UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2569

Provision of Contex Information to CSCW Applications: A Web Service Approach

Renato de Freitas Bulcão Neto
Carlos Odenique Jardim
José Antonio Camacho-Guerrero
Renan Gonçalves Cattelan
Maria da Graça Campos Pimentel

Nº 227

RELATÓRIOS TÉCNICOS

São Carlos – SP
Marc./2004

Provision of Context Information to CSCW Applications: A Web Service Approach

Renato de Freitas Bulcão Neto
Carlos Odenique Jardim
José Antonio Camacho-Guerrero
Renan Gonçalves Cattelan
Maria da Graça Campos Pimentel

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
Av. do Trabalhador São-Carlense, 400 - Centro
Caixa Postal 668
CEP 13560-970 São Carlos, SP

{rbulcao, patrao, jcamacho, renan, mgp}@icmc.usp.br

Resumo. *To tackle the demand for support to building CSCW applications, one approach has been the provision of reusable components and infrastructures that deal with factors such as synchronization, communication and access control. In particular, regarding the CSCW applications' necessity of using context information to adapt their services following users' and groups' needs, the literature has reported the provision of models and associated implementations that allow applications to specify, share and report relevant context information. We propose that context-aware CSCW applications can take advantage of the benefits provided by Web Services, alleviating differences in heterogeneous environments and allowing the interchange of information between applications and infrastructures. We present the Context Kernel Web Service, which models classic dimensions for context information. We also show how typical operations demanded by CSCW applications can use the Context Kernel Web Service to store, retrieve and exchange context information via the Web.*

Keywords: CSCW, Context-awareness, Web Services, Context Kernel.

Contents

List of Figures

1. Introduction

Context-aware computing focuses on the ability of a computational entity to adapt its behavior based on context information sensed both from the physical and computational environment. Context has been defined as "any relevant information about the user-application interaction, including the user and the application themselves" [Dey, 2001]. This definition can be extended so as to observe that the context information is, in essence, dynamic [Greenberg, 2001] — a most relevant requirement to CSCW applications.

Many non-CSCW context-aware applications, such as the pioneers *PARC-Tab* [Schilit et al., 1993] and *Active Badge* [Want et al., 1992], and more recent ones, such as the *CampusAware* [Burrell et al., 2002], implement their own mechanisms for capturing, storing and processing context information. The literature reports on efforts geared towards providing support for the construction of services dedicated to capture, store and process context information. Examples include the *Context Fabric* [Hong and Landay, 2001] service infrastructure, the *Aura* [Garlan et al., 2002] architectural framework and the *GaiaOS* [Hess et al., 2002] meta-operating system.

The literature reports on exploiting the Web as platform for providing CSCW functionalities. One example is the work by Girgensohn and Lee, who examined complementary group awareness related services demanded by users depending on their task [Girgensohn and Lee, 2002]. Regarding context information relative to visitors to Web sites, Gellersen and Schmidt reported on requirements for providing awareness of visitors to one's Web site as well as introduced the idea of gathering information from the site of origin of the visitor [Gellersen and Schmidt, 2002].

CSCW applications depend on the availability of information with respect to individuals and the groups that they belong to (group awareness) in order to provide the most basic capabilities for social awareness, which "includes information about the presence and the activities of people in a shared environment" [Prinz, 1999]. Ideally, such applications should also exploit context information to adapt their services according to users' and groups' needs: even if one user belongs to a group (e.g. a instructor and her pupils), it is not often that activities of a group are the focus of attention of a user (e.g. the instructor is in a conference with her supervisor).

The need for infrastructures to support building CSCW applications has been long discussed. The main approach has been the building of components which implement functions that can be used in many scenarios: the idea is to facilitate that a new application be built by reusing the components that implement the desired features. In this case, the components are designed so as to support that applications run in a distributed environment, and infrastructures deal with synchronization, flow control, object sharing and other pertinent features [Roseman and Greenberg, 1996] [Chabert et al., 1998]. As a result, the application is composed of distributed components that share the same information, privileges and constraints — this means that the distributed users make synchronous or asynchronous use of the same application. In some cases, the components

are associated with architectural models so as to facilitate the design as well as the evolution of applications, such as the case of Dragonfly [Anderson et al., 2000] and Clover [Laurillau and Nigay, 2002].

The literature also discusses the need for integrating heterogeneous environments. Li and Li discuss the demand for providing support to the collaborative use of different instances of an application with the same functionality — for instance, the same text editor running in different operating systems, or even different editors that have a common set of functions [Li and Li, 2002]. Keith Edwards et al. have demonstrated the use of a framework for recombinant computing to support building applications providing impromptu peer-to-peer collaboration to independent users and their own working environments and resources [Edwards et al., 2002] .

Efforts have also been directed towards providing CSCW applications with generic infrastructures handling event notification [Prinz, 1999] [Fitzpatrick et al., 1999] [Patterson and Kucan, 1996], and specifically sharing context information [Gross and Prinz, 2003] [Rittenbruch, 1999] [Fuchs, 1999]. The main focus is on supporting building integrated CSCW applications — such as the use of context information by shared workspaces [Gross and Prinz, 2003].

However, there is an interesting opportunity for independent CSCW applications to share context information regarding social awareness, for instance in a repository of group-related context information. The advantage is twofold. First, by consulting the repository to get context information, the application does not have to gather and maintain that information by itself. Second, applications can interoperate seamlessly just by using the same information.

Challenges faced by developers of context-aware applications, CSCW or not, include the support for several levels of heterogeneity and the distribution of responsibilities between applications and infrastructures. An alternative to deal with these two issues is to take advantage of the benefits provided by Web Services.

Web Services have become a key success factor as infrastructure for several development and integration projects [Arsanjani et al., 2003]. The essence of Web Services [W3C, 2002] is the use of the Internet infrastructure to bridge a myriad of Internet systems transparently and independently of differences in network technologies, devices, operating systems and programming languages. Web Services are largely based on HTTP as the application-level protocol, and open XML-based specifications, such as XSD (*XML Schema Definition Language*) [W3C, 2001], WSDL (*Web Service Description Language*) [W3C, 2003b] and SOAP (*Simple Object Access Protocol*) [W3C, 2003a].

Context-aware CSCW applications can take advantage of the benefits provided by Web Services, alleviating differences in heterogeneous environments and allowing the interchange of information between applications and infrastructures. We present the Context Kernel Web Service, which models classic dimensions for context information such as *who*, *where*, *when* and *what* [Arruda Jr et al., 2003]. We also show how typical compo-

nents demanded by CSCW applications can use the Context Kernel Web Service to store, retrieve and exchange context information.

Section 2 presents current research in context-aware ubiquitous computing in general, and in CSCW literature, in particular. Section 3 introduces the Context Kernel Web Service. Section 4 illustrates the use of the Context Kernel by typical CSCW components and applications. Section 5 discusses some limitations of the Context Kernel implementation, as well as the concluding remarks and future work.

2. Infrastructures for context-awareness

Making applications context-aware is one of the challenges pointed out in the ubiquitous computing literature [Abowd et al., 2002], which reports results on mobile and context-aware systems, such as the pioneers *Xerox PARCTab* [Schilit et al., 1993] and *Olivetti Research Lab's Active Badge* [Want et al., 1992], as well as in toolkits, frameworks and service infrastructures.

