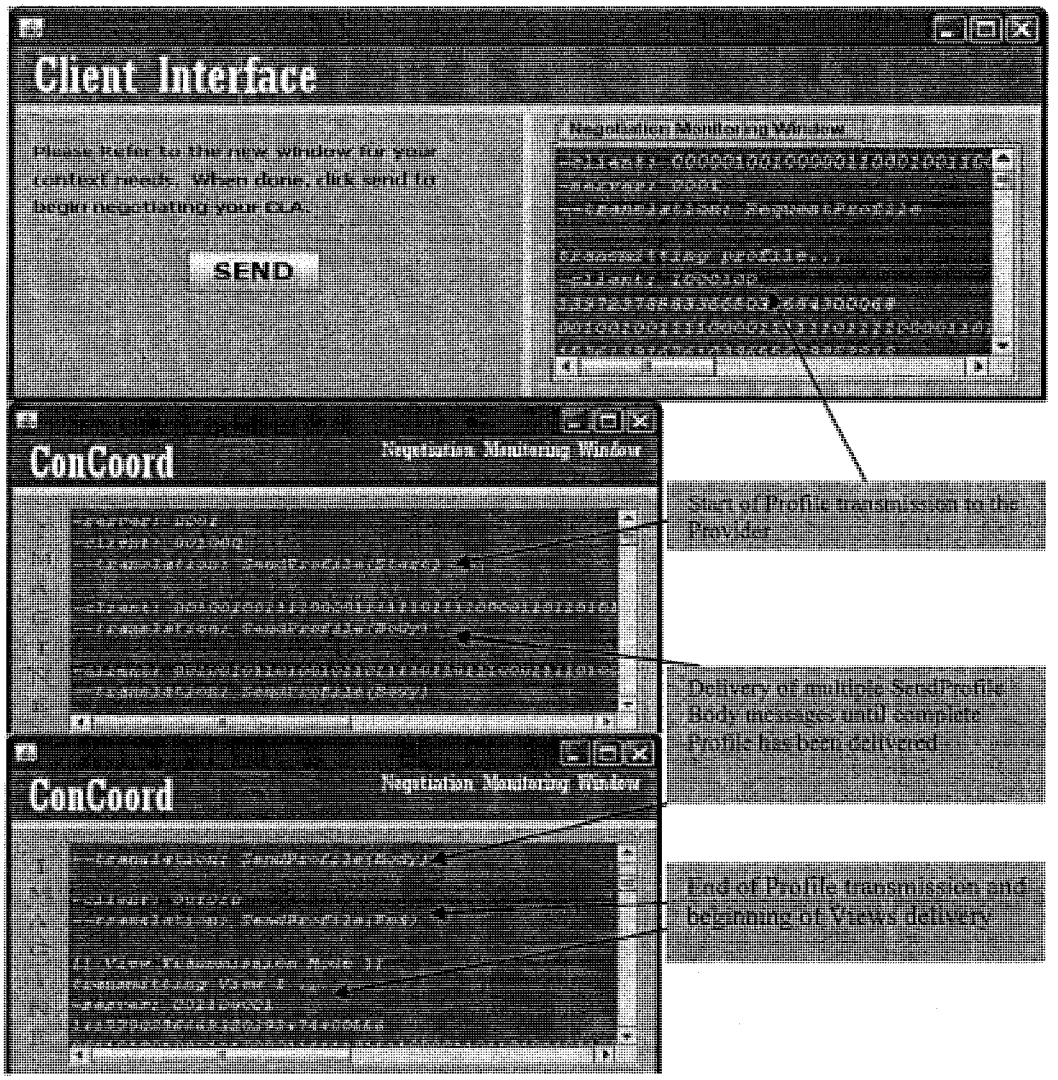


Figure 6-15. Exchange of Profiles

As explained in chapter 5, delivery of the Client's Profile occurs in three stages. In the first stage, the Client informs the Provider that Profile delivery is under way with its START message. The second stage involves sending multiple BODY messages, which carry sections of the Client's Profile until the complete Profile document has been successfully delivered. Finally, the Client sends its END message, signaling to the context Provider that the complete Profile should have been received at this point.

Figure 6-15 illustrates these three types of messages where the Provider continues to accept *SendProfile* messages until the END message is received.

Once the complete Profile is available to the Provider the search for the fittest Views for the Client begins. In the above scenario, the Provider can determine which Views must be delivered to Client Yousif by using the identity information present within the Profile. Yousif's Views are composed of three documents. The first document, the Root View, contains the general list of context information accessible to the Client, where the Time Ontology and Location Ontology are concerned. From the Location ontology, a list of accessible individuals from the Room class is provided along with all context properties and relationships accessible for those Rooms. The next two documents contain additional context properties accessible for rooms listed in the Root View but whose accessible context properties exceed those provided within the Root View document. Once all three View documents have been received by the Client, the user is informed of the situation; generation of the Client's negotiation window thus commences (see figure 6-16).

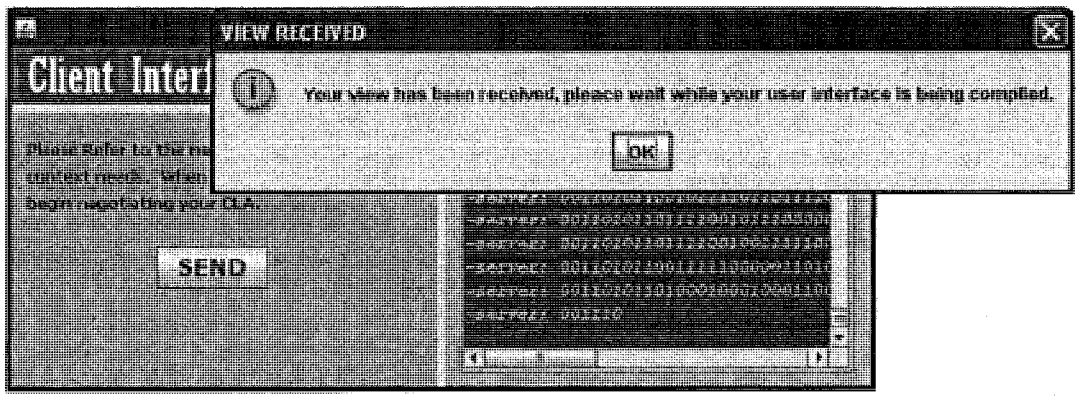


Figure 6-16. Complete of Profile and View exchange

The negotiation interface window is dynamically populated with accessible context information found within the Client's Views. Therefore all individuals (instances of the Room class) and their respective properties help to constitute an easy-to-use graphical interface for Clients. This allows the Client to perform its context-level negotiations by using tabs for the three main in context-level negotiation steps:

- CLA Validity: used to form all necessary negotiations regarding CLA Validity conditions; the latter could be Periodic, Logical Conditional, or Hybrid.
- Context Requests: used to perform negotiations related to Context Requests. The user (representing the Client) indicates the context information it is interested in receiving (notification, property type delivery, class type delivery), the conditions under which the context information should be delivered (Context Request Applicability conditions), and the quality of context levels to be met in order for the Provider to deliver the context information to the Client.
- View Update: used for negotiating the method by which changes to Views (if any) are reflected back to the Client.

