The Journal of Systems and Software 92 (2014) 59–70

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Programmable context awareness framework

Bachir Chihani [a,b,*], Emmanuel Bertin [a,b], Noël Crespi [b]

[a] Orange Labs, 42 rue des Coutures, 14066 Caen, France
[b] Institut Mines-Telecom, Telecom SudParis, CNRS 5157, 9 rue Charles Fourier, 91011 Evry, France

ARTICLE INFO

Article history:
Received 28 September 2012
Received in revised form 9 July 2013
Accepted 22 July 2013
Available online 1 August 2013

Keywords:
Software engineering
Context-awareness
Privacy
Adaptation
XML

ABSTRACT

Context-awareness enables applications to provide end-users with a richer experience by enhancing their interactions with contextual information. Several frameworks have already been proposed to simplify the development of context-aware applications. These frameworks are focused on provisioning context data and on providing common semantics, definitions and representations of these context data. They assume that applications share the same semantic, which limits the range of use cases where a framework can be used, as that assumption induces a strong coupling between context management and application logic. This article proposes a framework that decouples context management from application business logic. The aim is to reduce the overhead on applications that run on resource-limited devices while still providing mechanisms to support context-awareness and behavior adaptation. The article presents an innovative approach that involves third-parties in context processing definition by structuring it using atomic functions. These functions can be designed by third-party developers using an XML-based programming language. Its implementation and evaluation demonstrates the benefits, in terms of flexibility, of using proven design patterns from software engineering for developing context-aware application.

1. Introduction

Two distinctive architectural approaches (Gutheim, 2011) have been used to design context-aware applications: architectures that follow a broker model and those based on a point-to-point model. Both models contain two types of elements: context providers in charge of collecting contextual data, and context consumers (i.e., context-aware applications) that use contexts to adapt their behavior. In the first model, a context broker is used as an additional element to decouple context providers and consumers, limiting or eliminating the direct connections between them. This broker is thus, in most cases, in charge of context modeling and inference (Gutheim, 2011; Naudet, 2011; Filho and Agoulmine, 2011; Hamadache and Lancieri, 2010; van Sinderen et al., 2006). However, these tasks are performed in a pre-defined way and are barely customizable by context consumers. As a result, the generated information may not fully match the specific needs of consumers. In the second model, context consumers know the providers and send their requests to them directly (Oh et al., 2010). This model is less sophisticated; context consumers need to know which provider should be addressed for any given contextual information and should also be aware of their state (e.g., awake or asleep).

In both cases, consumers continuously request contextual information from their sources (either the context broker or directly from context providers) or subscribe to be notified with context updates. Upon reception, this information must then be processed by the context consumers to determine if it would impact their behavior. When the updating load increases, consumers become overloaded with messages, many of which are not at all relevant to them. Moreover, consumers have to store and handle context information locally to maintain a consistent vision of the user's situation and to adapt their behavior accordingly.

We have thus identified the need for a better separation within context-aware architectures. The application business logic and the context management operations should be kept separate, so that one of these logics can be modified without having to modify the other. To enable this separation, we propose to host all operations related to context management in the context broker, outside of context-aware applications. All of the context management operations to be executed in the broker will need to be specified with an appropriate language. For a given context-aware application, a script or a program written in this language will be executed by the context broker as the specified context events occur (e.g., events from context providers). The result of each execution is then sent back to the corresponding application, which does not have to maintain or retrieve this data.

* Corresponding author at: Orange Labs, 42 rue des Coutures, 14066 Caen, France. Tel.: +33 2 31 83 90 05.
E-mail addresses: bachir.chihani@orange.com (B. Chihani), emmanuel.bertin@orange.com (E. Bertin), noel.crespi@it-sudparis.eu (N. Crespi).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.07.046

We present a context management framework based on a broker architecture, which gives context consumers the ability to design, via an XML-based scripting language, their specific context processing that will be executed by the broker. This will allow context handling to be centralized into the broker, instead of being partially processed by the different context-aware services.

The rest of the paper is organized as follows. Section 2 presents our motivations and challenges for building effective context management architectures. Section 3 illustrates our framework architecture and the implementation details. Case studies showing the possible applications of the framework are presented in Section 4. In Section 5, we present an evaluation of the proposed framework through a real case study and performance evaluation in a simulated environment. Section 6 discusses related work and the proposed framework. Finally, we conclude the paper and present future work in Section 7.

2. Motivation and challenges

In this section we highlight the benefits of a modular design of context management frameworks by separating the main features of the framework into different non-overlapping functions, thereby making it much simpler to build different types of context-aware applications.

2.1. Context-awareness

Context-awareness aims to enhance services (e.g., tracking, navigation, information, communication, and entertainment services) by making them sensitive to users' situations, and therefore making them more adapted to user's needs (Chihani et al., 2011a). Context management frameworks are used to empower the development of such services by gathering context-related functions in a common component and exposing its functionalities to third-party services looking for context awareness.

Many of the efforts in building context management frameworks are focused on specific domains (e.g., IPTV (Song et al., 2010), IMS (IP Multimedia Subsystem) environments (Simoes and Magedanz, 2011), and document management systems (Balinsky et al., 2011)) making their proposed frameworks virtually impossible to re-use for building context-aware applications in other domains. Other works have proposed more generic frameworks (Gutheim, 2011; van Sinderen et al., 2006; Plesa and Logrippo, 2007) enabling different types of applications to coexist and to be built on top of these frameworks. However, these frameworks do not offer customization capabilities – a serious lack since applications often have different requirements.

Our contribution is a generic framework, based on the Observer design pattern (Gamma et al., 1994), which is flexible enough to be customized for building context-aware applications in different domains. Herein, flexibility refers to dynamic adaptation, customization and component reusability. The use of the Observer pattern allows the framework core component (i.e., the context broker) to be observed by third party applications context consumers. In addition, these consumers can define, through a specification document, the set of internal states of the broker they want to observe in order to be notified of any specific changes. Furthermore, this specification defines how the framework should process context data, as well as the application callback address on which to be notified when an event of interest occurs. When the broker processes new published context data, it may change its internal state and as a result notifies the consumers interested in these events by sending messages to their callback addresses.

2.2. Design considerations

Contextual information is dynamic by nature. The measurements change over time (e.g., temperature, location) with a variable rate. Also, a context may lose consistency (become outdated or incorrect) because the sensing operation may fail or the sensing environment may become too noisy. These characteristics bring a high degree of complexity to context management, and consequently to context-aware applications, as context management is usually coupled with application logic.

To address the dynamic characteristics of contextual information and its complicated management issues, we identify the following requirements for a 'developer-friendly' context management framework:

- The framework should support the rapid development of context-aware applications through the reduction of context management complexity.
- To facilitate the re-use of the architecture in different environments (e.g., Telcos, Machine-to-Machine) the communication protocol used to transport information between all components of the architecture should be generic and not specific to a given technology.
- Information representation should be very flexible so that application developers are not forced to respect strict formatting rules in order to circulate data among the architecture's components. Applications should be able to subscribe to a subset of all context events generated by updates from providers for given contextual information.
- The mechanisms provided to application developers for customizing context processing should be kept as simple as possible to reduce the learning curve for developers.
- The framework should provide appropriate mechanisms allowing the respect of user privacy enabling users to grant or prohibit third-party applications to access his/her context data and as a result controlling the adaptation level of these applications.

3. Context management framework

3.1. Framework architecture

In order to meet these requirements, we propose a programmable framework (Fig. 1) for processing contextual information, based on six primitive functions related to context management: *produce*, *filter*, *abstract*, *select*, *aggregate*, and *consume*. Each of these functions corresponds to a specific action from the well-known layered approach (Schmidt, 2006; Chihani et al., 2011a) in context management.