The *Context Toolkit* [Dey et al., 1999a] offers a framework with generic services and abstractions — context widgets, interpreters and aggregators — in order to overcome the lack of support for a standard development of sensor-based context-aware applications. The history of each type of context information is stored on a relational database by its corresponding context widget for providing any application interested with that valuable information. For instance, a particular context widget has been used to sense the presence of a user and be able to identify him/her in a whiteboard capture application for impromptu meetings [Brotherton et al., 1999]. In essence, the *Context Toolkit* provides applications with capture, storage, conversion, aggregation, access and distribution of context information. The *Context Toolkit* has been used to build CSCW applications including the *Conference Assistant* [Dey et al., 1999b] and the *Family Intercom* [Nagel et al., 2001].

The *Context Fabric* [Hong and Landay, 2001] is a service infrastructure for context-aware applications focusing on a storage model for flexible and distributed context information. Similarly to the *Context Toolkit*, its infrastructure is responsible for the storage of context information. However, the *Context Fabric* differs from *Context Toolkit* in two significant ways [Hong, 2002]. First, *Context Fabric* takes the architectural model in the *Context Toolkit* and generalizes it to two services, an event service and a query service, handling events and queries as main abstractions. Second, the *Context Fabric* separates the specification of context needs from its processing. This allows applications to take changes in the environment into account when processing context needs, such as when someone moves from one place to another one. The handling of events and sensors has been implemented in the *NESSIE* generic infrastructure for contextual event notifications [Prinz, 1999], while the modelling of context information so as to allow the mapping of incoming events to a context of origin is the core of the proposals by Gross and Prinz [Gross and Prinz, 2003], Fuchs [Fuchs, 1999] and Rittenburch [Rittenbruch, 1999].

The *GaiaOS* [Hess et al., 2002] is a component-based middleware designed to provide support for a *User Virtual Space*, an abstraction that comprises context information, tasks and devices associated to each user. A user virtual space is characterized as a proactive entity in a system software, because it must reconfigure itself while its components — the current location or the device being used — change. By using the *GaiaOS* architecture, the *ConChat*, a context-aware chat application, allows users to query and exchange their chat partner's context information such as location, proximity of other users and activity. Such typical CSCW application uses context information to improve electronic communication, since it conveys a better understanding of the social context of each user on the chat. For example, if a user knows that the other person is involved in some urgent activity such as a meeting, he expects a slower response [Ranganathan et al., 2002].

The *Aura* project [Garlan et al., 2002] provides an architectural framework for user mobility in ubiquitous computing environments that allows the adaptation to users context and needs. Its main component includes a representation of users' tasks as a collection of service descriptions which can be mapped on *Service Suppliers* currently available from the environment. The *Task Manager* is responsible for the allocation of tasks to available resources; the *Environment Manager* is aware of which service suppliers are currently available from the environment; finally, the *Context Observers* provide the environment manager with information from the physical context [Cheng et al., 2002]. Some CSCW applications have been implemented to exploit the *Aura* architecture such as the *Portable Help Desk* [Salber et al., 2001], which allows a mobile student to locate his colleagues and find useful resources (e.g. printers, restaurants and vending machines). *Portable Help Desk* relies on a highly accurate location service, which enables students to instantaneously locate one another in order to manage their team meeting schedules and projects.

Teamspace [Fuchs et al., 2001] defines a framework that target both synchronous and asynchronous collaboration — aiming at supporting team coordination in general, instead of specific tasks. The underlying model is defined upon concepts that include task structured workspace (by means of persistent articulation objects), place-based adaptation to work modes (using persistent places to facilitate transitions between modes), and synchronous and asynchronous communication and awareness (by means of a low bandwidth multimedia tool providing instant message and audio and video conference services). The task structured workspace is organized according to three activity categories: work-related, meeting-related and people-related so as to support task awareness relative to the activities of the team.

Several efforts have been directed towards providing CSCW applications with generic infrastructures handling event notification to provide task awareness, such as *NESSIE* [Prinz, 1999], *Notification Service Transfer Protocol* (NSTP) [Patterson and Kucan, 1996] and *Elvin* [Fitzpatrick et al., 1999]. More specifically regarding the modelling of context information, the literature includes work by Gross and Prinz [Gross and Prinz, 2003], Fuchs [Fuchs, 1999] and Ritten-

bruch [Rittenbruch, 1999]. The models are focused on supporting synchronous interaction in CSCW systems, which brings important constraints in terms of response time as well as the homogeneity of the applications. *AREA* [Fuchs, 1999], however, focuses on modelling both synchronous and asynchronous interaction. In some cases, those infrastructures systems leave to applications (or to the users) the responsibility of providing low-level information concerning which (awareness or context) information is desired at each time. As a result, models formalize elaborate specifications that must be provided by applications — or the user — so that the associated rules can be computed by the corresponding infrastructures. Such effort corresponds to the tradeoff of having a generic service manipulating the context information.

The user is responsible for providing heterogeneous spaces in applications built in a peer-to-peer basis with the infrastructure proposed by Keith Edwards et al.: in this case no explicit model of context information is given by the user or the application, but user actions are mapped to the system's underlying model of impromptu collaboration so as to give access both to common and complementary resources [Edwards et al., 2002].

3. The Context Kernel Web Service

The essence of Web Services [Stal, 2002] is the use of the Internet infrastructure to allow applications to communicate seamlessly and independently of heterogeneous hardware and software. Web Services are accessed using standard Internet protocols as HTTP operating on top of TCP, and can be defined by self-describing messages referencing information to understand the message. Open XML-based specifications — XSD, WSDL and SOAP — are the building blocks of the basic Web Services architecture [Burner, 2003].

The XML language is used for representing the data inside messages using element and attribute names. The XML Schema specification (XSD) provides a common collection of data types for describing XML attributes and elements in order to achieve a syntactic level of interoperability across the Internet.

For the exchange of messages, Web Services may use the SOAP protocol, which has two main characteristics: simplicity and extensibility. The SOAP protocol defines the XML-based syntax, the semantics and the order of messages exchanged between peers (applications or services). SOAP messages include an optional Header element and a mandatory Body element, all wrapped by an Envelop element.

Web Services may take advantage of the WSDL specification to define the service contract, i.e., the collection of messages which a service accepts and produces. Moreover, WSDL describes the mapping between abstract messages to particular objects or methods to be used by applications requesting the services [Fremantle et al., 2002]. The WSDL language defines:

- *types* as optional elements used to build messages;
- the request and response *messages*;

- *portTypes* to map messages to abstract operations;
- *bindings* to map *portTypes* to concrete protocols;
- *ports* to define the real communication addresses of services;
- and *services* as collections of ports.

We advocate that infrastructures aimed at supporting context-aware CSCW applications can take advantage of the benefits leveraged by Web Services. In particular, infrastructures would allow applications to exchange data with communication efficiency, loose coupling and asynchrony [McKusick, 2003]. This means that applications built upon Web Services-based infrastructures would not only store and retrieve, but also exchange information through the Web.

In the next subsections, we discuss how the Web Services approach can be used to allow the storage, retrieval and exchange of context information.

3.1. Representation schema of context information

The Context Kernel is a Web Service that allows applications to handle context information based on the classic dimensions *who*, *where*, *when*, *what*, *why* and *how* discussed in the ubiquitous computing literature (e.g. [Abowd et al., 2002] and [Truong et al., 2001]) by formalizing a set of XML-based operations associated to those dimensions. The Context Kernel Web Service classifies those dimensions as follows [Arruda Jr et al., 2003]:

- *primitive dimensions*, i.e., those handled independently of other dimensions;
- *derivative dimensions*, i.e., those obtained by relating other dimensions — primitive or derivative.

