Context retrieval and distribution in a mobile distributed environment

Alisa Devlic and Erik Klintskog

Appear Networks
Kista Science Tower, 16451 Kista, Sweden
{alisa.devlic,erik.klintskog}@appearnetworks.com
http://www.appearnetworks.com

Abstract. Context-aware services are gaining momentum in mobile computing. To enable rapid development of context-aware services, context information has to be retrieved from the environment, modeled, processed, and distributed to these services.
MIDAS is a European research project concerning 3G and beyond, which aims to define and implement a platform to simplify and speed up the task of developing and deploying mobile applications and services. MIDAS context engine provides mechanisms to retrieve, model, synthesize, and distribute context information in a distributed, mobile environment. This paper presents a way to retrieve and distribute context information using context queries and triggers. A novel approach to perform context synthesis will be presented using operators.

Key words: Distributed context engine, context query, context trigger, context synthesis, context retrieval and distribution

1 Introduction

Mobile applications, unlike desktop applications, need to adapt to environmental changes such as change in the user's location. As the user changes location, the situation and environment around him/her changes (nearby people, places, objects), i.e., different context information becomes available [1]. The challenge is to utilize available sources of context information within a suitable time and make this information visible and usable by applications in a distributed system.

1.1 Motivation

This paper describes an approach to be used by a context engine to retrieve distributed context information concerning users and their surroundings, and to provide this context information to mobile applications.

The context information is retrieved from its source based upon an application's request. *Context information sources* are wrappers that provide semantic markup for raw context information coming from physical and virtual sensors, which can be separated hardware devices or software running on the user's

device. Often, the node where the context source resides is different from the node which runs the end user's application.

Applications use high-level context information, which is abstracted from the information obtained from context information sources. This high-level context information is inferred from the existing information using application-specific inference rules. This reasoning process is called *context synthesis*. The problem with context synthesis using existing rule-based reasoning are the long response times which the end-user (or their application) needs to wait for result to his context query [2], especially when large data sets and rule sets are applied [3].

Contexts in mobile environments may change at very high rate. A problem arises if contexts are updated in the database as soon as new context information becomes available, but this information is subsequently not used by context consumers (i.e. applications). Frequent transfer of context information over the network leads to high resource consumption on the devices which host context information sources (in terms of power consumption, bandwidth utilized, etc.).

1.2 Contribution

Distributed context retrieval and distribution raises many issues, some of which have just been elaborated. The proposed solutions for those problems are the following:

- *Retrieval of context information from the remote context information source.* Our approach is to retrieve context information directly from its source only when it is needed, rather than simply when new value is available. The retrieved value is also cached in a database until its validity expires.
- *Context queries and context triggers for context retrieval.* To enable simple application requests for context information, while hiding from them the underlying transformation process, we have utilized *context queries* and *context triggers*. *Context queries* are used for stateless retrieval of context information, e.g. "What is the temperature of this room". *Context triggers* are queries for stateful context information, these trigger a predefined action when context information reaches a specified state, e.g. "Alert me when the temperature of this room reaches 28 degrees". Context queries can be simple (i.e., requiring only a database query for the specific context information) or complex (i.e., requiring context synthesis).
- *Context synthesis using operators.* The approach we propose is to use *operators* for context synthesis. Operators are functions that take as input certain context information, and produce as output new context information (e.g., "UsersInRange" takes "UserID=alice" and "Range=500" as inputs, and produces "UserIDBag=bob, ted" as output). Operators are described with a common ontology, similar to the representation of context. They are implemented as programs that perform the operation described in the operator's ontology. Operators can be application- or domain-specific. Therefore, they are meant to be inserted into the context engine by the application or system developer by extending the existing ontology and implementing the operator's function for an application or domain purpose.

2 Context management

Context management encompasses activities starting with acquiring context information, context modeling and reasoning, to providing high-level context information to services relevant to the end user. This paper focuses on providing context information produced by context providers to context consumers.

A distributed approach for retrieving, synthesizing, and disseminating context information for all end users to mobile applications, based upon the MIDAS project, will be presented in this paper. In MIDAS, context consumers are mobile applications that retrieve context information using a context engine. The context engine provides mechanisms to retrieve, model, synthesize, and distribute context information in a distributed, mobile environment.

2.1 Context query

A context query is a request for context information. The context query can use an operator to execute an operation in order to produce the desired context information. Operators can be considered as functions that take an input (or list of inputs), perform an operation, and produce an output. Operators are briefly described in Section 3, which illustrates the process of context synthesis.