The "*produce*" function consists of producing raw contextual information. It is implemented by context providers that wrap sensors to comply with the framework API for publishing context.

The "*filter*" function provides signal processing functionalities that aim to eliminate or at least reduce noise in contextual

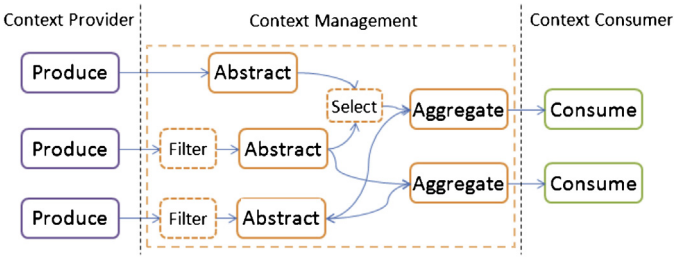

Fig. 1. A layered framework for context management.

Listing 1. DTD of CPDL (Context Processing Definition Language).

```
<!DOCTYPE Definition [
<!ELEMENT Definition (Filter | Automata | Select | Rule)*>
<!ELEMENT Filter EMPTY>
<!ATTLIST Filter Src CDATA #REQUIRED Type CDATA #REQUIRED Dst CDATA #REQUIRED >
<!ELEMENT Automata (State*, Start-State, End-State)>
<!ATTLIST Automata Id CDATA #REQUIRED Name CDATA #REQUIRED>
<!ELEMENT State (Transition)*>
<!ATTLIST State Name CDATA #REQUIRED >
<!ELEMENT Transition (Condition | Event)*>
<!ATTLIST Transition Dest CDATA #REQUIRED >
<!ELEMENT Select (Input | Output)* >
<!ATTLIST Select Param CDATA #REQUIRED Opt CDATA #REQUIRED >
<!ELEMENT Input EMPTY >
<!ATTLIST Input Channel CDATA #REQUIRED >
<!ELEMENT Output EMPTY >
<!ATTLIST Output Channel CDATA #REQUIRED >
<!ELEMENT Rule (Condition | Event)*>
<!ATTLIST Rule Id CDATA #REQUIRED Name CDATA #REQUIRED >
<!ELEMENT Condition EMPTY >
<!ATTLIST Condition Type CDATA #REQUIRED Key CDATA #REQUIRED Operator CDATA
#REQUIRED Value CDATA #REQUIRED >
<!ELEMENT Event EMPTY >
<!ATTLIST Event Type CDATA #REQUIRED To CDATA #REQUIRED Message CDATA #IMPLIED
>
]>
```

information and that are collected as physical measurements. It is implemented via the "Filter" element that defines which signal processing algorithm to use for any given contextual information available from the corresponding source. Some examples of signal processing algorithms that can be selected are the Kalman Filter and the Simple Moving Average (SMA), which is a type of low-pass filter. For example, a moving average filter can be applied to the returned values of a temperature sensor to smooth the returned values and eliminate possible noise.

The "*abstract*" function transforms raw contextual information into a higher level of abstraction. It is implemented via the "Automata" element that defines a finite state machine composed of the different states that context data may have. As a trivial example: the context 'availability' may have two states, 'available' and 'not available'. The context 'availability' transits between these states following changes occurring at a raw context level, such as 'presence', sensed from an instant messaging system. A finite state may transit from one state to another only if some preconditions are satisfied. These preconditions are related to context publication from outside (i.e., from context providers), to internal events triggered from another finite state machine, or are the result of timeout expiration when the condition is implemented as a timer. A transition may lead to events, which can be internal (to trigger another finite state machine or an aggregation rule), or external, such as to notify a context consumer of its callback address.

The "*select*" function enables the selection of the 'best' available context (from among those provided by multiple sources) based on programmable criteria. It is implemented with the "Select" element that defines the parameter to compare in the selection process, and that describes the operation to apply for this selection process (i.e., choosing the maximum or the minimum value). The compared parameter might characterize the quality of a context, such as accuracy or precision. For example, the abstract location (e.g., home, work) of a person can be obtained by abstracting GPS information and also from location information acquired from an instant messaging system. The difference between the two location sources is in the precision, which is high for the first source and low for the second. A tracking application may choose to select a source that provides the maximum available precision.

The "*aggregate*" function performs an aggregation on a set of contextual information in order to generate a composite context to be exposed to context-aware applications. The "aggregate" function is implemented thanks to the 'Rule' element that defines a set of conditions to be met by different contextual data, and a set of actions, in the form of events and notifications to a context-aware application. The aggregate function represents an IF-THEN rule that can be triggered by one or more abstract functions under special states, i.e., when some pre-conditions have been verified. The condition part of an aggregation rule is composed of a set of key-value pairs where keys represent context data (e.g., availability) and values represent references or thresholds related to this context data (e.g., available). The verification of the condition part is performed after that the rule has been triggered.

Finally, the "*consume*" function is performed by context consumers (i.e., the context-aware applications) as they consume the context of any level (raw, abstract or composite) and adapt their behavior accordingly.

3.2. Specification language

The context management framework implements the *filter*, *abstract*, *select* and *aggregate* functions. All these functions are programmable by context-aware applications with the help of an XML language (that we developed and have named CPDL, for Context Processing Definition Language). Listing 1 depicts the DTD (Document Type Definition) corresponding to CPDL.

The finite state machine (*abstract* function) listens for raw contextual information to know when to transit between its states. For certain transitions, an internal event that will trigger a local adaptation rule may be specified (*aggregate* function). This aggregation rule will compare the current context values to a given situation which is defined in the condition part of the rule, and then notify context-aware applications when a match occurs. As these two functions are completely programmable, context-aware application developers are able to define and implement a part of their own reasoning logic into a remote and central component (i.e., the context broker). This provides a distinct benefit to applications, since context updates are filtered and part of their intelligence is implemented on the context management framework, enabling the development of lightweight context-aware applications.

An example of an aggregation rule is a rule that checks a user's availability when a new call arrives to decide how it should be

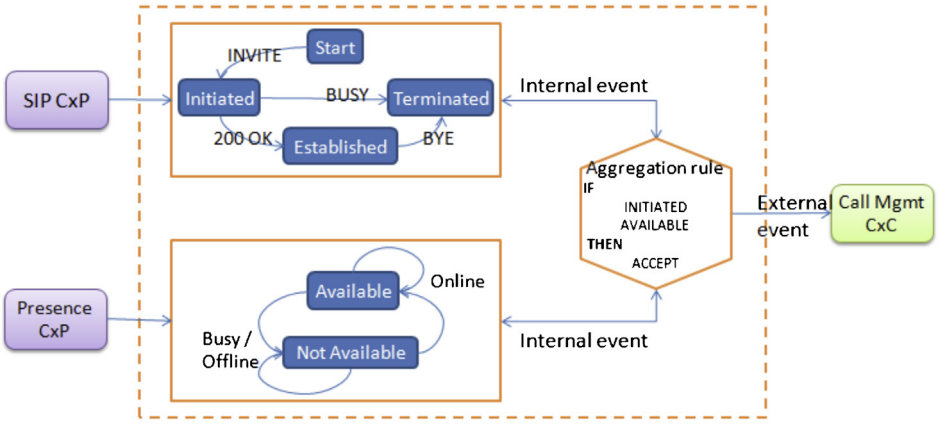

Fig. 2. Example of *abstract* and *aggregate* functions.

handled (e.g., reject the call). If an abstract function is used to abstract SIP (Session Initiation Protocol) communication states into 'start', 'initiated', 'established' and 'terminated', then the aggregation rule is triggered when the finite state machine corresponding to the SIP abstract function transits from state 'start' to state 'initiated' (e.g., after reception of a SIP INVITE message). The rule will check the current user context (e.g., availability) and if the current value is available it notifies the context-aware application. Sent notifications correspond to the appropriate messages described in the CPDL (i.e., event element) for the current situation.