The Context Kernel stores *primitive dimensions* by means of a premise defined by a tuple containing *type, value* and an optional *qualifier*. The following example represents an instance of the primitive dimension *who*:

```
<premise dimension="who" type="login" value="mgp" qualifier=""/>
```

A *derivative dimension* is defined by means of a *rule* that contains at least one *premise* and one *inference*: a *premise* is defined as above and *rules* are grouped in *schemas*. A set of related schemas is then contained in an element *context*. Following that classification, any dimension can be primitive or derivative, being strictly dependent on the application requirements. Therefore, the applications themselves are responsible for the specification of which kind of data and rules are particularly relevant to them, as it is the case with the other infrastructures reported in the literature.

The following XML excerpt (Example 1) illustrates the vocabulary defined by Context Kernel: a set of premisses relative to the *who* dimension gives details relative to a user (login, name, password and email) and defines, by means of the *what* dimension, that the user is a member of the group *soccer*.

```
<!--            Example 1            -->
<context>
  <schema>
    <rule>
      <premiseS>
        <premise dimension="who" type="login" value="cjardim"/>
        <premise dimension="who" type="name" value="Carlos Jardim"/>
        <premise dimension="who" type="password" value="********"/>
        <premise dimension="who" type="email" value="cjardim@icmc.usp.br"/>
      </premiseS>
      <inferenceS>
        <inference dimension="what" type="group" value="soccer"/>
      </inferenceS>
    </rule>
  </schema>
</context>
```

3.2. Context Kernel architecture

As a typical Web Service, the Context Kernel architecture can be thought as having five distinct layers. In the top, the *application* layer (i) corresponds to applications making use of the Context Kernel service by invoking SOAP messages. The next two layers refer to the *contract* layer (ii) using WSDL for the service description, and the *protocol* layer (iii) using SOAP on top of HTTP, respectively. The *functional service* layer (iv) describes the service interface, i.e., the main functions provided by the Context Kernel service, such as registering and getting status of applications as well as storing and retrieving context information.

The lower *implementation* layer (v) refers to the logic implementation of the Context Kernel, which adopts a document-oriented API implementation [Burner, 2003], abstracting the system architectures and creating a loosely-coupled connectedness that withstands changes to its underlying implementation.

3.3. Using the Context Kernel

The Context Kernel publishes its WSDL document definition that contains the description of the service so that it is available to interested applications. For the case that a specific group of applications wants to make exclusive use of the service — for security or privacy reasons, for instance, or just for relevance relative to the applications domain — this publication can be carried out in restrict forms.

Once the context-aware application designer retrieves the WSDL document, he/she obtains the service contract and the network endpoints that honor this contract. In the Context Kernel environment, endpoints are Java servlets used to extend the capabilities of servers in a *request-response* programming model.

Once knowing the WSDL specification, the designer builds an application that uses the Context Kernel Web Service as follows:

- Based on the service contract, applications send HTTP messages (POST or GET) using the SOAP packaging protocol for requesting the available services;

- The Context Kernel stores/retrieves context information provided by applications on/from a repository. The result of the processing of context information is returned to the requesting application via SOAP messages.

The next section details the API implemented by the Context Kernel. As far as the exchange of context information among loosely coupled applications is concerned, an application:

- first, obtains the public identifiers of other applications registered with the Context Kernel (details in *section 3.4.3.. Checking the Applications Registered with the Context Kernel*), and
- next, retrieves the set of rules stored by those applications using the GetRules service (details in *section 3.4.5.. Retrieving Information from the Context Kernel*).

That mechanism allows application designers to be able to write applications that share context information.

3.4. Context Kernel API

The software platform used in the implementation of the Context Kernel includes the Linux operating system, the Apache web server extended with the Tomcat servlet container, the Java programming language, tools to handle W3C document specifications (XML, XSD, WSDL and SOAP) and the Postgres database.

The Context Kernel API offers four categories of services: *registry*, *status*, *storage* and *retrieval*. Given the Web Services approach, all those services are invoked by applications using the SOAP protocol encapsulating XML-based messages.

3.4.1. Registering with the Context Kernel

In order to use the Context Kernel, the first step for an application is to register its own information, which is to be made available to other applications.

An application invokes the InfoApp service to *register* its metadata, such as *name*, *description* and *developers*, and receives back one public and one private identifier. The private identifier is used to store context information; the public identifier is used by third-party applications to access the context information in a read-only basis. Once an application A holds the public identifier of an application B, the application A can use the other services of the Context Kernel API to query the Context Kernel to obtain information stored by application B.

In the following XML excerpt (Example 2), the *WebMemex* System (detailed later) registers its metadata with the Context Kernel using InfoApp.

```
<!--            Example 2            -->
<env:Body>
  <ck:InfoAppRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/InfoAppRequest.xsd">
    <application>
```

```
      <name>WebMemex</name>
      <description>Web-based Recommender System</description>
      <url>http://linkserver.icmc.usp.br/webmemex</url>
      <lastRelease>2003-12-01</lastRelease>
      <version>1.0</version>
      <status>academic</status>
      <developers>
        <author email="jcamacho@icmc.usp.br">Jose Antonio</author>
        <author email="khai@cc.gatech.edu">Khai Truong</author>
      </developers>
    </application>
  </ck:InfoAppRequest>
</env:Body>
```

3.4.2. The Context Daemon Notification Service

An important limitation resulting from using the Web Services approach is lack of support to *notification services* — a most important requirement to CSCW toolkits and generic infrastructures. To overcome this problem, the Context Kernel includes a notification service called *Context Daemon*, which works as follows.

Any application demanding event notification must specify, during registration with the CK using the InfoApp service, the callback address by means of which the Context Deamon will be able to send results when they become available. As an example, the following excerpt would be inserted after the `developers` element in the code given in Example 2.

```
<callback_address>
  <host>mexcal.intermidia.icmc.usp.br</host>
  <port>9321</port>
</callback_address>
```

As a result, the Context Kernel will try to validate all rules stored by that application on the repository (using PutRules service detailed later). This means that, when any information is stored by the application on the repository (using PutData service detailed below), the Context Kernel will check if any of its rules have been met. Although this is quite a consuming approach, it may be necessary for some applications. An alternative syntax and associated code is being built so that an application will be able to specify which rules should be monitored and when they should be monitored.

3.4.3. Checking the Applications Registered with the Context Kernel

An application can retrieve *status* information stored by another application. This can be achieved by invoking the StatusApps service which returns the public identifiers (`<publicID>`) and the metadata of *all* applications registered with the server. Until the time of this writing, the Context Kernel does not implement a discovery service.