We split context queries into two categories, depending on whether they contain an operator or not: *complex* and *simple context queries*. Context queries that contain an operator, whose inputs determine the context information that needs to be obtained, are called complex queries. The other type of context query obtains its context information directly from the repository, i.e. the database, without using operators, these are called simple queries. A context query also contains a so called context quantifier, which influences the way context information is retrieved (specifically whether one waits for *one* or for *all* context values, or waits for context values for a specified *number of milliseconds*), before composing and sending back the result.

The context engine distinguishes between *static* and *dynamic context information*, and issues *local* and *remote context queries* accordingly, as illustrated in Fig. 1. Static context information is the type of context information that doesn't vary with time and/or it is not influenced by other processes (e.g. user profiles). Local context queries are queries for static context information from the context repository on the same node. Dynamic context information changes frequently (e.g. a user's location). Such information is retrieved by sending remote context queries to context information sources (on remote nodes), providing this context information. Context information sources request the raw data from sensors and model this data as context information. This context information is cached locally on the context repository.

The context engine contains a context mapping component that serves as a yellow pages for finding addresses of context information sources that provide different types of context information. The context mapping component issues local and remote context queries. Communication between the context mapping component and the context repository is based on queries/responses.

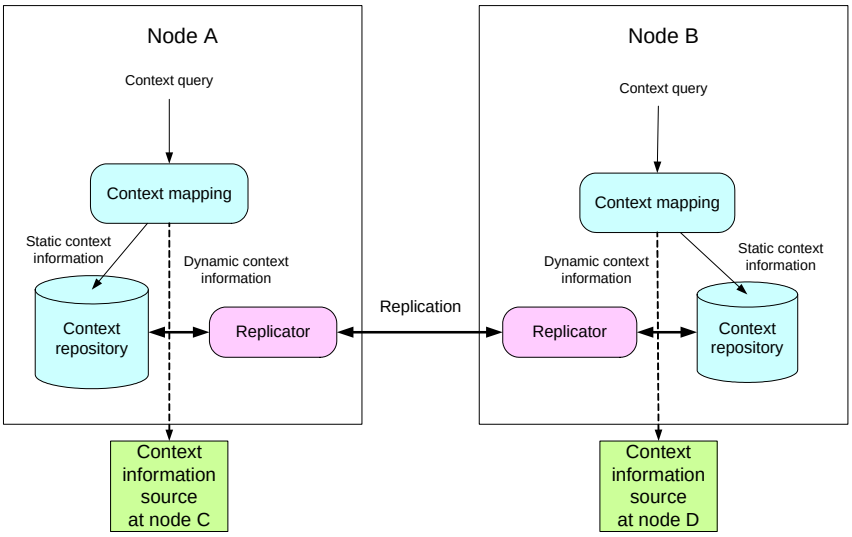

Fig. 1. Local and remote context query

Communication between the context mapping component and context information source(s) is(are) realized by activating context triggers and exchange of queries/responses.

Static context information is stored in the context repository and is replicated across nodes in a distributed system (see Fig. 1). Details about the replication mechanism will not be elaborated upon in this paper.

2.2 Remote retrieval of context information

As an example of the retrieval of location information from the remote context information source, Fig. 2 shows the mapping of context to its source and the querying of a remote source. Note that nodes A and C in Fig. 2 correspond to the nodes A and C in Fig. 1. The context mapping component on the node A maps context parameters to nodes that provide values for those parameters. For the location information, there is a mapping in the mapping component's table to nodes C and D (see Fig. 2). When the context query arrives and the context mapping component is asked for location, the first available node (node C), is selected.

The context mapping component will send the query for location to the node C, i.e. *C.getRemote(Location)*. The communication and routing component from the MIDAS framework will take care of finding the remote node and transferring the appropriate message.

On the node C, the context information source is realized by an application (e.g., GPS application that talks to a GPS sensor and wraps the GPS coordinates to location information). This application needs to register its instance and the