The first step the Client must take is to choose the appropriate method for CLA activation and deactivation. As discussed in chapter 5, the Periodic CLA Validity condition (figure 5-4) required a clear description of the period's start time and end time. The Client was provided with several lists and combo boxes from which to choose the date's day, month and year, as well as the time's hour, minute, and second. All choices made by the Client (figure 6-17A) are translated by the MI to their low-level formats, as explained in chapter 5.

Meanwhile, the Logical Conditional option must give the Client the flexibility to express logical conditions using both of the two groups of conditions: WHILE and START/END TRIGGER. For both groups, the Client should be able to freely express its required condition in terms of accessible classes, individuals, and properties. Since we have limited the WHILE groups window, the Client can express conditions composed of no more than two sub-conditions; for instance figure 6-17B shows a CLA Validity condition that must be activated while Room B-502 has a humidity status of 'Comfortable Dry' or when room A-504's humidity level is greater than, or equal to 85%.

This said, the example we provide in this section utilizes the Hybrid CLA Validity condition. As explained earlier, Hybrid conditions are divided into two groups: Periodic Conditional and Conditional Periodic. This is reflected in the options the user is provided within its interface. Choosing the Periodic conditional, as in the case in figure 6-17A, requires that the Client express the condition to be validated within a given time period in order for a CLA to be active. The example shows that the Client is interested in activating its CLA whenever the lights turn on in room B-502, at any time, between 09:29:09 A.M. on February 3rd, 2008 and 21:04:04 P.M. on August 4th, 2008.

Pressing the SAVE button trigger the MI to translate the user's entries into their respective bit-level representations as established in our context-level negotiation protocol, and to deliver this *offer* to the Provider.

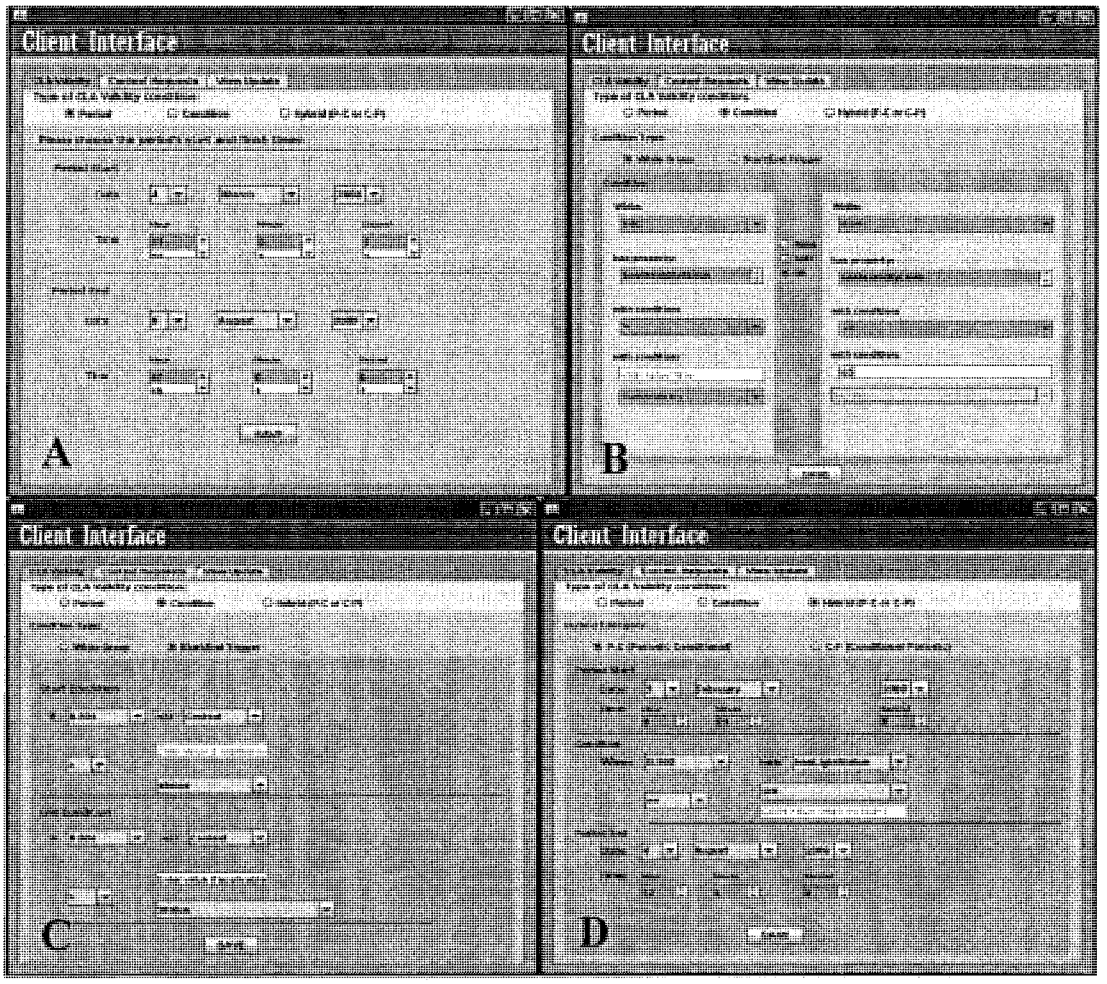


Figure 6-17. CLA Validity conditions window.

The Provider either accepts or rejects the offer based on its capabilities and on the context information permissible to the Client in light of its Views. As shown in figure 6-18, the Hybrid P-C was rejected by the Provider, and two messages were sent to the Client, a rejection message and a counter-offer message. The counter-offer message is decoded for the user by the MI, and displayed by an option window asking the Client to either accept or reject the counter-offer. If the Client accepts the Provider's offer, as in our example, an acceptance message is sent to the Provider. Otherwise, a rejection message is sent along with a new counter-offer from the Client. The process is repeated until the Client and Provider reach an agreement over the CLA Validity conditions that must be met in order for the CLA to be activated and deactivated.