In the following excerpt (Example 3), an application invokes the StatusApps service (no identifier is necessary) and receives back the metadata of two applications already registered with the Context Kernel: the *WebMemex* and the *WebChat* (both detailed

later) applications. The element <toCK> stores the particular date and time in which an application has registered with the Context Kernel.

```
<!--            Example 3            -->
<env:Body>
  <ck:StatusAppsResponse
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/StatusAppsResponse.xsd">
    <applications>
      <application>
        <name>WebMemex</name>
        <description>Web-based Recommender System</description>
        <url>http://linkserver.icmc.usp.br/webmemex</url>
        <lastRelease>2003-12-01</lastRelease>
        <version>1.0</version>
        <status>academic</status>
        <publicID>CSagsqFg07bmP4NBgwd6mUPOhyGI5e</publicID>
        <toCK>2004-01-18T14:36:20</toCK>
        <developers>
          <author email="jcamacho@icmc.usp.br">Jose Antonio</author>
          <author email="khai@cc.gatech.edu">Khai Truong</author>
        </developers>
      </application>
      <application>
        <name>WebChat</name>
        <description>Web-based Chat System using XML technologies</description>
        <url>http://iclass.icmc.usp.br/webchat</url>
        <lastRelease>2003-10-12</lastRelease>
        <version>2.0</version>
        <status>academic</status>
        <publicID>raBG6ADnr0Edix5EyvMnJUDhp</publicID>
        <toCK>2004-01-184T15:06:13</toCK>
        <developers>
          <author email="renan@icmc.usp.br">Renan Cattelan</author>
        </developers>
      </application>
    </applications>
  </ck:StatusAppsResponse>
</env:Body>
```

Once an application obtains the public identifiers of other applications registered with the Context Kernel, that application can retrieve the set of rules stored by them via the GetRules service (as detailed in *section 3.4.5.. Retrieving Information from the Context Kernel*) — which is the mechanism that allows applications to share and exchange information.

3.4.4. Storing Information in the Context Kernel

The Context Kernel provides services which allow applications to *store* and *retrieve* context information relative to *schemas*, *rules*, *premises* and *inferences*, as demanded by the code shown in Example 1. It is worth noting that Context Kernel relies on context-aware applications regarding the validity of the data and rules being stored. Moreover, the relevance of each piece of context information or rule of context is prerogative of the applications themselves.

The PutData service allows applications to *store* context information relative to *primitive dimensions*; in this case, the element <*context*> includes primitives. In the

following excerpt (Example 4), a sensor-based application requests the PutData service to store that "a *person* whose username is *jcamacho* logged in to the *sap02* machine at the *lab3 room* on *December 4th* at *10:32:04 AM*".

```
<!--            Example 4            -->
<env:Body>
  <ck:PutDataRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/PutDataRequest.xsd">
    <privateID>0LT5ia5Ejn5CWB1ii17YkEzBFTmaVq</privateID>
    <context>
      <primitive>
        <whoS>
          <who type="login" value="camacho" />
        </whoS>
        <whereS>
          <where type="machine" value="sap02" />
          <where type="room" value="lab3" />
        </whereS>
        <whenS>
          <when type="date" value="2003-12-04T10:32:04Z" qualifier="datetime" />
        </whenS>
      </primitive>
    </context>
  </ck:PutDataRequest>
</env:Body>
```

The PutRules service allows the storage of context information relative both to *primitive* and *derivative dimensions*. Example 5 shows an application requesting the storage of the rule "a *person* called *Carlos Jardim* is at the *lab4 room* every *Friday* morning because he attends a *software engineering class*". Applications may associate an expiration date for their rules; in this case, the rule is valid only for the *first semester year 2004*. When the validation date of a rule expires, it will continue stored on the server, but with solely historical purpose, not being used for querying any more.

```
<!--            Example 5            -->
<env:Body>
  <ck:PutRulesRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/PutRulesRequest.xsd">
    <privateID>BTbhtrGhP8anQ5MChve7nVQPizHJ6f</privateID>
    <context>
      <schema>
        <rule begin="2004-01-01" end="2004-07-01">
          <premiseS>
            <premise dimension="who" type="name" value="Carlos Jardim"/>
            <premise dimension="where" type="room" value="lab4"/>
            <premise dimension="when" type="weekday" value="Friday"/>
          </premiseS>
          <inferenceS>
            <inference dimension="what" type="activity"
                       value="software engineering class"/>
          </inferenceS>
        </rule>
      </schema>
    </context>
  </ck:PutRulesRequest>
</env:Body>
```

3.4.5. Retrieving Information from the Context Kernel

The Context Kernel service makes available the services GetRules, GetAny and GetInverse for querying context information.

The GetRules service allows applications to retrieve the rules associated to context information stored by other applications based on the public identifier of that application. This allows that one application be able to obtain the formal specification of the information stored by other applications — which is necessary to applications designers to understand the semantic of the rules and to specify appropriate queries. As a result, applications designers are able to write applications that share context information.

Example 6 illustrates a response message for the GetRules service. All rules stored by a specific application are returned, but the attributes "value" are empty.

```
<!--            Example 6            -->
<env:Body>
  <ck:GetRulesResponse
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/GetRulesResponse.xsd">
    <schema>
      <rule>
        <premiseS>
          <premise dimension="who" type="login" value="" qualifier="" />
          <premise dimension="who" type="name" value="" qualifier="" />
          <premise dimension="who" type="password" value="" qualifier="" />
          <premise dimension="who" type="email" value="" qualifier="" />
        </premiseS>
        <inferenceS>
          <inference dimension="what" type="" />
        </inferenceS>
      </rule>
      <rule>
        <premiseS>
          <premise dimension="who" type="login" value="" qualifier="" />
        </premiseS>
        <inferenceS>
          <inference dimension="what" type="group" />
        </inferenceS>
      </rule>
    </schema>
  </ck:GetRulesResponse>
</env:Body>
```

The GetAny service allows applications to retrieve context information based on the value of *premises* or *inferences* associated to *primitive* and *derivative dimensions*. Moreover, it is possible to specify responses based on the dimensions themselves. Applications may also specify the maximum number of answers or even combine premises using boolean operators as illustrated in Example 7: a sensor-based application requests "the three most recent (*last value=3*) activities (*dimension=what*) which a *person* called *Renan Cattelan* carried out at the *lab4* room on *June 4th, 2003*".

```
<!--            Example 7            -->
<env:Body>
  <ck:GetAnyRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/GetAnyRequest.xsd">
    <publicID>h4YNEXzZURxesfJnljRVgvTpx</publicID>
    <last value="3"/>
```

```
    <context>
      <premiseS>
        <boolean type="AND">
          <premise dimension="who" type="name" value="Renan Cattelan"/>
          <premise dimension="where" type="room" value="lab4"/>
          <premise dimension="when" type="date" value="2003-06-04"/>
        </boolean>
      </premiseS>
      <inferenceS>
          <inference dimension="what" type="activity"/>
      </inferenceS>
    </context>
  </ck:GetAnyRequest>
</env:Body>
```

The GetInverse service allows applications to retrieve context information using the inference to obtain the corresponding premises. Moreover, it is also possible to specify responses based on the dimensions themselves. This kind of context interpretation follows the semantics of a backward reasoning, while the GetAny service can be thought as a forward reasoning. Example 8 shows an application requesting "where can someone called *Renato* be found?".

```
<!--            Example 8            -->
<env:Body>
<ck:GetInverseRequest
    xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/GetInverseRequest.xsd">
    <publicID>h4YNEXzZURxesfJnljRVgvTpx</publicID>
    <context>
      <inferenceS>
        <inference dimension="where" type="room"/>
      </inferenceS>
      <premiseS>
        <premise dimension="who" type="name" value="Renato"/>
      </premiseS>
    </context>
  </ck:GetInverseRequest>
</env:Body>
```

In the next section, we illustrate the use of the Context Kernel Web Service by representative CSCW applications.

4. Context Kernel in use by CSCW applications

This section illustrates how loosely-coupled CSCW applications can take advantage of the benefits leveraged by Context Kernel. We have built a set of applications that exploit users and groups information stored by one of them on the Context Kernel. For each application we briefly present it and detail its use of the Context Kernel.

4.1. The WebLogin application

The *WebLogin* (Figure 1(a)) is an application used as the entry point for other applications in our CSCW environment. The first thing a user does is to log in to that application; since the application itself is a component that can be activated by any other application, the typical scenario is that the user will be to log in after having started some other application.