Fig. 2 depicts an example of a graphical representation of context reasoning defined with CPDL. A finite state machine is used to abstract SIP messages (e.g., INVITE) into a communication state (e.g., initiated), and another state machine is used to abstract presence states (e.g., online) into availability states (e.g., available, not available). The aggregation rule is triggered to verify the current context against a given situation described in the condition part of this rule. If the rule conditions are satisfied in the current situation then a message is sent to the context consumer on the callback address specified in the CPDL.

3.3. Implementation

We implemented these concepts with a context management platform, brokering between context-aware applications (installed on smartphones) and sensors (located either on smartphones or on external servers).

Fig. 3 illustrates the implementation: CxPs are context providers responsible for the retrieval of contextual information from sources and for sending it to the context management platform (CMP) that is in charge of managing context and its distribution. context consumers (CxCs) are applications that consume contextual information and adapt their behavior accordingly. The context broker

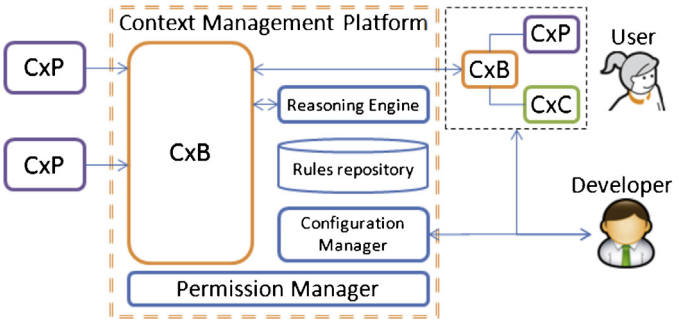

Fig. 3. Context management framework.

(CxB) is responsible for distributing context from sources (CxPs) to its consumers (CxC). A local context broker is used at the device level as a Cache to store contextual information locally (and temporarily until expiration) to speed up future context requests. CxPs installed on user devices should publish their produced context to the local CxB that is in charge of forwarding it to other components (Chihani et al., 2011c). A configuration manager (CM) manages the reasoning configuration files and assures their validation. A rules repository is used to store the reasoning rules. A reasoning engine (RE) is responsible for applying the reasoning rules defined by the context-aware applications to the context events generated by updates from CxPs. A permission manager (PM) is used to enforce user policies, such as which components (CxC) are allowed to request a user's contextual information.

The user is the owner of the smartphone on which the context-aware application is installed. The developer is in charge of producing a context-aware application.

The developer(s) upload a CPDL file describing the context-aware application's reasoning configuration to the CM component. The CM component instantiates the reasoning rules corresponding to the configuration file after its analysis and its validation by checking it with the corresponding DTD file(s). After instantiation, the rules become operational and the RE component will apply them every time a new context event is received.

3.4. Context distribution

The framework is based on a RESTful (Representational state transfer) architecture where components are web services and contextual information is addressed via a Context URI (Unified Resource Identifier) called a 'channel'. This architecture greatly facilitates context distribution among the different framework components. The context provider (CxP) chooses a channel (addresses are used to uniquely identify a context) on which it will publish the produced context. Providers of the same contextual information (e.g., location) can publish content on the same specific channel. The CxB contains a mapping table that matches context data and corresponding URIs for requests and subscriptions to context, and another lookup table (Table 1) is used to track who is publishing to and who is consuming from a given context

Table 1
An example of a CxB context lookup table.

Channel	Context providers	Context consumers
A = "/{user_id}/presence"	CxP_1	CxC_3
B = "/{device_id}/settings"	CxP_2	CxC_1, CxC_3
C = "/{POI}/location"	CxP_1	CxC_2

Listing 2. A snippet of an XML document representing location context.

```
<!DOCTYPE Definition [
<!ELEMENT Definition (Filter | Automata | Select | Rule)*>
<!ELEMENT Filter EMPTY>
<!ATTLIST Filter Src CDATA #REQUIRED Type CDATA #REQUIRED Dst CDATA #REQUIRED >
<!ELEMENT Automata (State*, Start-State, End-State)>
<!ATTLIST Automata Id CDATA #REQUIRED Name CDATA #REQUIRED>
<!ELEMENT State (Transition)*>
<!ATTLIST State Name CDATA #REQUIRED >
<!ELEMENT Transition (Condition | Event)*>
<!ATTLIST Transition Dest CDATA #REQUIRED >
<!ELEMENT Select (Input | Output)* >
<!ATTLIST Select Param CDATA #REQUIRED Opt CDATA #REQUIRED >
<!ELEMENT Input EMPTY >
<!ATTLIST Input Channel CDATA #REQUIRED >
<!ELEMENT Output EMPTY >
<!ATTLIST Output Channel CDATA #REQUIRED >
<!ELEMENT Rule (Condition | Event)*>
<!ATTLIST Rule Id CDATA #REQUIRED Name CDATA #REQUIRED >
<!ELEMENT Condition EMPTY >
<!ATTLIST Condition Type CDATA #REQUIRED Key CDATA #REQUIRED Operator CDATA
#REQUIRED Value CDATA #REQUIRED >
```

channel. A consumer (CxC) interested in requesting or subscribing to specific contextual information can ask the broker (CxB) for access by providing the corresponding channel. Context consumers subscribe to these channels to be notified when new information is published.

The provisioned context corresponds to information such as user location, proximity (nearby wifi access point or Bluetooth devices), activity, profile information, device status, etc. It is represented in XML for an easy distribution among the framework components. The following listing represents an example of contextual information about a known/detected user location that is made available by the context broker on channel "/p000/location".

The use of channels in conjunction with an XML-based context representation provides a hierarchical addressing schema allowing the composition of context data. For instance, if Alice's presence information is published on channel '/alice/presence' and her location information is published on '/alice/location' then Alice's composite context information (presence and location) can be requested from channel '/alice'. In the same way, if location information is represented in XML and contains an element for GPS coordinates (longitude and latitude) and another element for civil address, then channel '/alice/location/gps' should return the GPS coordinates.

External components have to register with the broker in order to participate in the management (publication, consumption) of user context. The registration consists of declaring descriptive information (depending on the type of management operations for the components involved) about the component on a special 'meta' channel. For a context producer, the required information includes the type of published context and the entity (e.g., a monitored user or a device) that owns the published context. For a context consumer, the required information can be derived from the corresponding CPDL file. If a context consumer only wants to subscribe to context updates and process them on its own, then that should be indicated at registration: both the requested context and the callback address to receive the notifications.

This registration information is used by the broker to route context updates from the CxPs to the CxCs. Privacy policies are used to regulate this routing as detailed in the next section. Thanks to this channel-based publish/subscribe messaging mode, the framework has a loosely coupled architecture in which consumers do not have to know the identity of providers since they get their context directly from the broker.

3.5. Privacy management

Controlling which element is accessing what information is a key to protecting user privacy, since there is automatic provisioning of user contextual information. Fig. 4 depicts our proposal for managing user privacy.

The role of the permission manager (PM) is to allow the users of context-aware applications to define permissions on a per-element basis, i.e., for each consumer and provider involved in the context-awareness process. It consists of a repository for storing users' privacy policies, and a privacy policy decision point (PDP) for evaluating and issuing permission decisions for CxCs requesting access to a given context. In addition to a brokering role, the CxB also acts as a privacy policy enforcement point (PEP) and is responsible for ensuring the enforcement of decisions made by the PM based on each user's policies regarding how his/her context data are to be used.