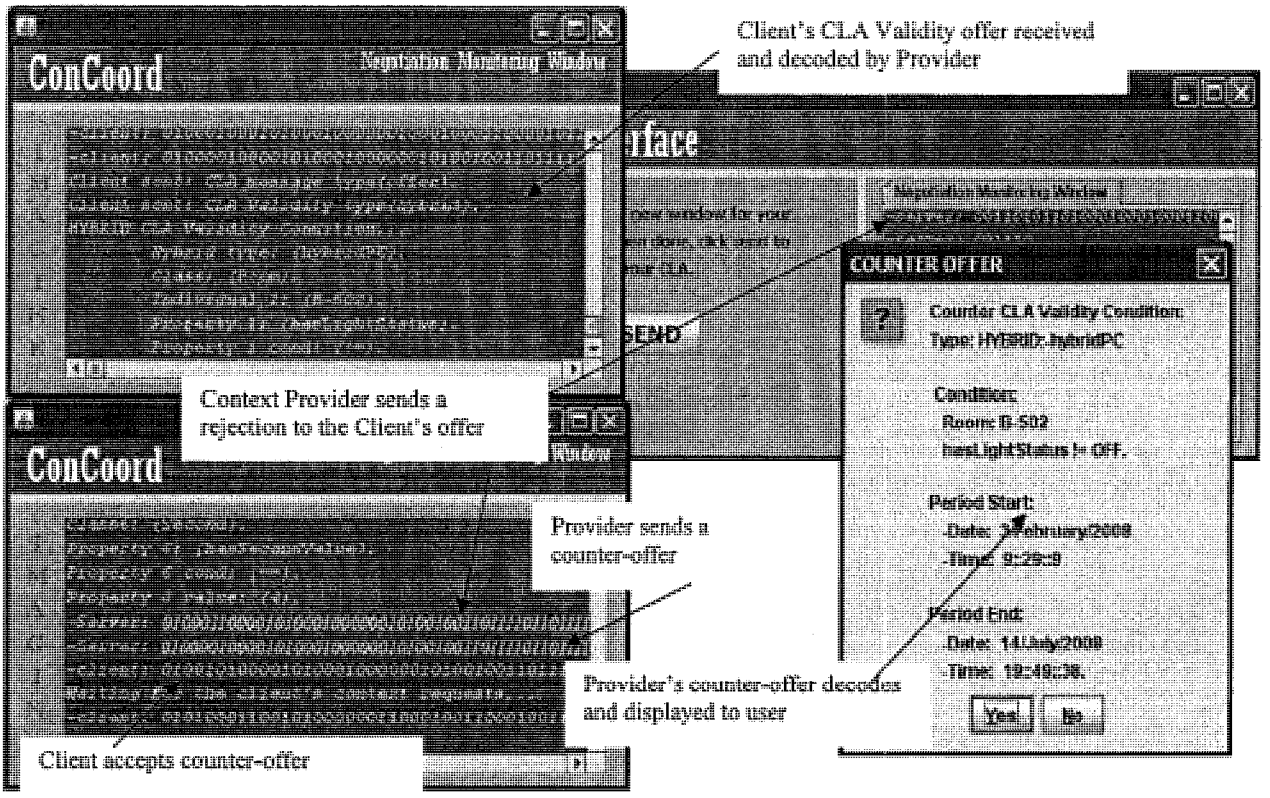


Figure 6-18. Provider rejection and counter offer for CLA Validity conditions

Once the CLA Validity Conditions have been agreed on, the Client and Provider proceed to perform negotiations for the context information Clients are interested in receiving. There are three types of context requests: Notification requests, Priority Value requests, and Class Value requests.

In this example, we will be making two context requests. The first is a request to receive a notification whenever the temperature within room A-508 is not hot, or when the humidity level within room A-504 is less than, or equal to, 95%. This request is shown in figure 6-19. The Provider evaluates the offer received from the Client, and if the Provider's current state allows it to deliver the needed context information with no conflicts between the requested context information and that presented in the Client's Views, then the Client receives an acceptance message, in our example. The acceptance message received by the MI component on the Client's side is decoded and displayed to the user as a notification window.

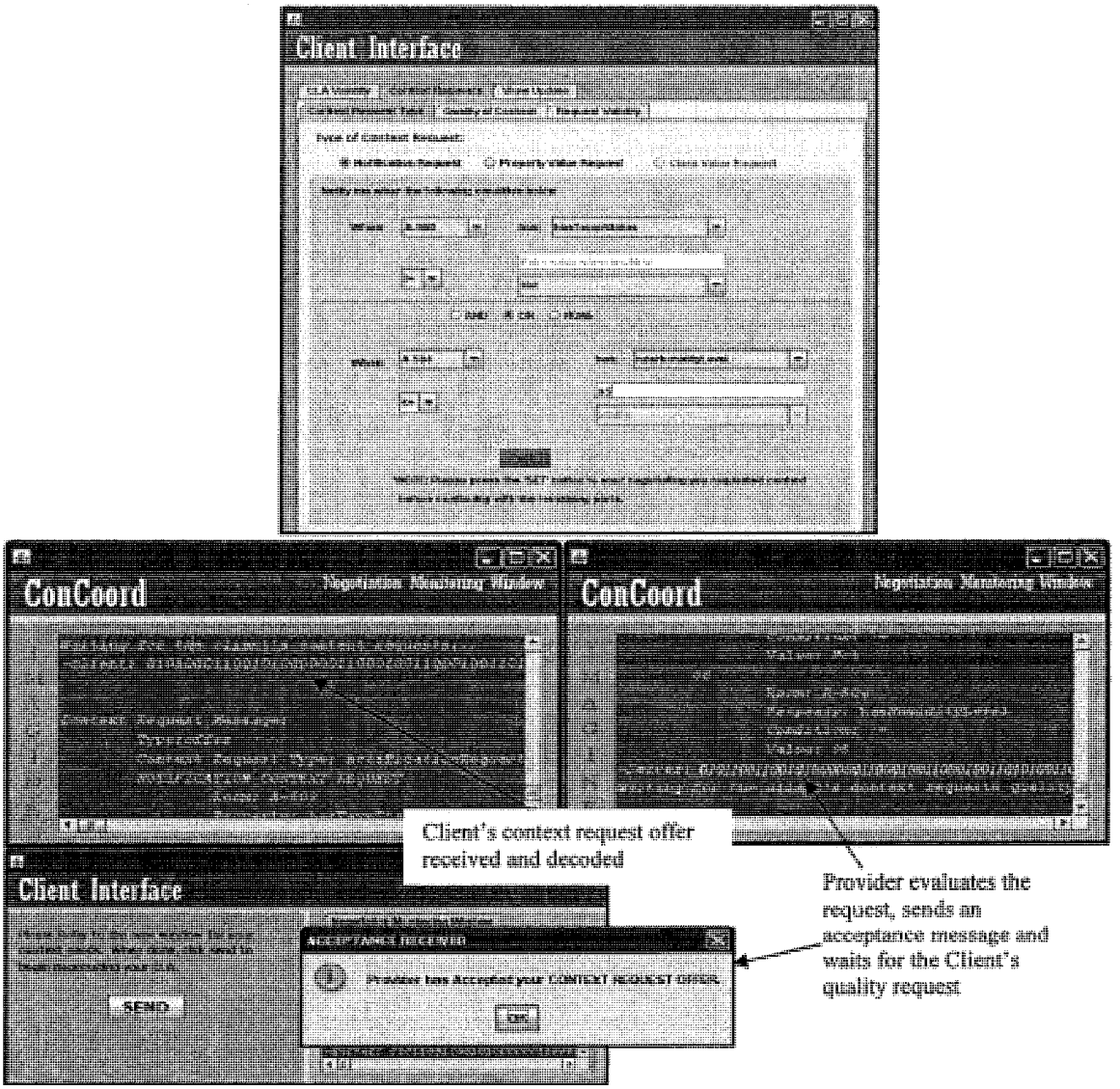


Figure 6-19. Context request formation and offer exchange

The context Provider then enters a waiting state during which the Client must submit the quality of context conditions it expects the requested context information to meet prior to delivery by the Provider.

The graphical prototype provides the user with three types of quality measures: accuracy, freshness, and certainty. For the above request, the user was interested only in receiving context information if the Provider could guarantee at least 80% accuracy, as shown in figure 6-20. The context Provider rejects the Client's request, sending a counter-offer informing the Client that it can only provide the requested information with 75% accuracy.