As it is the case with any application, before using the Context Kernel (from now also called CK), *WebLogin* needs to register itself with the service: *WebLogin* calls the InfoApp service and obtains its public and private identifiers. From this point, as with any other application, *WebLogin* must provide one of those identifiers when communicating with the CK. In the following XML excerpt (Example 9), the *WebLogin* application registers its metadata with the CK by means of the InfoApp service.

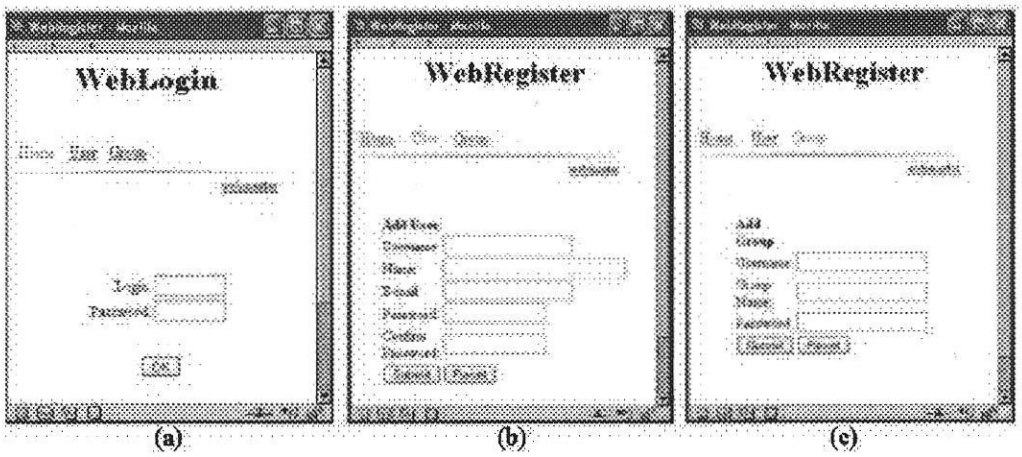

Figure 1: (a) The login user information will be verified in the Context Kernel; (b) the new user provides information to be registered with the Context Kernel; (c) the interface for registering groups with the Context Kernel.

```
<!--            Example 9            -->
<env:Body>
  <ck:InfoAppRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/InfoAppRequest.xsd">
    <application>
      <name>WebLogin</name>
      <description>Web-based Logging System</description>
      <url>http://linkserver.icmc.usp.br/weblogin</url>
      <lastRelease>2004-02-19</lastRelease>
      <version>1.0</version>
      <status>academic</status>
      <developers>
        <author email="jcamacho@icmc.usp.br">Jose Antonio</author>
        <author email="rbulcao@icmc.usp.br">Renato Bulcao</author>
      </developers>
    </application>
  </ck:InfoAppRequest>
</env:Body>
```

When a user tries to log in to the *WebLogin* application, it is necessary to check if the current user has been already registered with the CK database. In order to share information between applications, it is necessary to obtain their public identifiers. In this case, user and group information is stored by the *WebRegister* application (detailed next). Therefore, *WebLogin* needs to call the StatusApps service to get the public identifier of the *WebRegister* application.

Similarly, all applications intending to query user and group information must include the public identifier of the *WebRegister* application. Moreover, in order to know the format of the rules stored by a specific application, in this case *WebRegister* that handles user and group information, all applications need to call the GetRules service as discussed earlier.

Due to design reasons, every user registered with CK has a group with her own name since a user can work by herself as well as in groups. Group information is stored as a *what* inference while user information is handled as WHO premises. By using the method User.exist() of the CK client API, the *WebLogin* application invokes the GetAny service that searches for some rule which includes a *what* inference where type has value "group" and the premises include the current username and the corresponding password. The following excerpt (Example 10) describes a user logging in to the *WebLogin* application. During registration, every user is associated with his own group, as discussed next.

```
<!--            Example 10            -->
<env:Body>
  <ck:GetAnyRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/GetAnyRequest.xsd">
    <publicID>h4YNEXzZURxesfJn1jRVgvTpx</publicID>
    <last value="1"/>
    <context>
      <premiseS>
        <boolean type="AND">
          <premise dimension="who" type="login" value="cjardim"/>
          <premise dimension="who" type="password" value="********"/>
        </boolean>
      </premiseS>
      <inferenceS>
        <inference dimension="what" type="group"/>
      </inferenceS>
    </context>
  </ck:GetAnyRequest>
</env:Body>
```

4.2. The WebRegister application

According to the example above, the *WebLogin* application receives a status message indicating whether a specific user exists or not. If she is a new user, *WebRegister* — an application designed for the management of users and groups (Figure 1(b)) — is started by *WebLogin*.

As with any other application, *WebRegister* was first registered with CK via InfoApp service. In regular use, *WebRegister* must provide its own private identifier (since it will execute an update) and information regarding the new user as input parameters for invoking the PutRules service. The premises corresponding to the new user are then registered with the CK database as well as the inference corresponding to his own group as in Example 11.

```
<!--            Example 11            -->
<env:Body>
  <ck:PutRulesRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/PutRulesRequest.xsd">
    <privateID>0LT5ia5Ejn5CWBliil7YkEzBFTmaVq</privateID>
```

```
      <context>
        <schema>
          <rule>
            <premiseS>
              <premise dimension="who" type="login" value="cjardim"/>
              <premise dimension="who" type="name" value="Carlos Jardim"/>
              <premise dimension="who" type="password" value="********"/>
              <premise dimension="who" type="email" value="cjardim@icmc.usp.br"/>
            </premiseS>
            <inferenceS>
              <inference dimension="what" type="group" value="cjardim"/>
            </inferenceS>
          </rule>
        </schema>
      </context>
    </ck:PutRulesRequest>
  </env:Body>
```

If a user wants to create a group (Figure 1(c)) — or else be included in an already existing group — *WebRegister* needs to check both if the current user and the current group have been already registered with the CK database.

Once the method User.exist() succeeds, *WebRegister* calls the method Group.existGroup() which, in turn, calls the GetInverse service so that CK checks the existence of some *what* inference where the type is "group" and value is equal to the name of the group given as input parameter (e.g. "intermedia"). The following XML excerpt (Example 12) illustrates that situation.

```
<!--            Example 12            -->
<env:Body>
  <ck:GetInverseRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/GetInverseRequest.xsd">
    <publicID>h4YNEXzZURxesfJn1jRVgvTpx</publicID>
    <context>
      <inferenceS>
        <inference dimension="what" type="group" value="intermedia"/>
      </inferenceS>
    </context>
  </ck:GetInverseRequest>
</env:Body>
```

If the group does not exist, *WebRegister* calls the PutRules service with the following input parameters: its private identifier, a triple corresponding to the premise and the *what* inference for groups. For instance, the premise (dimension = "who", type = "login", value = "cjardim") and the inference (dimension = "what" type = "group" value = "intermedia") — as detailed in the following XML excerpt (Example 13).

```
<!--            Example 13            -->
<env:Body>
  <ck:PutRulesRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/PutRulesRequest.xsd">
    <privateID>0LT5ia5Ejn5CWB1ii17YkEzBFTmaVq</privateID>
    <context>
      <schema>
        <rule>
          <premiseS>
            <premise dimension="who" type="login" value="cjardim"/>
          </premiseS>
          <inferenceS>
```

```
            <inference dimension="what" type="group" value="intermedia"/>
          </inferenceS>
        </rule>
      </schema>
    </context>
  </ck:PutRulesRequest>
</env:Body>
```

Therefore, the username whose value is "cjardim" belongs to two groups: his own group and the "intermedia" group. As far as the applications illustrated in this section are concerned, group and user information is the core of the communication between the Context Kernel and integrated applications such as *WebMemex* and *WebChat* described next.