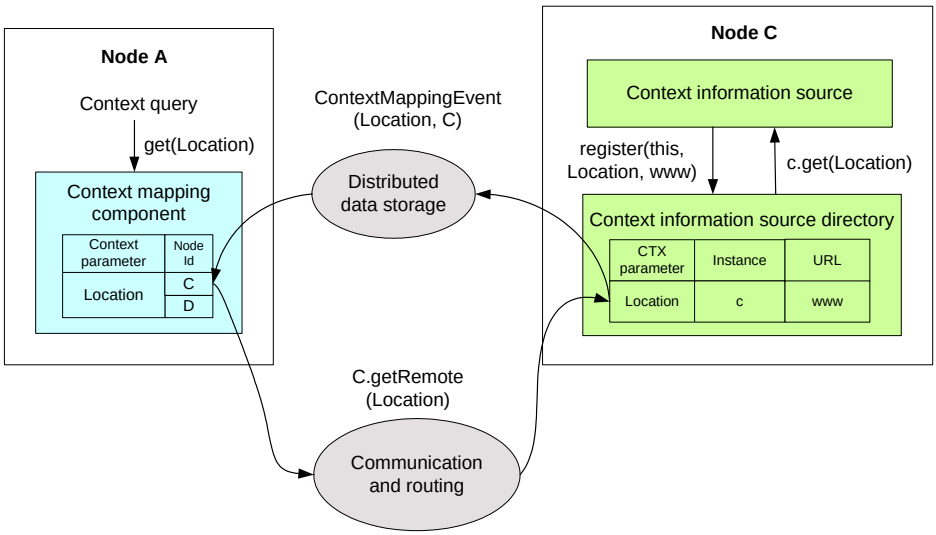

Fig. 2. Retrieval of location information from remote context information source

context parameter it provides to the context information source directory, at the startup. Optionally, this application can be replaced by a web service, in which case it would register its URL, instead of the application instance, to the context information source directory. After this registration is stored in the directory, the context mapping event is fired, containing the pair of context parameter and node identifier (i.e., *ContextMappingEvent(Location, C)*). The context mapping component will store this pair into its table. Thus, by listening to those events the context mapping component updates its context mappings. Distributed data storage acts as a mediator for context mapping events, as it stores all data in the MIDAS framework, and will pass the received event to the node A.

When the query for a location arrives at node C, the context information source directory will be searched for the instance of context information source, which will be invoked to get the current location (i.e., *c.get(Location)*).

2.3 Context trigger

Statefull context queries are implemented using context triggers. Context triggers execute a predefined action when a specified condition is fulfilled, i.e. context information has reached a certain state. A context trigger is specified with the context condition that needs to be fulfilled in order to trigger a context action, the context action itself, and the type of context condition event (" *OnEnter*" - when user enters or " *OnLeave*" - when user leaves the context). A context condition is specified as context parameter name-value pair. A context action specifies an action to be performed when the context condition event is fired, the time when the action should be triggered, the duration of the action, and/or the interval in which the specified action is periodically triggered.

When created, context triggers are inserted directly into context sources. Context sources pool sensors for context information and compare the retrieved value with the specified context condition. When the match is discovered, the context condition event is fired. The context engine listens to those events and executes an appropriate action.

3 Context synthesizing

Context synthesizing is a process of generating new knowledge (in the form of more abstract context), as a result of a reasoning process applied to context information that was already present in the system (e.g., deriving the abstract concept of weather by combining temperature, humidity, and wind speed). This context synthesis requires some rules (which are presented by operators in this paper) to drive the reasoning process forward.

In our distributed context retrieval, context synthesis begins with interpretation of a context query to retrieve a description and implementation of the operator that matches the requested operator's description in the query. The context query is represented by an expression of operators, arguments, and a context quantifier e.g. *(InRange (alice, 500, Users), All)*, which according to the context query definition means "find all users within 500 meters from alice".

The description of the retrieved operator specifies the operation performed by this operator, the required input arguments, and the output returned by this operation. Before the operation is performed, the missing input values are either obtained from the context information source (e.g. users locations) or are explicitly stated by user in the query (e.g. user id, range). The output of the operation is sent to the application as a result of the context query. This result is called a *synthesized context*, since it is generated by context synthesis.

3.1 Operators

This subsection will provide a specification of operators. Let **Op** be a set of operators:

$$\mathbf{Op} = \{op_1, op_2, \ldots, op_n\}, n \in \mathbb{N}$$

An operator $op_i \in \mathbf{Op}$ is represented with a bundle of *operator's description* and *implementation*:

$$op_i = \{\mathbf{desc}(op_i), \mathbf{impl}(op_i)\}$$

For every $op_i \in \mathbf{Op}$, let:

- **In** denote a set of required inputs $\mathbf{In} = \{in_1, \ldots, in_n\}, n \in \mathbb{N}$
- **Out** denote a set of possible outputs $\mathbf{Out} = \{out_1, \ldots, out_m\}, m \in \mathbb{N}$
- **F** denote an operation that takes provided inputs and produces an output $\mathbf{F} : In \rightarrow out_j$, $out_j \in Out$
- **Uses** denote a set of used operators $\mathbf{Uses} = \{op_i, \ldots, op_j\}, i, j \in \mathbb{N}$