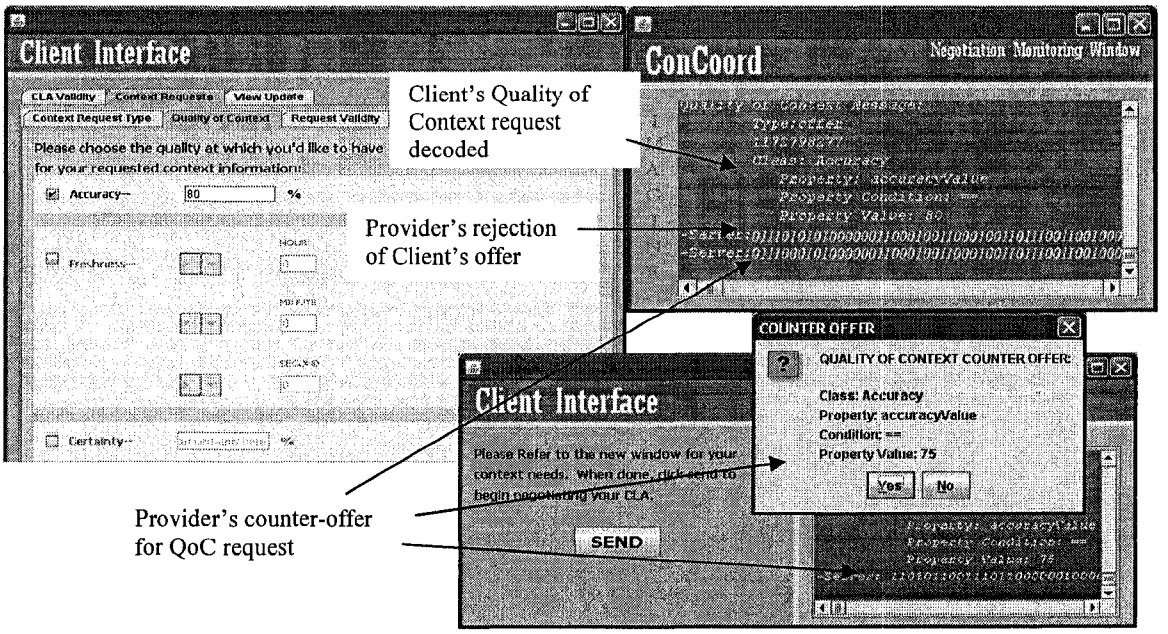


Figure 6-20. Wireless Mote and sensing board

If the client accepts the Provider's counter-offer, both parties proceed to the next step: negotiating the context request's Applicability Conditions. Clients are provided with the same options for performing their Context Applicability requests as those shown earlier for the CLA Validity Conditions.

Figure 6-21 shows the Client's interest in having the previous context request with an applicability condition that is periodic, from June 6th, 2008 at

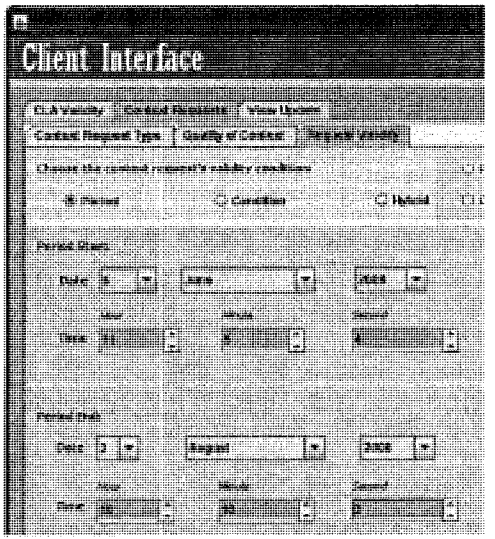


Figure 6-21. Context request Applicability condition formation.

11:05:04 A.M. to August 3rd, 2008 at 10:10:03 A.M. This request is accepted by the Provider and an acceptance message is subsequently delivered to the Client.

At this point in the negotiation session, the Client has fully established one context request with its respective Quality of Context request, as well as the Request Applicability condition. In chapter 5, we had indicated that Clients are permitted to have multiple context requests in every Context-Level Agreement (CLA) established, by simply repeating the steps for the Context Request, Quality of Context Request, and Applicability condition for every context information request.

Our example will include a second context information request: a Property Value request. Property Value requests differ from Notification Requests in that the most updated context value for a specific property is delivered to Clients. Alternatively, Notification requests simply send Clients a notification whenever the condition established with the Provider is validated. In figure 6-22, we show a Client's request for the current temperature level and humidity status for room A-504. The Provider accepts the requested offer and replies with an acceptance message to the Client.

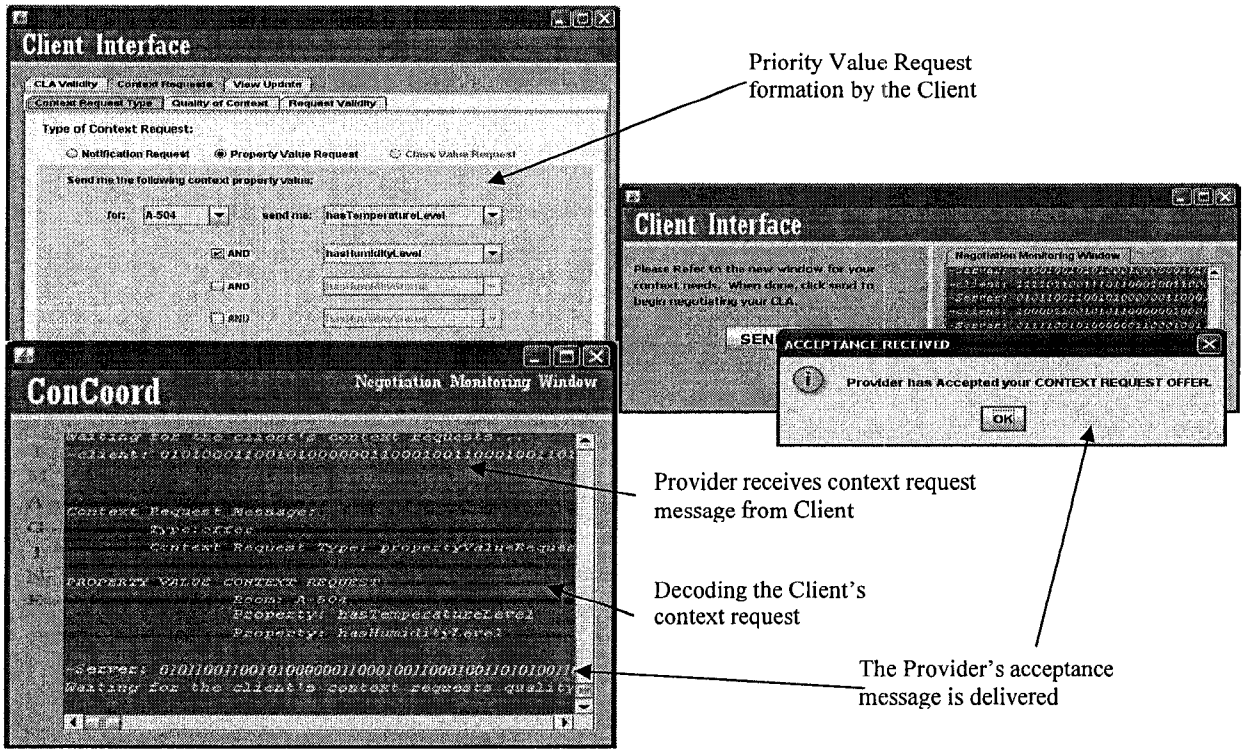


Figure 6-22. Priority value request formation and offer exchange

The Client continues to negotiate the Quality of Context levels and Applicability conditions particular to this context request, as was done earlier with the first context request. Once the Client is satisfied with all context requests included in its current CLA negotiations, the final step remaining concerns the method in which the Client is interested in receiving updates to changes that may occur to its Views. As noted earlier, these changes stem from increases or decreases in the Provider's capabilities, or from CLAs establishes with other Clients.