4.3. The WebMemex application

Several web-based recommendation systems have been proposed in the literature, the essence in many cases being that members of a group are well-suited to recommend each other appropriate references. Ribak et al. have built and evaluated ReachOut [Ribak et al., 2002], a system that support users in interested groups to explicitly ask recommendation to other group members. In another original approach, McNee et al. use the underlying network of referencing in research papers to support building a system to recommend citation of research papers — in this case the group is formed by the corresponding set of authors of the papers [McNee et al., 2002] .

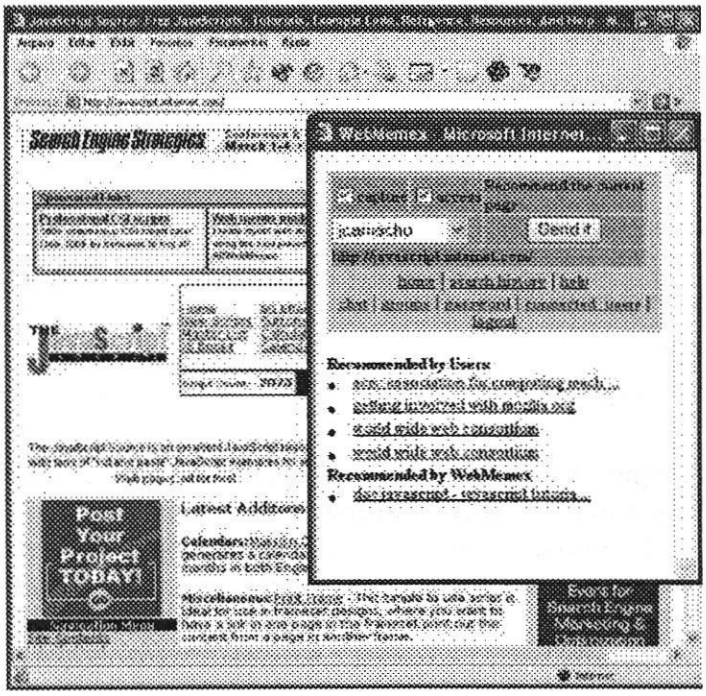

Figure 2: The WebMemex interface for the user *jcamacho* (foreground).

WebMemex is an application that recommends web pages related to what the user is currently visiting. It was developed on top of a high level-architecture that provides user authentication, capture, linking, storage, retrieval and access capabilities [Macedo et al., 2003]. The *WebMemex* application captures and recommends web pages for groups of users by means of its own web proxy server. In order to identify users and group memberships, the application relies on the fact that the web browser client must establish a connection to that web proxy server for each HTTP request. The interface of *WebMemex* is depicted in Figure 2.

When a user tries to log in to the *WebMemex* by means of the *WebLogin* application, it is necessary to check if the current user has been already registered with the CK database.

WebMemex can be made aware of users as they become online by registering its demand in the *Context Daemon* notification service: the requirement is that a callback address has been specified when registering with the CK by means of InfoApp service.

When *WebMemex* has registered using InfoApp, it has informed its callback address. This means that the Context Kernel will try to validate all the rules *WebMemex* has registered every time there is an update of related information in the CK. Example 14 illustrates that *WebMemex* needs to be notified when the user "cjardim" becomes active after having logged in to *WebLogin*.

```
<!--            Example 14            -->
<env:Body>
  <ck:PutRulesRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/PutRulesRequest.xsd">
    <privateID>RNAIIbTfjDCwwLyo3weq11AZPfW2D2</privateID>
    <context>
      <schema>
        <rule>
          <premiseS>
            <premise dimension="who" type="login" value="cjardim"/>
          </premiseS>
          <inferenceS>
            <inference dimension="what" type="weblogin" value="active"/>
          </inferenceS>
        </rule>
      </schema>
    </context>
  </ck:PutRulesRequest>
</env:Body>
```

When a user registers with *WebLogin*, that application invokes the PutData service to store that user "cjardim" is active (excerpt in Example 15). The *Context Daemon* validates this context information against the rules stored on Context Kernel. Thus, the applications that registered rules (in this case, *WebMemex*) will be notified that the user (has just become) is active.

```
<!--            Example 15            -->
<env:Body>
  <ck:PutDataRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/PutDataRequest.xsd">
    <privateID>ER1XtZj0byeDNsmb5uNLHoXAXq</privateID>
```

```
      <context>
        <primitive>
          <whoS>
            <who type="login" value="cjardim" />
          </whoS>
          <whatS>
            <what type="weblogin" value="active" />
          </whatS>
        </primitive>
      </context>
    </ck:PutDataRequest>
  </env:Body>
```

When using *WebMemex*, a user needs to be aware of the set of groups that she belongs to (so she can change groups, for instance). Knowing who the current user is, *WebMemex* requests the method User.getGroups(), that calls the GetAny service which, in turn, searches for all (last value = "ALL") *what* inferences with type "group" and a premise with information that matches the current username. In this case, the triple (dimension = "who", type = "login", value = "cjardim") represents that premise, while (dimension = "what" type = "group") describes the group. As a design decision, the result of this query does not return the group corresponding to the user herself. Example 16 shows the corresponding GetAny request message.

```
<!--            Example 16            -->
<env:Body>
  <ck:GetAnyRequest
      xmlns:ck="http://mexcal.intermidia.icmc.usp.br/ck/xsd/GetAnyRequest.xsd">
    <publicID>h4YNEXzZURxesfJnljRVgvTpx</publicID>
    <last value="ALL"/>
    <context>
      <premiseS>
        <boolean type="AND">
          <premise dimension="who" type="login" value="cjardim"/>
        </boolean>
      </premiseS>
      <inferenceS>
        <inference dimension="what" type="group"/>
      </inferenceS>
    </context>
  </ck:GetAnyRequest>
</env:Body>
```

With the list of all users that belong to a group, *WebMemex* offers the opportunity for the users to enter a chat session with all the users of the group that are currently online: this is done simply by activating *WebChat* (detailed next) with the current user, which causes *WebChat* to open a chat session with those users.

4.4. The WebChat application

To illustrate the use of the facilities of the Context Kernel and the *WebLogin* and *WebRegister* applications, we built *WebChat*, a web-based chat system. In this case, a typical scenario is a user starting the *WebChat* application to start a session with the members of one of the groups that he belongs to — those that are currently online (group awareness). For instance, if a user recommends a web page to a friend using the *WebMemex* application, it is likely that they want to talk about it (Figure 3).

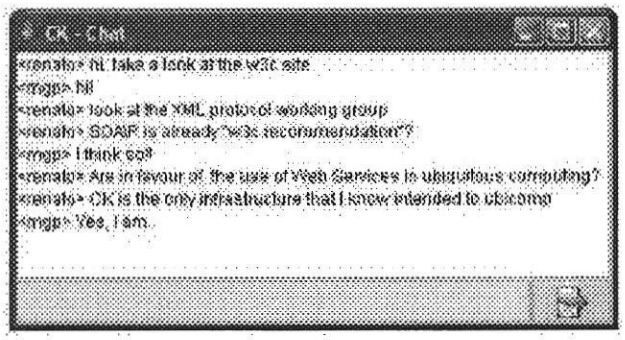

Figure 3: The WebChat interface illustrating a conversation about a web page recommended by one of the members of the chat session.