An operator's description $\mathbf{desc}(op_i)$ is defined as:

$$\mathbf{desc}(op_i) = \{name_i, F_i, In_i, out_i, Uses_i\}$$

where:

- $name_i$ is the name of op_i
- F_i is the operation provided by op_i
- In_i list of inputs for F_i
- out_i an output produced as a result of F_i
- $Uses_i$ list of other (simpler) operators op_i uses in its execution

An operator's implementation $\mathbf{impl}(op_i)$ is defined as:

$$\mathbf{impl}(op_i) = \mathbf{impl}(F_i(In_i)) = out_i$$

The above definition specifies operator's implementation as an implementation of the operation F_i, which takes In_i as arguments, produces out_i as a result, and invokes implementations of used operators (i.e. $\mathbf{impl}(Uses_i)$) in its execution.

```
program F(In)
   begin
     if Uses is empty
        out=perform operation on In
     else
        for each op from Uses
            In_New=perform operation on In
            out=op.F(In_New)
     return out
end.
```

An example implementation of the operation F_i is the above program F(In). The program takes a list of inputs, here represented by variables *In*. At the beginning, the program checks if the list of used operators is empty, and performs specified operation on the list of inputs. If the list of used operators is not empty, then the program will invoke each operator and pass as arguments the newly obtained inputs. For simplicity operations of op_i and other operators in the program have the same name (i.e. F).

3.2 Operators Ontology

Consider the following context queries:

1. "Find all users in range 500m from Alice."
2. "Find all streets in range 500m from Kista Centrum."
3. "Find all towns in range 20km from Stockholm."
4. "Find all phones in range 50m from my office."

   ....

The number of these and other similar context queries (which are **very specific** and implement the same functionality, but take different input and produce different output types) is quite **extensive**. If each context query would employ its own implementation of an operator, it would significantly increase the database storage required along with the time needed to find the right one. Furthermore, the right operator might **not be found**, unless the exact relation between the requested and the desired operator's input is specified. Relationships between operators and their inputs and outputs are described in the operator's ontology (e.g. the context query asks for streets in the user's range, and available "InRange" operator implementations return postal codes instead of streets).

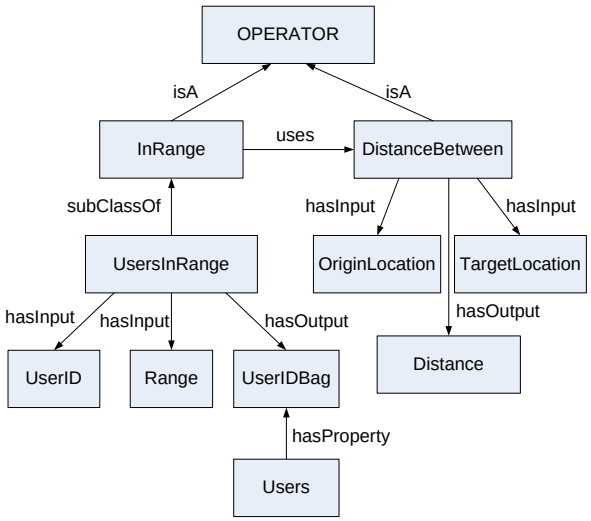

Fig. 3. Ontology of "InRange" operator

Operators often have dependencies on another operator(s): the "InRange" operator utilizes the "DistanceBetween" operator, as shown in Fig. 3. Furthermore, the "DistanceBetween" operator can have multiple implementations, such as: "DistanceBetweenUsers", "DistanceBetweenCities", "DistanceBetweenPhones", etc. The querying process becomes even more complex for solutions without proper semantics and hierarchical relations.

The following example will show how to find an appropriate operator for the context query: "Find all users in range 500m from Alice". Fig. 3 shows a subset of the operators' ontology, defining two operators: "InRange" and "DistanceBetween". "InRange" has a subclass called "UsersInRange" that takes two inputs "UserID" and "Range", and produces "UserIDBag" as an output. "UserIDBag" is a set of "UserID"s, and represents a property of the class Users.

The reasoning process will find the appropriate operator that satisfies input and output requirements set in the query. The subclass of the oper-

ator "InRange", "UsersInRange", with its method *List<User> getUsersInRange(userID, range)* and its dependency operator "DistanceBetween" having the method *Distance getDistanceBetween(originLocation, targetLocation)* are the result of this reasoning process. An execution of *getUsersInRange* and *getDistanceBetween* will yield a synthesized context, which will be returned to the application and cached in the context repository for the time specified by the "InRange" operator.