Users are asked, through a web portal, to give their permission (allow) or not (deny) for the execution of certain operations related to the management (production, storage and consumption) of their context data. These permissions are stored as privacy policies in the PM repository. At any time, users can view their current privacy policies and modify them. Users can also discover which component is accessing their context data, as well as the details of the information provided. For example, users can see the context consumers that are using their location data and the latest updates sent, or the last-requested information. In a future version, we plan to implement an email notification to users whenever a new context consumer appears.

The infrastructure is able to provide this information thanks to the use of channels (i.e., context URI) for addressing context, and lookup tables for associating components to a given context as producers or consumers. If there is a policy forbidding a CxC

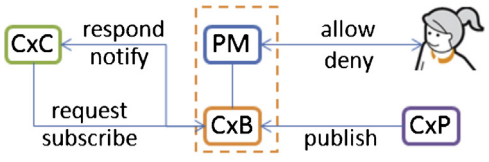

Fig. 4. Context-aware privacy management.

from requesting a specific context, then no new publications corresponding to this particular context will be forwarded to that CxC.

We believe that the availability of information about how contextual information is used by the different components helps to best select and enforce privacy-related decisions. In fact, users can easily decide how to set their privacy policies and modifications at runtime, e.g., after observing some not-so-suitable behavior from a certain component. This privacy manager can be used as a technical foundation for simple user interfaces.

4. Case study

The proposed framework can be used in many different ways, which can be classified into one of the following categories based on how the reasoning is performed: at the application or the broker side, or distributed across both sides. These different categories are presented in the following sub-sections with representative applications.

4.1. Hosted reasoning

In the hosted reasoning approach, the context-aware application hosts the reasoning on its side and the framework is only used for connecting and facilitating the communication between the different parts (i.e., CxP, CxC) of this application. An example of such an application is HEP: enhanced un-interruPtability (Chihani et al., 2011b,c). This sample application generates workload information related to the use of communication tools (SMS, phone) for users by collecting and processing communication logs. Then, instead of sharing presence information, HEP shares workload information with the user's contacts. Such information helps a user to assess whether his/her contacts can receive additional communication requests or not.

HEP consumes the logs histories and processes them locally on the user device, and then relies on the framework to dispatch the generated information to all the other subscribed instances. When a new HEP instance is installed on the user device, this instance registers itself to the broker (CxB) to declare the channel where the workload of the corresponding user will be published. Each time this user adds a new contact to his/her address book, HEP will send a request to the broker to subscribe to the workload information of this contact. As a result, the broker context look-up table is updated and each publication of the newly added contact's workload information is forwarded to the subscribed user.

In this scenario, the benefit brought by the use of the framework consists of reducing the effort needed for connecting several instances of HEP to circulate the workload information of the corresponding user in real time.

4.2. Hybrid reasoning

In the hybrid approach, part of the reasoning is performed on the broker side and another part is performed on the application side. For the part made at the broker side, the application has to upload its reasoning configuration file that will be instantiated by the broker. As an example of such application, we present CAAB (Context-Aware Address Book), an android application designed to enrich users' address books with contextual information representing their contacts' availability. Availability statuses (green for "available", red for "do not disturb", orange for "busy", and yellow for "unknown") are generated by aggregating a set of raw contexts, namely:

- the presence information provided by a communication suite (Microsoft Lync);
- the indoor location, which is information provided by a Bluetooth proxy; and
- the workload information related to communication tools (SMS, phone), that are collected from the smartphone communication logs (as described in the previous section).

Fig. 5. Screenshot from CAAB.

CAAB was developed for business users to help them contact their colleagues without disturbing them. This application also provides a means for users to rapidly send notifications to their peers in the form of predefined messages augmented with contextual information (e.g., a sender's location). For example, in Fig. 5, a notification message augmented with the indoor location of the sender is displayed to the receiving user. The moment of the notification delivery is conditioned by the receiver's actual workload information, i.e., the notification will be delivered only if the receiver's workload is not equal to "do not disturb", which refers to highest degree of 'busyness'.

The part of the reasoning performed by the application concerns the generation of the workload information. However, it is the broker that is used to combine this information with presence and location information. Fig. 6 depicts the deployment of the different parts of the application, as well as the instantiation of the context reasoning on the broker side. In this particular case, the reasoning aims to generate availability information that corresponds to a single user, Alice. It is composed of two finite state machines that perform the abstraction of a set of context data (location and workload) and a single aggregation rule to describe user availability. Presence information does not need to be abstracted as it already has an abstract nature. Regarding this configuration, a user is considered to be available if he/she is in a 'free' state (i.e., has a workload rating that is less than 50%), has a presence status 'online' on a given communication channel (e.g., instant messaging), and is located at the 'office'. Otherwise, a user is considered to be unavailable.

This is a simple way to derive availability information from other contextual information. We can easily change how availability is derived by modifying the corresponding CPDL structure. Also, reference values used in this CPDL (e.g., threshold 50 for workload level) can easily be personalized for a given user. These customizations can be performed without having to modify an application's behavior.

In this scenario, the framework allows the segregation of the application into multiple independent parts and their deployment across different locations (devices or servers) without introducing any complexity regarding the exchange of context data.

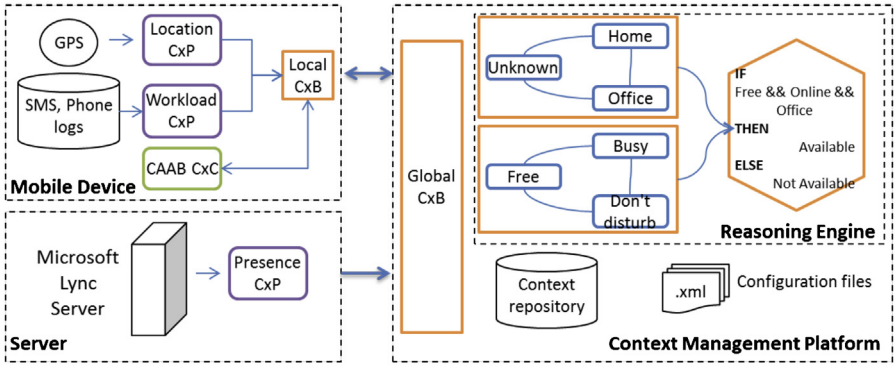

Fig. 6. CAAB context management.

4.3. Delegated reasoning

With the delegated approach the reasoning is performed on the broker side and the application only receives messages containing the action that should be performed. One example of an application that uses this approach is an IMS (IP Multimedia Subsystem)-based context-aware communication system (Chihani et al., 2012) that considers user context and preferences in handling incoming calls (e.g., accepting or rejecting a call).

Fig. 7 depicts the deployment of the different parts of the call management application. It also shows an example of a call management rule where user context is modeled as abstraction state machines, and its preferences for handling incoming calls are implemented as aggregation rules. Multiple types of contexts are used in the management rule. User location is obtained from the location provider and then abstracted to two main locations (work and home); other locations which are not labeled are abstracted as unknown state. The activity context is related to the active application (e.g., document processing software, client's email application) currently in use. This context is abstracted to the type of application: *office* gathers applications like MS Word or Outlook, *business* gathers specific applications such as CRM (Customer Relationship Management), *internet* is related to internet browsing. The communication context is related to communication activities, whether the user is in communication or not. Information from the call provider corresponds to the state evolution of an incoming call; when a call is initiated it triggers the aggregation rule that will check the current status of the different state machines to decide how to handle that call. The action resulting from the aggregation rule will be sent to the call management component for execution.

In this scenario, the framework allows a significant simplification in the development and maintenance of applications as a result of separating the context management (collecting it from sources and reasoning on it) from the core functionalities of the application, here the control of communications.

