RestContext
A Service Framework for Context Retrieval

Leon O. Burkard, Andreas C. Sonnenbichler and Andreas Geyer-Schulz
Karlsruhe Institute of Technology, Institute of Information Systems and Marketing, Kaiserstrasse 12, 76131, Karlsruhe, Germany

Keywords: Context Service, Distributed, Autonomous, Restful, XMPP, Dynamic Software Architecture, Interoperability.

Abstract: Todays context frameworks provide solutions for context mechanisms for individual applications only: context aware working spaces, easier mobile development frameworks or higher-level context abstractions. RestContext solves this problem with a service logically separating context as a set of information that can characterize a situation from further context interpretation mechanisms. RestContext is a resource oriented architecture which manages sensors of different types. A context may consist of sub-contexts as well as sensors that are linked to one or many contexts. With the help of RestContext it is possible to create topologies of contexts. Sensors can interact with context instances by push and pull mechanisms. We demonstrate how RestContext can be used in a distributed weather forecasting example.

1 INTRODUCTION

Today's mobile (and internet) services depend increasingly on context information. A context is the characterisation of a situation which consists of data that in sum describes the situation. Contexts can change over time due to changes in the data describing the situation. Based on the context, predictions about future behaviour can be made. A context can be described as a vector space $C_{s,t}$ that describes a situation s related to time t. Each vector $v_{i,t}$ stands for the values of sensor device $i \in I$ at time t. The value of $v_{i,t}$ can be of any type:

$$C_{s,t} = \begin{pmatrix} v_{1,t} \\ v_{2,t} \\ v_{i,t} \\ \dots \\ v_{I,t} \end{pmatrix}$$

For example, sensor 5 measures a temperature of 28.8 °C and a humidity 70% at $t = 0$. This results in $v_{5,0} = (28.8; 0.7)$. Agent-based systems can be used to describe the process of collecting data and acting upon the situation. According to Russel and Norvig, an agent is "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" (Russell et al., 1999, p.34). From this definition we see that agents receive input from so-called sensors. Various agents may receive data from different sensors. Sensors retrieve different kinds of data (e.g. geo-location, speed, temperature, humidity) as $v_{i,t}$. All (relevant) data received from the sensors $i \in I$ for an entity s in a time period t we call "context" $C_{s,t}$. Agents process information they receive from sensors and other agents. Additionally, agents may relay information to other agents and, thus, can be used for building up structured agent networks (with certain topologies). Depending on their internal agent's logic, actuators can then be used to influence the environment.

A definition of a topology is given by Bourbaki (Bourbaki, 1998, p. 17): A topology on a set $\mathfrak{X}$ is a structure given by a set $\mathfrak{D} \in \mathcal{P}(\mathfrak{X})$ with

$$\forall d_1, d_2 \in \mathfrak{D} : d_1 \cup d_2 \in \mathfrak{D} \wedge d_1 \cap d_2 \in \mathfrak{D}$$

In reference to agent-based systems the set of all sensors x_i is $\mathfrak{X}$. A context is then a subset d_i of $\mathfrak{D}$ with $\mathfrak{D}$ being a topology on $\mathfrak{X}$. Topologies are especially useful in distributed environments where a high number of elements (e.g. sensors) is used.

Context data from sensors is an important feature for mobile devices like smartphones. E.g. geo-location information is used for services like restaurant and hotel recommendations, navigation or additional information for search queries. Another example is the collection of weather data. Let us consider the infrastructure necessary to collect weather data for a certain area like the federal state of Baden-Wuerttemberg. Baden-Wuerttemberg is split into sev-

O. Burkard L., C. Sonnenbichler A. and Geyer-Schulz A..
RestContext - A Service Framework for Context Retrieval.
DOI: 10.5220/0004505502210232
In *Proceedings of the 4th International Conference on Data Communication Networking, 10th International Conference on e-Business and 4th International Conference on Optical Communication Systems* (ICE-B-2013), pages 221-232
ISBN: 978-989-8565-72-3

eral counties. In each county, at certain locations weather data is measured (temperature, wind speed, wind direction, humidity, air pressure, ...). These sensor data are reported through certain hubs (e.g. county-wise) to a central data storage (for the whole of Baden-Wuerttemberg). The data collection of all sensor data at all locations is then used to create longitudinal data rows in order to forecast the weather. In this example, each specific sensor matches the general sensor definition of Russel and Norvig.