The graphical interface for Clients gives users four possible choices. The first is *No_Update*, which provides Clients with no updates whatsoever. The second is *Periodic*, a method through which the Provider periodically supplies the Client with new Views every time the period expires. The third option, and the one used in this example, is that of *Upon_Request*. This method supplies the Client with its latest View every time the Client sends a request to the Provider. Finally, the fourth method is *On_Change*, which provides the Client with an updated View every time a modification to the View is performed on the Provider's side.

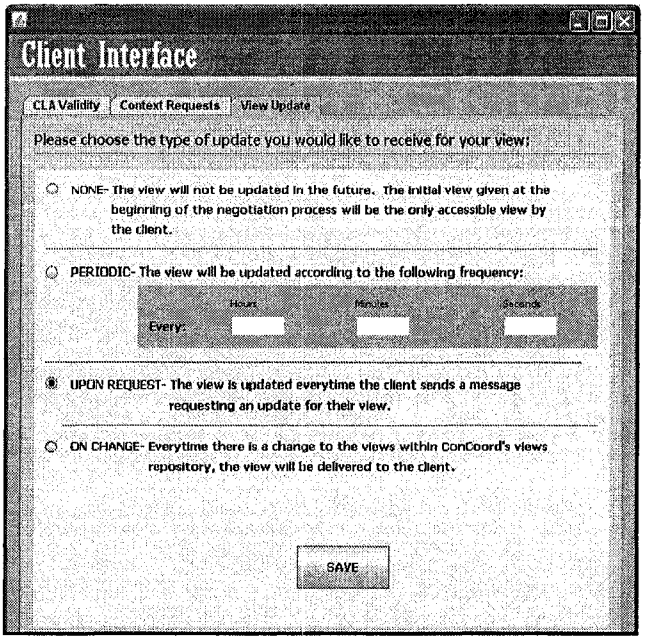


Figure 6-23. View update request formation

Once the Client and Provider agree on the View Update method, context-level negotiations are completed and the CLA is successfully established between the two parties. It is the Provider's responsibility to meet the requirements and expectations it had accepted during its negotiations with the Client.

As explained in section 5.2.7, the policy-based CLA enforcement suggested in our architecture has been replaced by a Java-based multithreaded programming method. This method provides only a temporary solution here, since policy generation from established CLAs remains a topic for future research. Still, our current approach satisfactorily meets our requirements for enforcing CLAs established between negotiating Clients and Providers. The ConCoord Access Point unit forwards the finalized CLA to a *CLA Enforcer Generator*

unit, whose responsibility is to generate Java classes capable of monitoring the CLA's conditions. The method used to monitor and enforce such CLA requires the initiation multiple threads for each finalized CLA.

Drawing on the current example, the first step of the CLA Enforcer Generator is to generate a java Class whose sole responsibility is to enforce the CLA Validity Condition. The CLA Validity Condition in the above example was Hybrid Periodic Conditional, whose validation was from February 3rd, 2008 09:23:09 A.M. until July 14th, 2008 at 19:49:38 P.M. if the lights within room B-502 were not turned off.

Translating this CLA condition into a Java condition follows the subsequent pseudo code that runs infinitely within a thread as long as the CLA has not been canceled between the Client and Provider:

```
WHILE (CLA is valid) {
    IF (current date_time >= start date_time && current date_time < end date_time) {
        IF (hasLightStatus (B-504) != "OFF") {
            IF (CLA is already active) {
                //CLA has already been triggered. Repeat evaluation of activation condition.
            }
            ELSE {
                // Instantiate context request classes.
                // Start context requests' threads.
            }
        }
        ELSE IF (hasLightStatus (B-504) == "OFF") {
            IF (CLA already active) {
                // Stop all context request threads that had been started.
            }
            ELSE {
                // CLA is still inactive, retrieve updated context values from Global Concrete Ontology
                and reevaluate CLA Validity condition.
            }
        }
    }
    ELSE IF (current date_time >= end date_time) {
        IF (CLA is already active) {
            // Stop all context request threads that had been started.
        }
        ELSE {
            // CLA is inactive. No action taken.
        }
    }
}
```

The above pseudo code is a general representation of the actual Java conditions generated for the sample CLA Validity condition discussed earlier. In figure 6-24, we provide a snapshot of the partial code used to monitor the CLA validity conditions. If the CLA validity

condition is found to hold validly, then (since that the CLA was inactive prior to this time), a thread is started for each context request Class. Since we had two separate context requests in our example, two threads were started in figure 6-24.

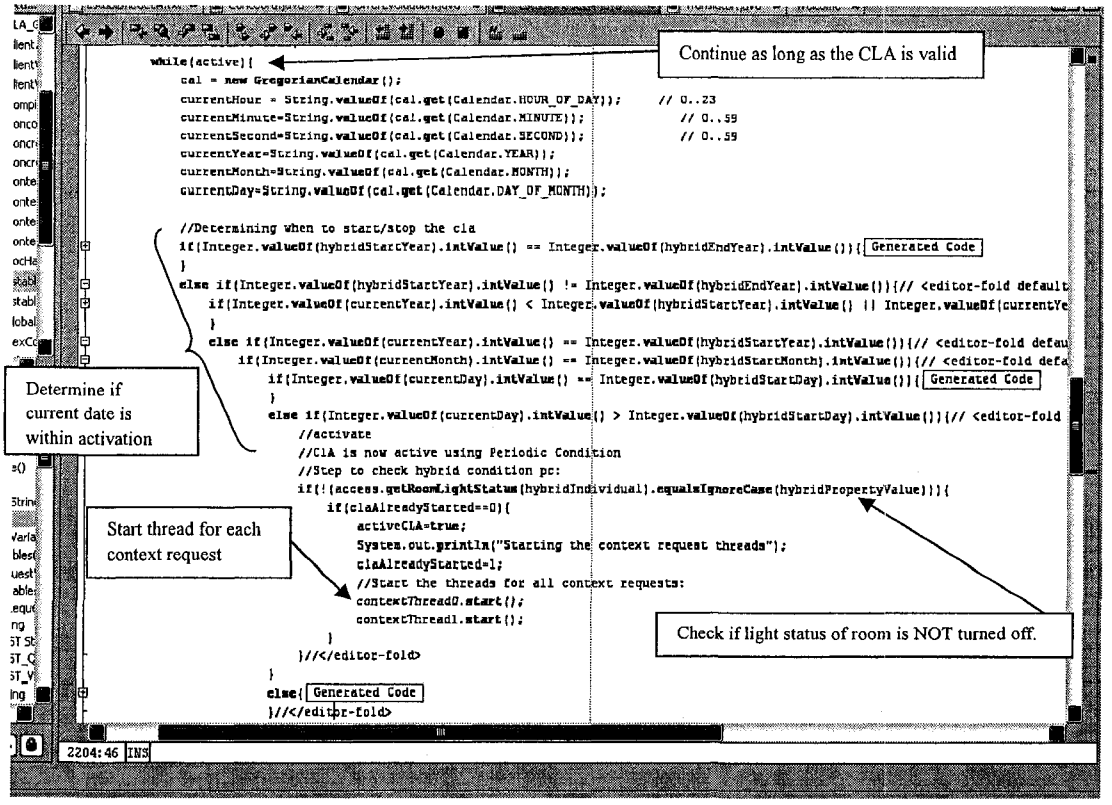


Figure 6-24. Generated CLA validity condition monitoring class

Each context request within a CLA is given a separate class where the thread triggered by the CLA Validity Condition class exists. This thread acts by continuously evaluating the Request Applicability condition determining when context information should be delivered to the Client. Once the condition is validated, the needed context information is retrieved from the GCO and delivered to the Client based on the type of request made - Notification, Property Value, or Class Value.