5. Evaluation

5.1. Efficiency

To evaluate the benefit of abstract–aggregate based reasoning compared to a regular case of the publish-subscribe paradigm where no reasoning is performed, we conducted an experimental study based on OMNeT++,[1] a powerful discrete event simulation toolkit. The topology used in the simulation is composed of a context broker to connect to the rest of the components, $n = 12$ context providers (CxPs) and $m = 8$ context consumers (CxCs). Context providers continuously publish context messages to the broker in an exponential distribution with a configurable mean (for each provider). Each context consumer subscribes to a random number of providers in a uniform fashion (all providers have a similar chance for a given subscriber $p = 1/n$). The context broker stores all the subscriptions in a routing table; when it receives a context update from a provider it duplicates the message to all the corresponding subscribers. The following table illustrates the simulation setup parameters.

The graphs in Fig. 8 present the throughput (number of context messages sent or received per second) on the context broker side. The red graph, at the bottom, represents the throughput at reception, i.e., the incoming context updates from the different providers. The graph in green, the upper graph, represents the throughput at emission when the abstract–aggregate reasoning approach is not deployed (i.e., all incoming messages are sent out to all subscribers). The blue graph, in the middle, represents the throughput at emission when the abstract–aggregate method is used to filter incoming updates and only send out relevant events. The throughput at emission is greater than that at reception because there is more than one subscriber for each context provider, and thus each incoming message has to be duplicated as many times as there are subscribers for the given context information.

As illustrated by these graphs, the use of the abstract–aggregate reasoning approach allows the reduction of emission throughput, since only relevant events (those relevant to a given application) are sent out.

Each context update from a provider is first consumed by the abstraction function that updates the current state of the corresponding finite state machine. The transition between the states may trigger, with a probability 'p', an event to the aggregation function that will verify the current situation, and as a result may trigger a notification to a corresponding context consumer with a probability 'q'. Thus the probability of an updated context to generate an event out to a consumer depends on both probabilities 'p' and 'q'.

The relation between the throughputs when the abstract–aggregate is used and the throughputs performed when the abstract–aggregate is not used is expressed in the following equation:

$$\text{Throughput}_{\text{with}} = \text{Throughput}_{\text{without}} \times p \times q \quad (1)$$

The results of the efficiency test suggest that the proposed framework significantly reduces the bandwidth consumption, as less context updates are circulated to consumers. However, these results represent the ideal case assuming an optimized context management configuration file. This may not be the case if the XML documents were not optimized, for instance if any context

[1] http://www.omnetpp.org/.

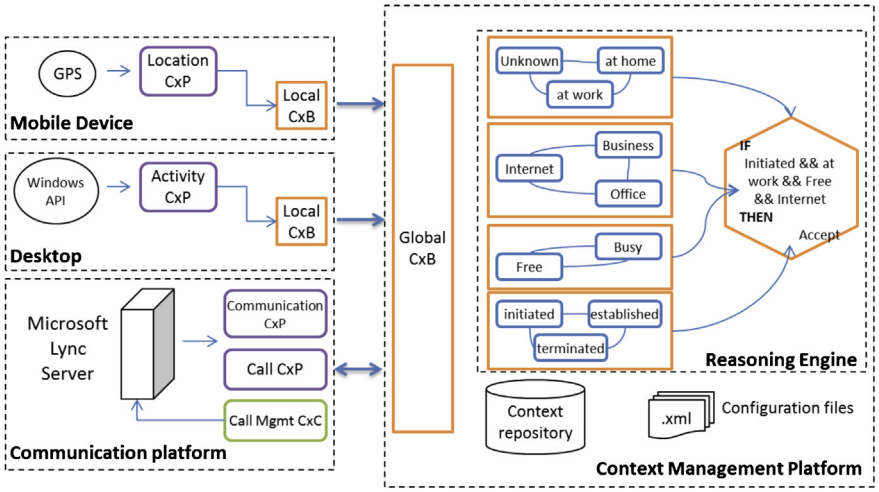

Fig. 7. Context-aware call management.

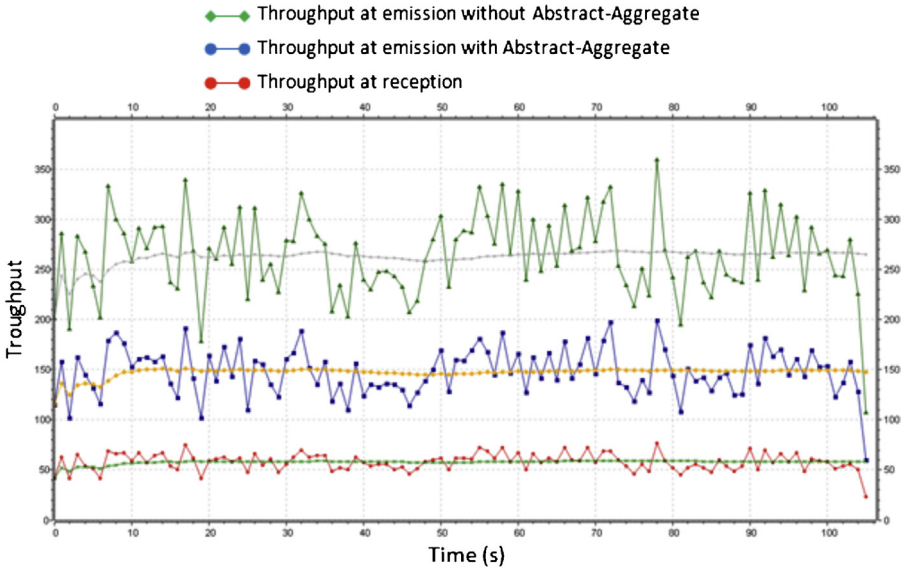

Fig. 8. Throughput comparison graph. (For interpretation of the references to color in the text, the reader is referred to the web version of the article.)

update would always satisfy the conditions of the aggregation rule so that all context updates will be forwarded to the consuming application.

5.2. Scalability

Another concern is to keep the framework response time reasonable to support scalability of the architecture and to support an increasing number of providers and/or consumers. From the context consumer, the response time corresponds to the amount of time needed for the provider to publish context to the broker plus the time needed for the broker to reason on the published context, plus the time needed for the broker to send a notification to the consumer. The transportation time (for publication and notification) depends on the network conditions and cannot be controlled. It is the reasoning time which is more interesting to study as it depends on the quantity of managed context and the number of hosted configuration files. To evaluate the overhead added onto the context broker when using the abstract–aggregate reasoning approach, we used an example of a context-aware system: the CAAB application described previously.

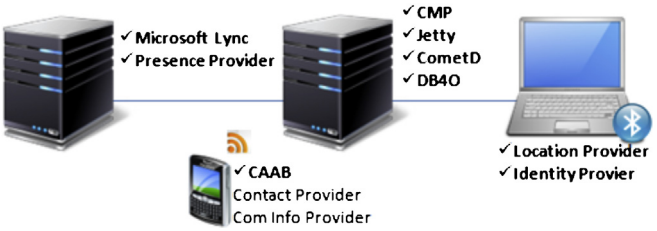

Fig. 9. CAAB deployment architecture.

Fig. 9 depicts the deployment architecture for CAAB. The central server runs a Microsoft Windows Server 2003 SP2 (Vendor: Intel, Model: Xeon, CPU: 2.8 GHZ, memory: 2 GB). A Jetty web server is installed on the server to run the Context Management Platform (CMP) web applications on top of the RESTlet[2] framework. CometD[3] is used as a notification server to send context updates to subscribed

[2] http://www.restlet.org/.
[3] http://cometd.org/.