In practice, sensors and agents use proprietary and specialized implementations to communicate with each other. No abstraction layer or generalized communication infrastructure is used. In the worst case this requires a specialized communication infrastructure and protocol for each type of sensor. This approach is error-prone and limits applications to benefit from a broader usage of distributed data.

Context frameworks are abstractions that help to generalize the communication: The introduction of a context framework standardizes the communication between sensors, hubs, and agents. In our weather data example, a temperature sensor uses the same communication protocol to communicate with a county hub as the hub does to communicate with the state hub. The same is true for a humidity sensor. Further, sensor data can be organized in contexts. E.g. all weather data collected at weather station "Karlsruhe-Center".

Instead of a direct communication with each sensor, a context is defined. The context is utilized for further communication. This abstraction omits detailed and specialized knowledge necessary if only proprietary communication is used. E.g. each sensor type requires a different protocol. Without a context framework it would be necessary to implement these individual communications for every new request type like wind, air pressure, etc. If a new generation of temperature sensors is introduced in the field without a context framework, each individual communication with the new sensor has to be adapted. By using a context framework, the communication stays the same, only internally the communication with the new sensor type has to be adapted. Thus context frameworks are useful to increase the usability on the device and the service side.

On the sensor side, a generic interface to transmit data from sensors reduces faults, overhead in development and platform specific implementations regarding the communication with the environment. New services like location-based emergency services (Geyer-Schulz et al., 2011) are also possible by using context-enabled applications.

As the amount of data that has to be processed can become large e.g. for weather forecasting, a context framework should provide the ability to build up topologies in order to balance the load. Therefore, agents should operate individually and offer the possibility to link them together. Additionally, as the definition of context is based on the collection of data forming information to characterize a situation, a generic approach to define a context is preferred. For example, to create a weather forecast for the federal state of Baden-Wuerttemberg we need to request information from many weather stations with various sensor types. The same infrastructure should also be usable for flood forecasting which uses other sensors.

In most cases, context frameworks provide only the ability to listen for updates from sensors and do not give the ability to request sensors for updates or status information like battery voltage. A possibility to control sensor nodes is desirable.

We present a context framework, that enables a distributed context service which communicates via Extensible Messaging and Presence Protocol (XMPP) (Saint-Andre, 2011) messages with sensors. It is possible to push sensor data from the sensor device to one or many context-service instances as well as to pull requests from context-service instances to the sensor node in order to get updates. On the sensor node an abstraction layer to the hardware is executed. It runs a XMPP-client that registers to the service-context and communicates via Extensible Markup Language (XML) based messages with the context-service instance. On the context-service side a RESTful web-interface (Representational state transfer, see section 4.2 for a more detailed explanation) offers the flexibility to combine subscribed sensors to contexts. Communication from the application side to the context utilizes Hypertext Transfer Protocol (HTTP) requests. Multiple context service instances can be linked together within contexts in order to achieve distributed sensor data handling. Each context service instance runs autonomously.

Our paper is structured as follows: In section 2 a definition of the term context-awareness is introduced. Also a classification of abstraction levels between a sensor and an agent are presented. Section 3 highlights context frameworks that inspired the work on our RestContext service. The main concepts of the new context service are introduced in section 4 followed by the implementation of our prototype in section 5. At last we discuss in section 6 possible enhancements for the RestContext service.

2 CLASSIFICATION OF CONTEXT SERVICES

In this section we define the term “context-awareness” and give a short overview about different architectures for context-aware systems.

2.1 Context and Context-awareness

M. Baldauf (Baldauf et al., 2007) describes different design principles and context models for context-aware systems and compares several frameworks and middleware technologies. At first he discusses definitions of the term *context*. One of the most accurate definitions in his point of view is given by Dey and Abowd: ”We define context as any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object” (Dey and Abowd, 2000, p. 3). We follow this definition: Our RestContext service perceives a context as a collection of different sensor-data at time *t* which can be enhanced by further information and sub-contexts that are contexts themselves as introduced in section 1.

Dey and Abowd also define *context-awareness* or *context-aware computing* ”as the use of context to provide task-relevant information and/or services to a user” (Dey and Abowd, 2000, p. 11). To summarize, Dey and Abowd define ”context” simply as a collection of information. In contrast, ”context-awareness” refers to the application utilizing contexts.