The second context request in the above example required the Provider to supply the Client with room A-504's humidity and temperature level properties. If we assume that this request's applicability condition is a WHILE Conditional, and that the condition the Client was interested in was that of the property values being only delivered if the current humidity status of room A-504 was NOT "Very Humid", then the pseudo code for this context request would resemble the following:

```
WHILE (true) {
    WHILE ( Humidity_Status (A-504) != "Very Humid") {
        // Retrieve room A-504 current temperature level
        // Retrieve room A-504 current humidity level
        // Send property values to ConCoord for deliver to the Client
    }
}
```

In our prototype system, the above pseudo code was successfully applied in order to perform the necessary actions of monitoring the Applicability conditions, retrieving the required context information, and delivering it to ConCoord Access Point for delivery to the Client. This is shown in the code in figure 6-25.

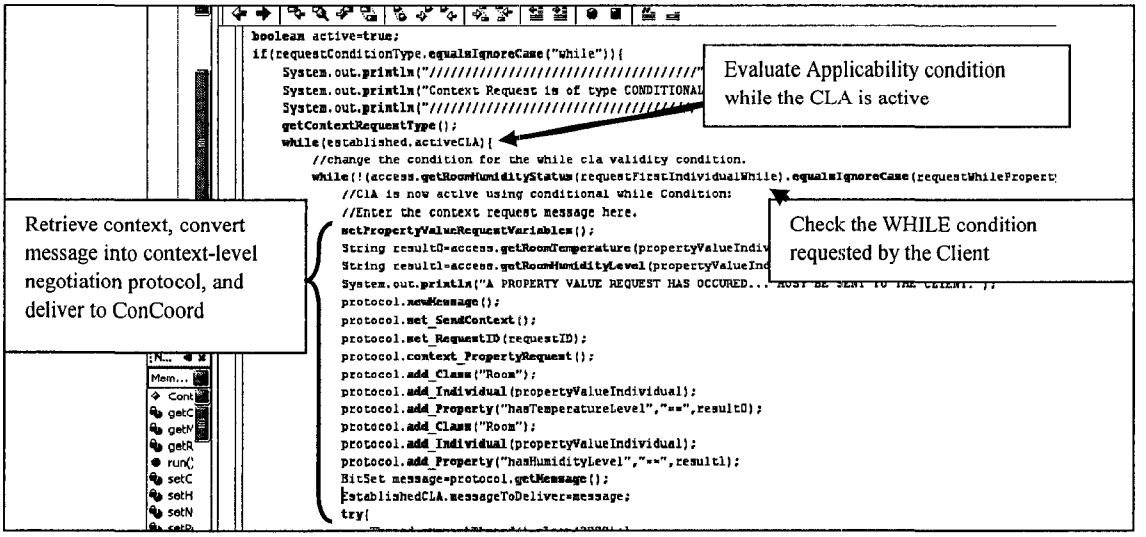


Figure 6-25. Generated context request monitoring class

Finally, context information that is received by the ConCoord Access Point is delivered to the Client in order to meet the requirements of the established CLA. We have provided a graphical window through which context messages received and decoded by the Client MI are displayed to the Client. Figure 6-26 shows the raw and decoded messages received by the Client, for the two context requests in the above example.

The example provided within this section has shown the use of our graphical user interface prototype, which allows human users to perform context-level negotiations with context Providers by using the context-level negotiation protocol presented in chapter 5.

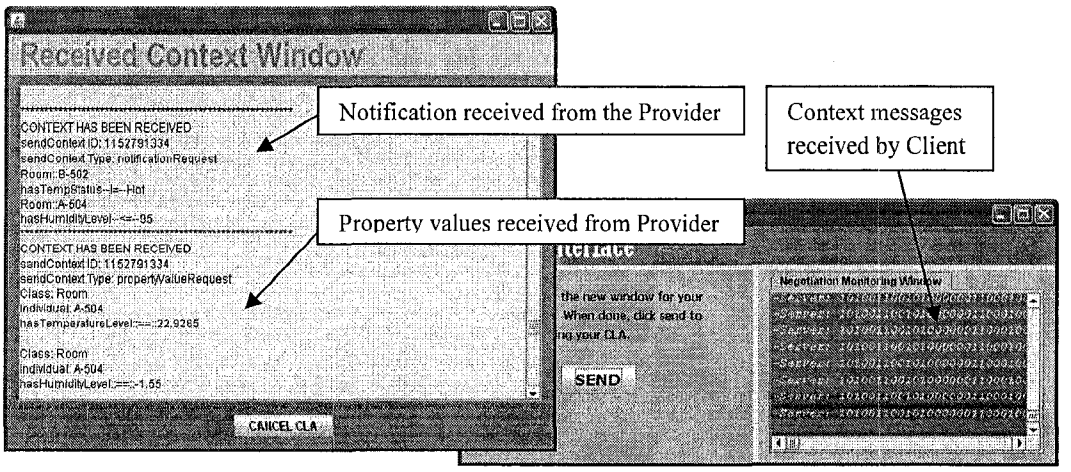


Figure 6-26. Successful delivery of context to Client

The example illustrated all the main components in our negotiation protocol: identifying the Client's identity and capabilities through Profiles exchange, informing the Client of its context access rights by supplying it with its respective Views, negotiating the CLA Validity conditions, negotiating context requests and with their respective applicability conditions and quality of context levels, negotiating the View update methods, and finally, enforcing the established CLAs by automatically generating multithreaded Java classes responsible for monitoring the CLA conditions and retrieving context information from the Global Concrete Ontology.

6.6. Chapter Summary

In this chapter, we presented a prototype system implementation representing the three main layers of our architecture: The Sensing layer, the context Provider layer, and the Client layer. The Sensing layer utilized a sensory toolkit provided by Crossbow, which included a set of wireless Motes with environmental sensor boards capable of sensing the surrounding temperature, humidity, light and pressure levels, and multi-directional acceleration. The Motes transmitted their sensed data to a base station whose duty was to forward all received messages to the context Provider's Context Acquisition Listener module.

The majority of the context Provider's components were implemented in Java 1.6. The mySQL-based context repository was updated with all context information arriving from sensors, along with those inferred by our inference engine. Our Global Skeletal Ontology was modeled using the OWL language. A subset of that ontology was extracted, extended,

and instantiated to create our Global Concrete Ontology based on the university domain within which the system was located, the users of the system, and the rooms from which context information was acquired.