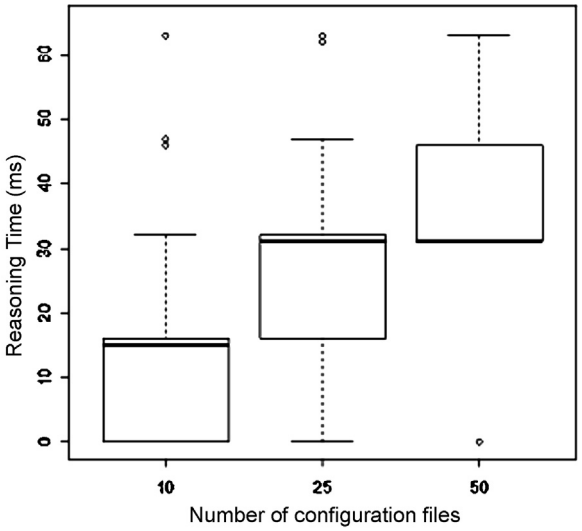

Fig. 10. Distribution of abstract–aggregate reasoning time.

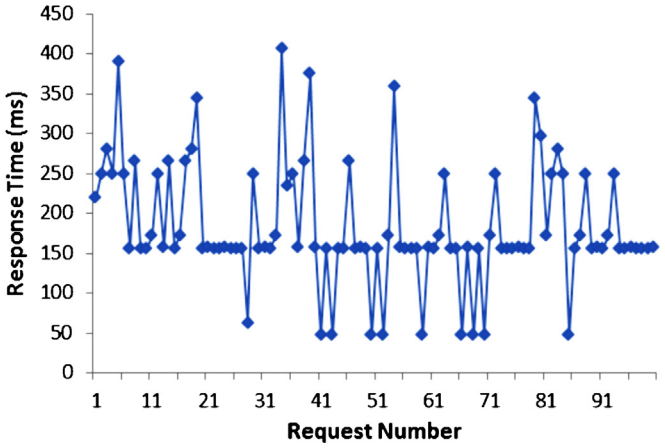

Fig. 11. Variation of response time.

clients. For data persistence, contextual information is stored on an object database, DB4O.[4]

Microsoft Lync is the unified communication solution. The Presence Provider is a UCMA 3.0 (Unified Communications Managed API) application that monitors the Microsoft Lync users' presence information. Both run on a single 64-bit Microsoft Windows Server 2008 R2 Enterprise SP3 (Vendor: Intel, Model: Xeon, CPU: 2.0 GHZ, and 2 GB of memory).

The Location Provider is an application that can detect nearby Bluetooth devices with the help of the Blucove[5] library. The Identity Provider is a web application that provides a directory service: mapping between the physical (Bluetooth) address of a device and the owner identifier, as well as mapping between the SIP (Session Initiation Protocol) URI (Uniform Resource Identifier) used by Lync and the user identifier. Both providers run on Microsoft Windows XP SP3 installed on a Dell D630 laptop (Vendor: Intel, Model: Core 2 Duo, CPU: 2.2 GHZ, memory: 2 GB).

Fig. 10 depicts a comparative summary of reasoning time distributions as a ratio of the number of CPDL configuration files uploaded to the context management framework. This figure utilizes box plots, a convenient means of data visualization that is especially useful for detecting the presence of extreme values in a distribution.

The different distributions are generated by sending approximately one hundred sample context publications to the framework and measuring how much time the framework spent processing each publication through the functions' abstract–aggregate. In a box plot, a darkened line is used to represent the median (50th percentile) of a distribution, the 25th percentile corresponds to the bottom side of the corresponding box, the 75th percentile is indicated by the box's top side, the distribution's 10th percentile corresponds to the box's bottom line, and the 90th percentile corresponds to the box's top line (McGill et al., 1978). Additional points or lines on the top or bottom of a box correspond to the distribution outliers, i.e., points showing the extreme values for a given distribution.

These box plots show that the median for 10 configuration files is about 15 ms, while it is about 30 ms for 25 and for 50 configurations, indicating a possible convergence in reasoning time that should be confirmed by further studies. The spread (the box height or the difference between the 75th and 25th percentiles) of the different distributions is very similar for each configuration grouping. Also, except for a few outliers, the reasoning time increases as the quantity of configuration files being managed increases.

The graph in Fig. 11 illustrates the variation in response times, in milliseconds, of the framework for multiple context publications. Data used to generate this graph are collected by sending multiple context publication requests to the framework and measuring the time to receive a response for each request. This response time includes the time required to establish the HTTP connection as well as the time needed for the framework to deal with this published information.

From the figure, we can easily see that the response time oscillates around 150 ms; reaches a maximum of 400 ms in extreme cases and a minimum of 50 ms for the best cases.

We can conclude from Figs. 10 and 11 that the response time of the proposed framework depends mostly on the time required for exchanging HTTP messages among the different elements of the framework. The reasoning time is minor compared to the connection time. An important cause of this latency in HTTP connections is due to the repetitive phases of TCP handshakes, as shown in Radhakrishnan et al. (2011). The framework performance, as far as context transportation, can be improved by the use of HTTP-persistent connections (Kozierok, 2005). The use of permanent connections will improve memory and CPU utilization related to the establishment of HTTP connections, as fewer connections will need to be opened and maintained.

6. Background and discussion

6.1. Related work

Our framework relies on the consideration of well-established design principles (separation of concerns and events processing) to provide better context management, enabling third-party developers to build flexible context-aware applications. Most of the earlier works lacked flexibility and did not take advantage of these principles; instead, they relied separately on just one of them. A selection of earlier works follows, grouped according to the main principles they are based on:

A dynamic approach to deal with context characteristics is to manage context lifecycle or validity duration to provide an efficient support to context-aware applications. Jih et al. (2009), proposed

[4] http://www.db4o.com/.
[5] http://code.google.com/p/bluecove/.

a simple context state transition to represent context lifecycles. Context can be 'active' as soon it is updated, 'suspended' when the activity performed by a user is disturbed, or 'terminated' when the user ends his/her current activity. Context evolves among these states as the result of capturing new values. The authors used this approach to provide a consistent repository of contextual information.

In some cases, the change in values of contextual information may not imply a change in the current situation. Thus, instead of waiting for changes to occur in the value of contextual information, it is more efficient to wait for relevant events that represent changes in a situation, as the relevance of an event may vary depending on the context a consumer needs. In Mo et al. (2011), the authors propose a UML-based and event-driven model for context-aware services based on two views: a context definition view (CDV) and a scene transition view (STV). The CDV represents static content such as context data and events triggered by changes in context. The STV models the dynamicity of a context-aware service by representing the different scenes (situations), the transitions between them, and the service behavior corresponding to a scene and that could be initiated as a response to a transition. Even though this proposed model is event-driven, the context-aware adaptation is provided for a given scene and not for an event. The preceding approaches propose to manage context via state machines to provide context-awareness via a reaction to changes in the context state. These state machines are predefined and cannot be customized dynamically for a given application.

The separation of concerns is one of the main design patterns in software engineering; it aims to capture redundant functionalities into a set of specialized components for better modularity and reusability. Few works have proposed the use of this concept to decouple context management from application logic. In Gutheim (2011), the authors presented a telecom platform that leverages NGN (Next-Generation Network) features to provide contextual information to mobile applications. The platform relies on a layered context management approach that uses third-party modules to infer knowledge from context data. The modeling approach is based on a predefined ontology, and application developers can supply their inference modules, which can also be reused by other developers. Context distribution relies on subscriptions to specific context data and notifications of when this data becomes available. Privacy management follows an installation-time permission approach.