When starting *WebChat*, if the user is not already logged in (via any other application that uses the CK), *WebChat* calls *WebLogin*. After the user has successfully logged in, *WebChat* requests the operation User.getBuddyList() that invokes the GetInverse service. That service searches for all set of premises that have a *what* inference with type "group" and value equal to the name of each group that the current user is member of. The following XML excerpt (Example 17) illustrates that situation.

```
<!--            Example 17            -->
<env:Body>
  <ck:GetInverseRequest
      xmlns:ck="http://coweb.icmc.usp.br/ck/xsd/GetInverseRequest.xsd">
    <publicID>h4YNEXzZURxesfJnljRVgvTpx</publicID>
    <context>
      <inferenceS>
        <inference dimension="what" type="group" value="intermedia"/>
      </inferenceS>
      <premiseS>
        <premise dimension="who" type="login"/>
      </premiseS>
    </context>
  </ck:GetInverseRequest>
</env:Body>
```

In this case, the Context Kernel may reply the following triples: (dimension = "who", type = "login", value = "cjardim"), (dimension = "who", type = "login", value = "rbulcao") and (dimension = "who", type = "login", value = "jcamacho"), i.e., users "cjardim", "rbulcao" and "jcamacho" belong to the "intermedia" group.

To verify the activity of a user, *WebChat* needs then invoke the *Context Daemon* notification service. *WebChat* must register its interest in being notified about user activity as detailed in Example 14. *WebChat* also provides options for a user to create a chat session with all users of his group or with specific online users.

4.5. Discussion

We have shown how the Context Kernel infrastructure can be used as an alternative to manage context information of CSCW applications. Some applications, such as the *WebMemex*, were chosen as a proof of concept of the contributions that the Context Kernel can provide.

It is worth observing that the former version of the *WebMemex* treated user authentication by means of the "Yahoo! Messenger" service [Macedo et al., 2003]. When users signed in to use *WebMemex*, the "Yahoo! Messenger" was called to the correct identifier and password combinations and provided the users' buddy lists, or their online circle of friends. Those lists allowed the *WebMemex* to determine with whom the users' captured browsing information should be shared. Once related links were computed and stored, the storage component provided the presentation component with a list of URLs to similar documents. The presentation component formatted these URLs into links presented to the users as recommendations displayed in a small pop-up browser window (Figure 2).

Some problems and limitations were identified by *WebMemex* users. The use of "Yahoo! Messenger" to manage the user and group authentication imposed overheads for users to interact with *WebMemex*. By combining the services provided by Context Kernel and by *WebLogin* and *WebRegister* applications, *WebMemex* can reuse independent modules of storage, interpretation and exchange of context information regarding to users and groups. As a result, by using Web Service-based version of *WebMemex*, users are able to:

1. recommend explicitly viewing URLs to a specific group;
2. create private or public groups using *WebRegister*;
3. choose from their list of groups which one they want to share information with;
4. share user and group information with other applications, such as the *WebChat* system.

As a conclusion, we advocate that the Context Kernel Web Service is a viable solution for managing context information of CSCW applications since:

1. applications do not need to implement the whole storage and exchange infrastructure of context information;
2. CK provides a uniform schema of representation of context information for applications;
3. context information can be shared among applications such as user and group information to provide group awareness;
4. CK allows to integrate applications that have not been designed to work together (e.g. the *WebMemex* and the *WebChat* applications).

Compared with other solutions from the literature [Gross and Prinz, 2003] [Rittenbruch, 1999] [Fuchs, 1999], the modelling of contextual information for CSCW applications is quite basic. This is the disadvantage of demanding that applications provide appropriate models for the information they manipulate. However, once a model

is defined (as the one implicitly defined by *WebRegister*), any interested application can share the corresponding information stored on the Context Kernel repository. The next section presents further discussions on the advantages and disadvantages of this approach.

5. Concluding remarks

Most efforts reported on the CSCW literature — in terms of architectural models, toolkits or generic infrastructures for event notification of context awareness control — are broader than the current version of the Context Kernel in terms of the amount, type and depth of the CSCW-related functionalities provided. The main contribution of the Context Kernel is relative to its proposal of a Web Service-related standards as the model for communication as well as the sharing of social awareness context information; the emphasis on social awareness is due to the expected delays associated with using the Web as the communication infrastructure. The implications include: (i) the support to several levels of heterogeneity among applications; (ii) applications can not only store but also exchange context information and (iii) components can provide independent but integrated distribution of responsibilities inter and intra applications. It is important to observe that existing applications can make use of the Context Kernel facilities towards making their information available to other applications or, else, to make use of information made available by other applications.

Another contribution results from the specification of the contextual information based on the classical dimensions for context: *who*, *when*, *where*, *what*, *why* and *how*. Such specifications can be mapped from those supported by other models, such as those proposed in *AREA*, *Atmosphere* and the *Extended ENI* architecture — since the context information itself can be described by means of primitives in the context dimensions and the relationships and hierarchies can be mapped to the rules that define the premisses and associated inferences. Although the verbosity of the model can be a problem in terms of processing and long-term persistent storage, we have been studying the possibility of incorporating, in the long term, artificial intelligence approaches on the server side so as to benefit many CSCW applications [Bulcão Neto and Pimentel, 2003].

An important limitation resulting to using the Web Services approach is the lack of support to notification services — a most important requirement to CSCW toolkits and generic infrastructures. This has been tackled by extending the specification to a support notification service and implementing the associated functionalities.

Other limitations of the current design of Context Kernel have been well discussed in the literature. One of them is related to privacy control, as deeply discussed by [Fuchs, 1999]. The current version supports basic control in terms of storage and retrieval. In terms of storage, the main restriction is that only the application that has generated a rule has writing permission on data relative to that rule. While this is quite a simple restriction, it is also powerful since it guarantees the proposed model of the Context Kernel being a repository for storage and sharing of information among applications.

On the other spectrum of privacy control is the fact that, ultimately, all applications have *read* access to the information stored by any other application. Although this is convenient for loosely coupled cooperative applications, it limits the use of the infrastructure by applications that have such an open class of information to share.

Another limitation is related to the persistence of the information. Although the Context Kernel has been designed to be a persistent storage as demanded by some types of ubiquitous computing applications, this model is not necessary for most CSCW applications. Therefore, alternative politics for dealing with persistence of information have to be implemented.

It is important to observe that the underlying model of the Context Kernel demands that the applications define what information they store and what information, stored by other applications, they obtain from the repository. This means that users are unlikely to be in charge of specifying rules for the impact of some context information — the approach is that the conceptual model of the applications take into account the existence of such a repository.

References

Abowd, G., Mynatt, E. D., and Rodden, T. (2002). The Human Experience. *IEEE Pervasive Computing*, 1(1):48–57.

Anderson, G. E., Graham, T. C. N., and Wright, T. N. (2000). DragonFly: Linking Conceptual and Implementation Architectures of Multiuser Interactive Systems. In *Proceedings of the ACM Conference on Software Engineering*, pages 252–261.

Arruda Jr, C. R. E., Bulcão Neto, R. F., and Pimentel, M. G. C. (2003). Open Context-aware Storage as a Web Service. In *Proceedings of the International Workshop on Middleware for Pervasive and Ad-Hoc Computing held as part of the ACM/IFIP/USENIX International Middleware Conference*, pages 81–87.