We illustrated the negotiation steps performed by Clients and Providers in our ontology-based context-level negotiation protocol. Our examples illustrated the use of a graphical implementation that followed the seven steps illustrated earlier in figure 4-1. Our protocol was built on top of TCP/IP to gaurantee the successful delivery of messages between Clients and Providers. All steps leading to the establishment of a CLA were displayed, along with our unique method for the CLA's enforcements.

We showed how, through the dynamic generation of multithreaded Java classes based on conditions established during CLA negotiations, a CLA with all its conditions validated results in the retrieval of required context from the GCO for delivery to the Client. This proves the efficacy and efficiency of using context-level negotiations between context consumers and Providers.

Evaluating the performance of context-level negotiations in context-aware systems is extremely difficult. There are no available standards or mechanisms by which designer can evaluate their designs or compare their designs' performances to thos of available systems in the community. Context-Level Negotiations is a new concept and has not been explored in any other studies and proposals in the field of context-awareness. Therefore to compensate for this lack of comparable systems, we have provided a prototype implementation that shows the feasability of using context-level negotiations between context consumers and context Providers. These Providers represented context-aware middleware. The use of an ontology-based negotiation protocol independent from the design of context sources and sensors, as well as that of the context Providers, allows Clients to acquire their needed context information without having to directly sense their needed context with their personal sensors - which might drain power in mobile devices with limited resources, power, and size.

The context-aware system architecture prototype implementation was used as a test bed for our proposed context-level negotiation protocol. The successful tests performed in establishing and enforcing the negotiated CLAs illustrated the feasibility of incorporating context-level negotiations into currently available or future context-aware systems.

The negotiation protocol presented in this thesis may have one drawback: the possibility of exchanging large numbers of messages between Clients and Providers. The size of Client Profiles and Views exchanged during the initial stages of negotiation can vary considerably from one system to another. This is because context Clients and Providers can vary in their abilities and interests according to the domains within which they exist and their particular designs. For example, a context Provider within an office building's domain, which is usually an enclosed domain with a well-defined set of possible Clients distinguishable by their roles, expects to receive Profiles and supply Views that are different from a Provider within the domain of a more publicly accessible location such as a museum.

Fortunately, the effects of exchanging large Profiles and Views are minimized due to two reasons. First, the exchanges are performed only once at the beginning of the negotiation session. Clients are only required to supply Providers with their Profiles at the time of initial contact. Any negotiations performed in the future do not require the exchange of Profiles since the Provider has been familiarized with the Client's identity and capabilities. Similarly, Views are only exchanged at the start of negotiations and whenever necessary based on the negotiated View Update conditions. Secondly, the OWL-based Profiles and Views use the ASCII-based RDF and Extensible Markup Language for their syntax. This reduces the amount of data exchanged between Clients and Providers which allows them to describe all ontology classes and properties using text. In addition, interpreting the semantics of OWL-based Views received by Clients does not introduce any overload on Clients.

Negotiations can enrich applications running on mobile devices by supplying them with a variety of context information. The in/out board and DUMMBO applications presented in [33] using widgets could easily be implemented through our context-aware architecture and negotiation protocol. The in/out board can establish a CLA with the Provider to supply it with context information about office members currently located within the building. Similarly, a DUMMBO-specified CLA can require the Provider to supply information about the number of users standing next to a whiteboard, the audio being recorded in the meeting room, and the notes written onto the whiteboard. Furthermore, CLA negotiations can enhance context-aware applications such as the above two and others by negotiating the

frequencies, conditions, and quality of context levels that must be met prior to delivering the context information to the applications (Clients).

Chapter 7

Conclusion and Future Work

7.1. Conclusion and Thesis Contribution

The goal of this thesis has been the development of a context-level negotiation protocol to establish Context-Level Agreements (CLA) in context-aware systems. We found two main problems with the majority of current context-aware systems. The first problem was weakness in expressing complex inter-context relationships, stemming from less capable approaches to modeling acquired context information. The other problem was the possibility of network flooding, unrestricted access to private context information, or the inability of context consumers to limit or personalize received context – all stemming from limitations in other systems' context-dissemination methods.

We have thus provided a generalized ontology context model using the OWL-language. This ontology model solves many of the problems with currently existing non-ontology based solutions, which lack a clear method of modeling knowledge within context-aware systems, are incapable of modeling complex relationships existing between different concepts, and are not easily sharable and extensible. Our ontology model also solved some of the problems that other ontologies faced. Many existing ontologies were too restrictive for the domains their designers built them to meet, or they were too abstract and difficult to extend in order to be useful in diverse context-aware systems. Therefore, our Global Skeletal Ontology (GSO) provides an easy solution to modeling context knowledge using ontologies, by proving an ontology that was neither restricted to specific domains, nor too abstract for easy use and extension to other domains.

The context dissemination problem seen in many context-aware systems led us to the conclusion that there should be a method for organizing and controlling the delivery of context information within context-aware systems. As a result, we developed the idea of establishing Context-Level Agreements (CLA) between Clients, or context consumers, and

context Providers. These agreements are used to clearly define the context information Clients require, and the responsibilities of Providers delivering that context information.

For such agreements to be established, we have designed a context-level negotiation protocol that is independent of any existing protocols in use current use. Our negotiation protocol was derived directly from the general structural organization of OWL-based ontologies which represents knowledge through a set of classes, properties, relationships and individuals. In addition, the protocol provides a sequential seven-step method to reaching the desired Context-Level Agreements. The exchange of ontology-based Profiles and Views supplies Providers with the necessary information for recognizing the identity and, in some cases, the abilities of negotiating clients. This exchange also provides Clients with enough information for them to send valid context request messages. In addition, our protocol grants Clients the ability to negotiate the conditions under which they desire to activate their CLAs, the context information they wish to receive, the conditions under which they wish to receive it, and quality of context levels that need to be met before any context is delivered to the Client.

Since no existing context-aware system architectures and middleware supported context-level negotiations or CLA enforcement, we also proposed a new context-aware system architecture that can perform the desired negotiations and agreement-enforcement. We propose the use of a three-layered architecture composed of context sources, context providers, and context consumers. Our design of context providers utilize ontologies to create a clear method for modeling the acquired context information, inferring new context, and generating Views describing the Clients' context access rights. The context Provider design also allows Providers to perform context-level negotiations with Clients and to enforce the generated CLAs through automatic policy generation. As well, our architecture exempts Clients from the need to access to their entire spectrum of context information through their personal set of sensors, or to have direct contact with context sources whose message protocol formats are design-specific. Instead, the architecture permits Clients to acquire their needed context information through our ontology-based negotiation protocols and Context-Level Agreements.

To prove the validity of our system architecture and negotiation protocol, we have also developed a prototype system based on our proposal. The prototype includes a sensor layer composed of a set of physical environmental sensors. The context Provider and Client were designed and tested successfully, and they used the context-level negotiation protocol to compose the needed Context-Level Agreement. Finally, to enforce the established agreements, Providers were designed with the ability to automatically generate multi-threaded Java classes. These monitored the conditions within the agreements in order to deliver the required context information to Clients based on the established CLAs.

7.2. Future Work

There are several remaining issues that require closer investigation. Given the newness of the concepts introduced within our context-aware system architecture, the automatic generation of CLA-based policies, their enforcement, and the possible effects on established CLAs remains an area in need of greater research. We are planning to further explore this field in the near future. We were capable of overcoming the problems of CLA-enforcement through automatic class and condition generations during the system runtime, as was seen in chapter 6. However, possible occurrence of a scenario with large numbers of threads may hinder the Provider's ability to serve a large number of Clients, as its resources would be quickly strained.