In van Sinderen et al. (2006), the authors propose a layered infrastructure to support context-aware mobile applications. These applications may delegate the execution of some context-dependent actions to the infrastructure, such as the invocation of specific actions triggered by context events. The infrastructure also provides a support for privacy and trust management by allowing users to specify their privacy policies regarding which applications are authorized to use their context. The context processing mechanisms (ontological reasoning for deriving new facts, machine learning to generate high-level models from low-level context) are predefined and cannot be customized by context-aware applications, hence the generated context events may not respond to the specific needs of a given end application. Privacy management follows a usage-time permission approach.

In Zhu et al. (2011), the authors present a framework that aims to decouple context management from the application layer to simplify application implementation. This framework also supports any requirement changes related to context management, without impact on the application layer. The framework relies on an RDF-based model that enables developers to define their specific context management modules. Developers have to comply with the model and the semantic provided, which are used internally by the framework to manage context data, as well as with the corresponding management operations. The use of ontology-based reasoning creates a scalability issue as the framework performance decreases considerably (the response time may take many minutes) when the number of managed contexts increases. Privacy issues are not considered.

In most current permission systems, users are asked to grant permissions (Felt et al., 2011) to a given resource (e.g., location information) at the time of use (e.g., Apple iOS), or at the install time (e.g., Android). In the first approach, users are prompted to give their permission (permanently, for a period of time, or for a single use) to a resource when the latter requests a privileged access. In the second approach, applications declare their requirements regarding privileged access during the installation process. If the user grants permanent permission then the application is installed, otherwise it will not be installed. Both approaches are used in mobile platforms (e.g., Android, Bada) to protect user privacy (Felt et al., 2012) and to grant access to user data to third party applications.

Information that is characteristic of a context, also called the quality of context (e.g., accuracy, precision), is additional information that may be of great value to some context-aware applications, since it can be used to assess context-aware actions (e.g., the automatic handling of incoming calls). In McKeever et al. (2009), the authors propose a UML-based quality of context model to augment contextual information with meta-information describing its quality (e.g., precision, frequency, confidence) and then to propagate this quality after abstracting contextual information. A context management framework that relies on quality of context (QoC) information to select a context source from multiple providers was presented to reduce the propagation of errors resulting from abstracting low level context and then basing reasoning on it (Filho and Agoulmine, 2011). The framework relies primarily on the probability of correctness of a given context as a QoC parameter, and thereby reduces the propagation of errors that result from basing reasoning on low-level context.

6.2. Discussion

In most of the proposals for building context-aware systems, context consumers are asked to continuously query for context, which may cause massive overhead. GPS locations, for example, are frequently updated, which means many messages are sent to every component that subscribed to the location of an entity. After receiving contextual information, consumers must re-implement or implement their own modeling approach in order to have a consistent view of the overall entity situation over the whole system's life-cycle. Finally, consumers must implement their own reasoning mechanisms to adapt their behavior according to an entity's situation. If consumers run resource-constrained devices (e.g., smartphones), then managing context will lead to additional (and potentially significant) overhead in addition to that caused by the business logic of the application itself. Existing frameworks focus primarily on simplifying tasks related to context provisioning (e.g., acquisition, distribution) so that context-aware application developers can concentrate on application development. These frameworks provide low-level development support, with limited capabilities for defining custom context processing and reuse.

We propose a new approach where context-aware applications specify their own specific approaches via XML encoded messages, and where common context processing routines are shared among multiple applications. This framework meets the requirement introduced earlier, in Section 2.2. It enables a reduction in the complexity related to developing context-aware applications by separating context management functionalities from application logic. The HTTP protocol is used for context transportation, making

Table 2
Simulation parameters.

Providers	Rate (publication/s)	Consumers
CxP1	40 (p/s)	CxC3, CxC8
CxP2	20 (p/s)	CxC1, CxC3, CxC7
CxP3	30 (p/s)	CxC3, CxC5, CxC8
CxP4	20 (p/s)	CxC2, CxC3, CxC5,
CxP5	20 (p/s)	CxC1, CxC3, CxC4, CxC5, CxC8
CxP6	10 (p/s)	CxC1, CxC3, CxC5, CxC7
CxP7	10 (p/s)	CxC1, CxC3, CxC4, CxC5
CxP8	50 (p/s)	CxC2, CxC8
CxP9	20 (p/s)	CxC1, CxC2
CxP10	40 (p/s)	CxC1, CxC3, CxC5, CxC6, CxC7, CxC8
CxP11	20 (p/s)	CxC1, CxC3, CxC5
CxP12	20 (p/s)	CxC1, CxC3

the integration of the framework with any application an easy task. Information is exchanged based on XML, providing flexibility to developers in terms of information representation. The framework supports QoC information through the "select" function. It allows consumers to define, in a flexible way, what source of context to consider in case there are multiple sources based on quality information. In addition, the framework supports privacy protection by allowing users to track which applications are using their context, and to specify their privacy policies. However, having to enter policies manually may become tedious as this could require a continuous verification of how context is being used, and a policy update may be deemed necessary.

The framework response time is only slightly influenced by the reasoning time required for the published context; instead it depends more on the HTTP connection established between the context source and the framework. An important cause of this latency in HTTP connections is the repetitive phases of TCP handshakes (Radhakrishnan et al., 2011). The framework performance, in terms of context transportation, can be improved by the use of HTTP-persistent connections (Kozierok, 2005). This technique maintains an HTTP connection between the framework components much longer than the minimum required for one single exchange. The use of permanent connections will improve memory and CPU utilization related to the establishment of HTTP connections, as fewer connections will need to be opened. Permanent connections will also enable the pipelining of multiple-context exchanges, and especially of context publications, since the frequency of context collection at the context provider is much higher than the time required for processing them at the broker (see Table 2).

We summarize this discussion in the above summary table (Table 3) where evaluation criteria are based on the design considerations introduced in Section 2. The '+' symbol means the existence of a support for the corresponding consideration, whereas the '–' symbol indicates its absence, and '+/–' indicates that support exists but needs to be enhanced.

6.3. Lessons learned

The presented context management solution aims to facilitate the development and maintenance of context-aware applications. Its advantages can be seen from different perspectives dependently on the role of the agent (i.e., user, developer, or administrator) that may be involved in the interaction with the framework at any phase.

From the user perspective, the framework enables lightweight applications to monitor specific changes in his/her context to adapt their behavior appropriately and reduce the configuration effort that may be needed to customize traditional applications. In addition, the user can be involved in customizing how raw data are abstracted. For instance, he can be asked to tag some personal locations (such as his/her home) to allow the application to use these tags in the abstraction process, i.e., to abstract physical GPS values into correct high level information.

From the application developer perspective, the framework enables them to focus on creating the application business logic and to leave the complex task of context management to the framework. In fact, the developer is in charge of developing the context providers and consumers in addition to specifying the configuration file that will express the connections between the different components; it is the framework that is responsible for circulating context among these components. Furthermore, the specification of the context management through the use of XML documents enables a seamless management of the reasoning logic as it is easy for developers to evolve it merely by replacing the configuration file with a new version.

From the administrator perspective, the framework enables the easy deployment of context-aware applications and provides a powerful support by upgrading their context management. Moreover, the use of XML configuration files allows the administrator to customize some of the reference values used in the abstraction process, thereby making it possible to experiment with many values to find the most suitable reference.

Although the capabilities of the proposed framework are promising, it does not eliminate all the complexity of context management. For instance, the XML document describing the context processing may become very large (due to a high number of managed providers), making any update to such a document a tedious task. Furthermore, how the architecture components are distributed may affect the efficiency, for instance, in the case where a consumer is located on the same device as the provider there is no need for hosting the reasoning in a remote side broker.

Another drawback is related to the management of the XML configuration files. Currently, it is the developer who is in charge of manually uploading these files, which is error-prone and may become tedious. Another possibility is to embed the reasoning configuration file in the application so that it is uploaded when the application is installed on a user's smartphone.