Arsanjani, A., Hailpern, B., Martin, J., and Tarr, P. (2003). Web Services: Promises and Compromises. *ACM Queue (Web Services)*, 1(1):48–58.

Brotherton, J. A., Abowd, G. D., and Truong, K. N. (1999). Supporting Capture and Access Interfaces for Informal and Opportunistic Meetings. Technical Report GIT-GVU-99-06, Georgia Institute of Technology.

Bulcão Neto, R. F. and Pimentel, M. G. C. (2003). Semantic Interoperability between Context-aware Applications. In *Proceedings of the Brazilian Symposium on Multimedia and Web Systems*, pages 371–385. (In Portuguese).

Burner, M. (2003). The Deliberate Revolution. *ACM Queue (Web Services)*, 1(1):28–37.

Burrell, J., Gay, G. K., Kubo, K., and Farina, N. (2002). Context-aware Computing: A Test Case. In *Proceedings of the International Conference on Ubiquitous Computing*, pages 1–15.

Chabert, A., Grossman, E., Jackson, L., Pietrowicz, S., and Seguin, C. (1998). Java Object-Sharing in Habanero. *Communications of the ACM*, 41(6):69–76.

Cheng, S., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B., Steenkiste, P., and Hu, N. (2002). Software Architecture-based Adaptation for Pervasive Systems. In *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing*, pages 67–82.

Dey, A. K. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7.

Dey, A. K., Abowd, G. D., and Salber, D. (1999a). A Context-based Infrastructure for Smart Environments. In *Proceedings of the International Workshop on Managing Interactions in Smart Environments*, pages 114–128.

Dey, A. K., Salber, D., Abowd, G. D., and Futakawa, M. (1999b). The Conference Assistant: Combining Context-awareness with Wearable Computing. In *Proceedings of the International Symposium on Wearable Computing*, pages 21–28.

Edwards, W. K., Newman, M. W., Sedivy, J. Z., Smith, T. F., Balfanz, D., Smetters, D. K., Wong, H. C., and Izadi, S. (2002). Using SpeakEasy for Ad Hoc Peer-to-Peer Collaboration. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 256–265.

Fitzpatrick, G., Mansfield, T., Kaplan, S., Arnold, D., Phelps, T., and Segall, B. (1999). Augmenting the Workaday World with Elvin. In *Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 431–450.

Fremantle, P., Weerawarana, S., and Khalaf, R. (2002). Enterprise Services. *Communications of the ACM*, 45(10):77–82.

Fuchs, L. (1999). AREA: A Cross-application Notification Service for Groupware. In *Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 61–80.

Fuchs, L., Poltrock, S. E., and Wetzel, I. (2001). TeamSpace: An Environment for Team Articulation Work and Virtual Meetings. In *DEXA Workshop*, pages 527–531.

Garlan, D., Siewiorek, D., Smailagic, A., and Steenkiste, P. (2002). Project Aura: Toward Distraction-free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31.

Gellersen, H.-W. and Schmidt, A. (2002). Look Who's Visiting: Supporting Visitor Awareness in the Web. *International Journal on Human-Computer Studies*, 56(1):25–46.

Girgensohn, A. and Lee, A. (2002). Making Web Sites be Places for Social Interaction. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 136–145.

Greenberg, S. (2001). Context as a Dynamic Construct. *HCI*, 16:257–268.

Gross, T. and Prinz, W. (2003). Awareness in Context: a Light-weight Approach. In *Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 295–314.

Hess, C. K., Román, M., and Campbell, R. H. (2002). Building Applications for Ubiquitous Computing Environments. In *Proccedings of the International Conference on Pervasive Computing*, pages 16–29.

Hong, J. I. (2002). The Context Fabric: An Infrastructure for Context-aware Computing. In *Proceedings of the ACM Conference on Computer-Human Interaction*, pages 554–555.

Hong, J. I. and Landay, J. A. (2001). An Infrastructure Approach to Context-aware Computing. *Human-Computer Interaction (HCI) Journal*, 16(2-3).

Laurillau, Y. and Nigay, L. (2002). Clover Architecture for Groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 236–245.

Li, D. and Li, R. (2002). Transparent Sharing and Interoperation of Heterogeneous Single-user Applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 246–255.

Macedo, A. A., Truong, K. N., Camacho-Guerrero, J. A., and Pimentel, M. G. C. (2003). Automatically Sharing Web Experiences through a Hyperdocument Recommender System. In *Proceedings of the ACM Conference on Hypertext and Hypermedia*, pages 48–56. ACM Press.

McKusick, K. (2003). Interview with Adam Bosworth. *ACM Queue (Web Services)*, 1(1):12–21.

McNee, S. M., Albert, I., Cosley, D., Gopalkrishnan, P., Lam, S. K., Rashid, A. M., Konstan, J. A., and Riedl, J. (2002). On the Recommending of Citations for Research Papers. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 116–125.

Nagel, K., Kidd, C. D., O'Connell, T., Dey, A. K., and Abowd, G. D. (2001). The Family Intercom: Developing a Context-aware Audio Communication System. In *Procéedings of the International Conference on Ubiquitous Computing*, pages 176–183.

Patterson, J.F., M. D. and Kucan, J. (1996). Notification Servers for Synchronous Groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 122–129.

Prinz, W. (1999). NESSIE: An Awareness Environment for Cooperative Settings. In *Proceedings of the European Conference on Computer-Supported Cooperative Work*, pages 391–410.

Ranganathan, A., Campbell, R., Ravi, A., and Mahajan, A. (2002). ConChat: A Context-aware Chat Program. *IEEE Pervasive Computing*, 1(3):51–57.

Ribak, A., Jacovi, M., and Soroka, V. (2002). Ask Before You Search: Peer Support and Community Building with ReachOut. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 126–135.

Rittenbruch, M. (1999). Atmosphere: Towards Context-selective Awareness Mechanisms Human-Computer Interaction: Communication, Cooperation and Application Design. In *Proceedings of the International Conference on Human-Computer Interaction*, pages 328–332.

Roseman, M. and Greenberg, S. (1996). Groupware with GroupKit, a Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106.

Salber, D., Siewiorek, D. P., and Smailagic, A. (2001). Supporting Mobile Workgroups on a Wireless Campus. In *Proceedings of the International Workshop on Human Computer Interaction with Mobile Devices*.

Schilit, B. N., Adams, N., Gold, R., Tso, M., and Want, R. (1993). The ParcTab Mobile Computing System. In *Proceedings of the Workshop on Workstation Operating Systems*, pages 34–39.

Stal, M. (2002). Web Services: Beyond Component-based Computing. *Communications of the ACM*, 45(10):71–76.

Truong, K. N., Abowd, G. D., and Brotherton, J. A. (2001). Who, What, When, Where, How: Design issues of capture & access applications. In *Proceedings of the International Conference on Ubiquitous Computing*, pages 209–224.

W3C (2001). XML Schema, W3C Recommendation. Available online in `http://www.w3.org/TR/xmlschema-0/`.

W3C (2002). Web Services Activity. Available online in `http://www.w3.org/2002/ws`.

W3C (2003a). Simple Object Access Protocol (SOAP) 1.2, W3C Proposed Recommendation. Available online in `http://www.w3.org/TR/soap12-part0/`.

W3C (2003b). Web Services Description Language (WSDL) 1.2, Working Draft. Available online in `http://www.w3.org/TR/wsdl12/`.

Want, R., Hopper, A., Falco, V., and Gibbons, J. (1992). The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102.