In summary, we are constantly reviewing our proposed context-level negotiation protocol, adding and modifying its structure in search of a protocol that can meet the requirements of Clients and Providers in all context-aware systems, regardless of their domains or capabilities. We believe that the current protocol can sufficiently meet these requirements. We continue to improve on this system and protocol by adding new options and capabilities as our research progresses.

References

[1] A. Al-bar and I. Wakeman, "A Survey of Adaptive Applications in Mobile Computing", in *Proceedings of International Conference on Distributed Computing Systems Workshop*, 2001, pp. 246-254.

[2] V. Akman and M. Surav, "The Use of Situation Theory in Context Modeling," in *International Journal on Computational Intelligence*, March 2003, pp. 427–438.

[3] Lexico Publishing Group, "The American Heritage Dictionary of the English Language, 4th Edition," October 2007, http://dictionary.reference.com/browse/context

[4] B. Schilit and M. Theimer, "Disseminating active map information to mobile hosts," *IEEE Network*, vol. 8, pp. 22-32, October 1994.

[5] A. Schmidt, M. Beigl, and H.W. Gellersen, "There is more to Context than Location" in *Proceedings of the International Workshop on Interactive Applications of Mobile Computing*, 1998, pp. 893-901.

[6] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications", in *First International Workshop on Mobile Computing Systems and Applications*, 1994, pp.85-90.

[7] A. Dey, "Understanding and Using Context", *Personal and Ubiquitous Computing*, vol. 5, pp. 4-7, February 2001.

[8] K. Ragab, N. Kaji, Y. Horikoshi, H. Kuriyama, and K. Mori, "Autonomous Decentralized Community Communication for Information Dissemination", *IEEE Internet Computing*, vol. 8, pp.29-36, May 2004.

[9] B. Truong, Y. Lee, and S. Lee, "Modeling and Reasoning About Uncertainty in Context-Aware Systems," in *Proceedings of IEEE International Conference on e-Business Engineering*, 2005, pp. 102-109.

[10] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *IEEE Workshop on Mobile Computing Systems and Applications*, 1994, pp. 85-90.

[11] J. Bauer, "Identification and Modeling of Contexts for Different Information Scenarios in Air Traffic," Diplomarbeit, Berlin Institute of Technology, Berlin, Germany, 2003.

[12] D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," *IEEE Computer Society Press*, vol. 36, pp. 25-31, March 2003.

[13] A. Held, S. Buchholz, and A. Schill, "Modeling of Context Information for Pervasive Computing Applications", in *Proceedings of 6th World Multiconference on Systematics, Cybernetics and Informatics*, 2002, pp. 79-117.

[14] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Generating Context Management Infrastructure from High-Level Context Models", in *Industrial Track Proceedings of the 4th International Conference on Mobile Data Management*, 2003, pp. 1-6.

[15] J. Bacon, J. Bates, and D. Halls, "Location-Oriented Multimedia", *IEEE Personal Communications*, vol. 4, pp. 48-57, October 1997.

[16] M. Smith (Michael.smith@eds.com), C. Welty (chris.welty@us.ibm.com), and D. McGuinness (dlm@ksl.stanford.edu), "OWL Web Ontology Language Guide. W3C Recommendation.", W3C Recommendation, February 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/

[17] M. Strimpakou, I. Roussaki, and M. Anagnostou, "A Context Ontology for Pervasive Service Provision," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, 2006, pp. 775-779.

[18] T. Gu, H. Pung, and D. Zhang, "A Middleware for Building Context-Aware Mobile Services", in *Proceedings of IEEE Vehicular Technology Conference*, 2004, pp. 2656-2660.

[19] H. Chen, "An Intelligent Broker Architecture for Pervasive Context-Aware Systems", Ph.D. dissertation, University of Maryland, Baltimore Country, USA, 2004.

[20] P. Krpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.J. Malm, "Managing Context Information in Mobile Devices," *IEEE Pervasive Computing*, vol. 2, pp. 42-51, September 2003.

[21] P. Fahy and S. Clarke, "CASS-Middleware for Mobile Context-Aware Applications", presented at Workshop on Context Awareness in the Second International Conference on Mobile Systems, Applications, and Services, Boston, Massachusetts, USA, June 2004.

[22] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, and J. Almann, "Context-Awareness on Mobile Devices-the Hydrogen Approach," in *Proceedings of the 36th Hawaii International Conference on System Sciences*, 2003, pp. 292-301.

[23] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, "A Middleware Infrastructure for Active Spaces," *IEEE Pervasive Computing*, vol. 1, pp. 74-83, December 2002.

[24] K. Markus (mgk25@cl.cam.ac.uk), "A Summary of the International Standard Date and Time Notation", University of Cambridge, Computer Laboratory, December 1995, http://www.cl.cam.ac.uk/~mgk25/iso-time.html.

[25] Parallel Understanding Systems Group, Department of Computer Science, University of Maryland at College Park, "SHOE-Simple HTML Ontology Extensions – Organization Ontology," October 2002, http://www.cs.umd.edu/projects/plus/SHOE/onts/org1.0.html

[26] P. Hickman (peter@semantico.com), "Ruby Simple Inference Engine", http://homepage.ntlworld.com/peterhi/sie.html, February 2001.

[27] B. McBride (bwm@hplb.hpl.hp.com) and C. Dollin (chris.dollin@hp.com), "An Introduction to RDF and the Jena RDF API", post to Jena.sourceforge.net, http://jena.sourceforge.net/tutorial/RDF_API/index.html, July 31, 2007.

[28] Crossbow Technology, Inc., "MoteView User Manual - Revision A," May 2007, http://www.xbow.com/Support/Support_pdf_files/MoteView_Users_Manual.pdf

[29] Crossbow Technology Inc., "MoteWorks Getting Started – Revision E," 2007, http://www.xbow.com/Support/Support_pdf_files/MoteWorks_Getting_Started_Guide.pdf

[30] P. Levis (pal@cs.stanford.edu), "TinyOS Programming", post to Stanford Computer Systems Laboratory, http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf, June 28, 2006.

[31] Sandia National Laboratories, Livermore, California, "Jess the Rule Engine for the Java Platform- The Jess FAQ," August 2008, http://www.jessrules.com/FAQ.shtml

[32] J. Hunter (jasonhunter@servlets.com), "JDOM and XML Parsing- JDOM makes XML manipulation in Java easier than ever," Oracle Magazine, September 2002, November 2002, and March 2003.

[33] D. Salber, A. Dey, and G. Abowd, "The Context Toolkit: Aiding the Development of Context-Enabled Applications", in *Proceedings of CHI'99 Human Factors in Computing Systems Conference*, 1999, pp. 434-441.

Publications

[RIDH 08a] Y. Al Ridhawi and A. Karmouch, "Ontology-Based Negotiation Protocol and Context-Level Agreements", To appear in *International Conference on Intelligent Environments (IE08)*, Seattle, USA, 2008.

[RIDH 08b] Y. Al Ridhawi and A. Karmouch, "Ontology-Based Context-Level Agreements and Negotiation Protocol", To appear in *Fifth International Workshop on Next Generation Networking Middleware (NGNM 2008)*, Samos Island, Greece, 2008.