Table 3
Comparaison summary.

	1. Ease of context manipulation	2. Ease of context reasoning management	3. Ease of application development	4. Ease of integration
Gutheim (2011)	+/–	+/–	+/–	–
Filho and Agoulmine (2011)	+/–	–	+	–
van Sinderen et al. (2006)	+/–	–	–	+/–
Jih et al. (2009)	+/–	+/–	+/–	+/–
Mo et al. (2011)	+	–	–	+
Zhu et al. (2011)	+/–	+	+/–	–
McKeever et al. (2009)	+/–	–	–	+/–
Our proposal	+/–	+	+	+

7. Conclusion

Context information is valid for a limited time, which makes it suitable to be handled as a series of events. We propose to provide context-aware applications with the ability to program a context-aware framework to handle context events and only notify an application with relevant information. An XML-based language is provided for this purpose. This approach is especially useful for applications running on resource-constrained devices that cannot process the many events generated by context changes. The benefits derived from the developed framework are reductions of both the cost and complexity related to the development of context-aware applications, since functionalities related to context management are no longer an application's responsibility. Another benefit is the adaptation efficiency of context-aware applications; they receive more accurate events of interest related to context changes and therefore only need to implement the logic that will respond to these events. The use of a local caching system enables response time enhancement. Nevertheless, our approach does not deliver substantial advantage to all kinds of context-aware applications. For example, a real-time supervision application that tracks the localization of a truck needs to receive all of that truck's GPS related events in order to maintain a correct visualization of the truck on a map.

As future work, we plan to build a Cloud-based context management platform that relies on the approach presented here to handle context events in large-scale distributed environments. We are also considering enhancing the framework with self-management capabilities to provide context-aware applications with the ability to dynamically modify how context should be processed.

References

Balinsky, H., Moore, N.C.A., Simske, S.J., 2011. Intelligent assistant for context-aware policies. In: 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Changsha, China.

Chihani, B., Bertin, E., Jeanne, F., Crespi, N., 2011a. HEP: context-aware communication system. International Journal of New Computer Architectures and their Applications (IJNCAA'11) 1 (1), 15–24.

Chihani, B., Bertin, E., Jeanne, F., Crespi, N., 2011b. Context-aware systems: a case study. In: Proceedings of International Conference on Digital Information and Communication Technology and its Applications (DICTAP'11), Dijon, France.

Chihani, B., Bertin, E., Crespi, N., 2011c. A Comprehensive Framework for Context-aware Communications Systems. ICIN, Berlin, Germany.

Chihani, B., Bertin, E., Crespi, N., Suprapto, I.S., Zimmermann, J., 2012. Enhancing existing communication services with context-awareness. Journal of Computer Networks and Communications 2012, 10 p.

Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E., 2011. Permission re-delegation: attacks and defenses. In: USENIX Security Symposium.

Felt, A.P., Egelman, S., Wagner, D., 2012. I've Got 99 Problems, But Vibration Ain't One: A Survey of Smartphone Users' Concerns. UC Berkeley Technical Report No. UCB/EECS-2012-70.

Filho, J.B., Agoulmine, N., 2011. A quality-aware approach for resolving context conflicts in context-aware systems. In: IFIP 9th International Conference on Embedded and Ubiquitous Computing (EUC'11), Melbourne, Australia.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, ISBN 0-201-63361-2.

Gutheim, P., 2011. An ontology-based context inference service for mobile applications in next-generation networks. IEEE Communications Magazine 50 (1), 60–66.

Hamadache, K., Lancieri, L., 2010. Towards ontological model accuracy's scalability: application to the pervasive computer supported collaborative work. In: Proceeding of International Conference on Machine and Web Intelligence (ICMWI'10), Algiers, Algeria.

Jih, W., Huang, C.C., Hsu, J.Y., 2009. Context life cycle management in smart space environments. In: AUPC'09, London, UK.

Kozierok, C.M., 2005. The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference. No Starch Press, ISBN 1-59327-047-X.

McGill, R., Tukey, J.W., Larsen, W.A., 1978. Variations of box plots. American Statistician 32 (1), 12–16.

McKeever, S., Ye, J., Coyle, L., Dobson, S., 2009. A context quality model to support transparent reasoning with uncertain context. In: 1st International Workshop on Quality of Context (QuaCon), Stuttgart, Germany.

Mo, T., Li, W., Chu, W., Wu, Z., 2011. An event driven model for context-aware service. In: 2011 IEEE International Conference on Web Services (ICWS'11), Washington, DC, USA.

Naudet, Y., 2011. Reconciling context, observations and sensors in ontologies for pervasive computing. In: Sixth International Workshop on Semantic Media Adaptation and Personalization (SMAP'11), Vigo, Spain.

Oh, Y., Han, J., Woo, W., 2010. A context management architecture for large-scale smart environments. IEEE Communications Magazine 48 (3), 118–126.

Plesa, R., Logrippo, L., 2007. An agent-based architecture for context-aware communication. In: 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07), Niagara Falls.

Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., Raghavan, B., 2011. TCP fast open. In: The 7th International Conference on Emerging Networking EXperiments and Technologies (ACM CoNEXT'11), Tokyo, Japan.

Schmidt, A., 2006. A layered model for user context management with controlled aging and imperfection handling. In: Modeling and Retrieval of Context. Springer, Berlin, Heidelberg, pp. 86–100.

Simoes, J., Magedanz, T., 2011. Contextualized user-centric multimedia delivery system for next generation networks. Telecommunication Systems 48 (3/4), 301–316.

Song, S., Moustafa, H., Afifi, H., 2010. Personalized TV service through employing context-awareness in IPTV/IMS architecture. In: Proceedings of Third International Workshop on Future Multimedia Networking (FMN'10), Kraków, Poland.

van Sinderen, M.J., van Halteren, A.T., Wegdam, M., Meeuwissen, H.B., Eertink, E.H., 2006. Supporting context-aware mobile applications: an infrastructure approach. IEEE Communications Magazine 44 (9), 96–104.

Zhu, J., Pung, H.K., Oliya, M., Wong, W.C., 2011. A context realization framework for ubiquitous applications with runtime support. IEEE Communications Magazine 49 (9), 132–141.

Bachir Chihani is a PhD candidate at Telecom SudParis and has been an R&D engineer at Orange Labs since October 2010. He obtained his engineering degree with honors from the Ecole Supérieure d'Informatique, Algeria in 2008. He received his MSc degree in Networks and Telecommunication with honors from ENSEEIHT in 2010. His research interests are context-aware services and pervasive computing.

Dr. Emmanuel Bertin is currently Senior Service Architect at Orange Labs and Adjunct Professor at Telecom SudParis. He holds a Master's degree (diplôme d'ingénieur) from Telecom Bretagne's graduate engineering school and a PhD from Paris 6 UPMC University. He has been working at Orange Labs since 1999 in the fields of IP communication services and enterprise architecture. His expertise lies in service engineering and web-based services, and he especially focuses on service composition in open environments. He has published more than 40 papers in international journals and conferences, and holds several patents in the area of next-generation services.

Prof. Noël Crespi holds Master degrees from the Universities of Orsay and Canterbury, a diplome d'ingénieur from ENST-Telecom ParisTech, a PhD and a Habilitation from Paris VI University. From 1993 he worked at CLIP, Bouygues Telecom and then at France Telecom R&D in 1995. In 1999, he joined Nortel Networks as Telephony Program manager. He joined Institut Telecom in 2002 and is currently professor and Program Director, leading the Service Architecture Lab. He coordinates the standardization activities for Institut Telecom at ITU-T, ETSI and 3GPP. He is also an adjunct professor at KAIST and is on the 4-person Scientific Advisory Board of FTW, Austria.