4 Related work

Authors of Context Toolkit mention in [4] that an automatic path creation can be adapted to be used for refining and transforming raw sensor data into higher-level context data (which we call context synthesis). This automatic path creation relies on operators, special services that transform data from one form into another (e.g. GPS location data to ZIP code data). Operators could be automatically composed based on high-level needs and on available resources.

The issue of matching application requests for context information and available context parameters has been tackled in [5]. If both sets of parameters are described in terms of ontologies, the matching problem lies in mapping the request parameters from an application-domain ontology to candidate matching context descriptions from the context domain ontology. A solution proposed was to synchronize changes in one ontology with another by replacing the unmatched candidate parameter with a semantically related one that has a match in the application-domain ontology. We mitigate this problem by introducing operators in context queries that specify the context information that needs to be retrieved.

In [3] context queries are performed on a context knowledge base using RDF Data Query Language [6]. The limitation of this approach lies in querying over the existing information stored in the context model using triple (<*subject, predicate, object*>) patterns. Our approach allows operations performed on the context data resulting in new information that previously did not exist in the system. This gives applications greater freedom in forming context queries.

Mobilife project [7] has developed Context Management Framework (CMF) for discovery, exchange, and reasoning on context information. Context information produced by Context Providers is discovered and delivered to Context Consumers using Context Brokers. An appropriate Context Provider is found by semantic matchmaking of required and advertised services. A context engine employs a direct mapping of a Context Provider node and context parameter provided. CMF uses a proxy-based design to manage distributed Context Brokers, where a single Context Broker proxy is the first point of contact to any request for a Context Provider. MIDAS context architecture utilizes the approach of Super Nodes (i.e. nodes having more resources: memory, CPU, etc.) which perform context synthesis and host context mapping component, being discovered by a procedure that finds the closest available Super Node to a Context Consumer. CMF uses rule-based and Bayesian model reasoning, whereas a context engine introduces operators for context synthesis.

5 Conclusion

We have presented a model for retrieving and distributing context information in a mobile distributed environment. Based on the observation that different context information has different update and read patterns, we provide two different mechanisms for context distribution. Context information which is static in its update pattern should be replicated among nodes and context which is volatile should be distributed by remote reads. Example of the former include user profiles which rarely changes. Example of the latter could be a users position.

Moreover, we have introduced the use of typed operators. These operators serve two purposes. First, an operator provides a functional approach to context data simplifying context synthesis and programming of context-aware systems in general. Second, the context engine applies operators dynamically based on description of input and output types. Operators can invoke other simpler operators within their function, based upon the operators' ontology. This results in a system flexibility, extensibility, and enhances code reuse.

We are currently implementing a prototype of a context engine utilizing the proposed mechanisms of context queries, triggers, and operators. Operators will be described using OWL-DL, implemented as Java scripts using BeanShell [8], and their performance will be compared with Semantic Web rule-based reasoning. Privacy concerns and context scope will be added later in the project.

Acknowledgments. This work is part of the EU IST MIDAS project sponsored by European Commission under contract 027055.

References

1. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In Proceedings of the Workshop on Mobile Computing Systems and Applications. IEEE Computer Society, Santa Cruz, CA (1994) 85–90
2. Wang, X.H., Zhang, D.Q., Gu, T., Pung, H.K.: Ontology based context modeling and reasoning using OWL. In Proceedings of Workshop on Context Modeling and Reasoning. Orlando, Florida USA. IEEE (March 2004) 18–22
3. Wang, X., Dong, J.S., Chin, C.Y., Hettiarachchi, S.R.: Semantic Space: An Infrastructure for Smart Spaces. Pervasive computing, July-September Vol. 3., IEEE (2004)
4. Hong, J.I., Landay, J.A.: An Infrastructure Approach to Context-Aware Computing. Human-Computer Interaction. Vol. 16 (2001) 287–303
5. van Kranenburg, H., Bargh, M.S., Iacob, S., Peddemors, A.: A Context Management Framework for Supporting Distributed Context-Aware Applications. Communications Magazine. Vol. 44. IEEE (September 2006) 67–74
6. Seaborne, A.: RDQL - A Query Language for RDF. W3C Member Submission. HP Labs Bristol (January 2004)
7. EU IST-FP6 Mobilife project: http://www.ist-mobilife.org (2006)
8. BeanShell lightweight scripting for Java, http://www.beanshell.org/