Our RestContext service follows the definition of ”context-awareness”. Additionally, it offers advanced functionality: it also enables the definition of a context as a collection of sensors or other contexts. This allows to create hierarchies of contexts and/or sensors. These contexts may be of different types and types may be mixed. In our prototype implementation, each contexts is represented by a uniform resource locator (URL).

2.2 Architectures for Context-aware Systems

According to H. Chen (Chen, 2004, pp. 16 – 18), three architectural styles can be distinguished to acquire contextual information:

Direct Sensor Access. In this style, a sensor is accessed directly by an agent through a proprietary protocol making not utilizing an abstraction layer. Advantages are high performance for gaining and processing sensor data. Hardware requirements are low. Direct sensor access requires a device-driver and a communication protocol for each version of a sensor type for each manufacturer.

Middleware Infrastructure. This architectural style encapsulates hardware access in a common software interface for easier requests for sensor data. Middleware components are ”typically built into the hosting devices or platform on which the context-aware applications operate” (Chen, 2004, p. 17). A middleware infrastructure has higher resource requirements for the hosting service.

Context Server. Multiple clients may access a remote data source via a context server. The context server need not be located on the sensor hosting hardware. It is an extension to a middleware infrastructure located on the sensor node responsible for the communication with the sensor. Clients access the context server in order to retrieve sensor data, resource intensive operations are done by the context server. Concurrent access by many users is possible because of less hardware resource limitations which also results in an improved scalability.

The decision for one architectural style depends on the location of sensor units, amount of possible users, hardware requirements and extensibility (Baldauf et al., 2007).

3 RELATED WORK

In this section, we describe existing middleware frameworks, context servers and a protocol for weather data that inspired the work on our RestContext-service.

The Gaia Project. The Gaia project (Roman et al., 2002) builds up an operating system ”but at another level of abstraction” (Roman et al., 2002, p. 75). It distinguishes between ”physical” and ”active” spaces. Physical spaces are geographic regions containing physical objects like network devices. Active spaces are extended physical spaces coordinated by context-based software in a geographic region. Sessions in active spaces allow mappings between applications and user data, supplied by an own context file system (CFS).

The Gaia system uses a three layer architecture as depicted in figure 1: the ”Gaia kernel”, an ”application framework” and ”active space applications”. Access to the hardware is accomplished by the component management core, offering basic services to the layer above. These basic services are called the ”Gaia

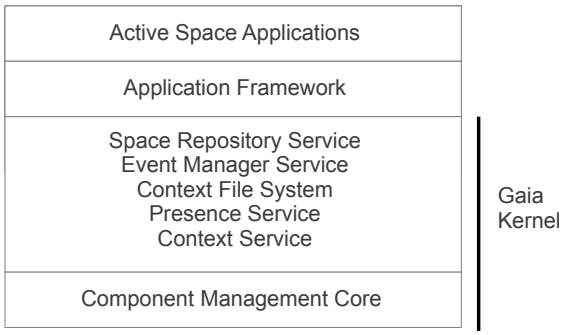

Figure 1: Gaia Architecture.

Kernel" and are part of the Gaia operating system. An application may query and register for particular context information to adapt to user behaviour and activities. "Active space applications" adapt applications to an active space like a conference room. Focus of Gaia is the execution of user-centric applications like a multi-device presentation tool adapting to the current users location and devices. Roman et al. (Roman et al., 2002) present as an application of the Gaia operating system a presentation manager: Each presenter has an own profile. This profile is loaded as soon as the person enters a Gaia enabled presenter room. With the help of the profile personal shares are mounted as one example.

Although the Gaia system fits into the definition of context and context aware systems, it has a fundamentally different focus as a RestContext architecture. Gaia's aim is to create context aware working places that add an adaptive environment in relation to users. The objective of RestContext on the other hand is the efficient distribution and aggregation of sensor data.

The Context Toolkit. Another solution is suggested by Salber et al. (Salber et al., 1999) and Dey et al. (Dey et al., 2001): As smartphones and ubiquitous computing in general have given users the expectation that information should be accessible everywhere at every time, the main objective of the "Context Toolkit" is the support of such context-aware computing systems. With the help of the "Context Toolkit", context-aware applications should be developed in an easier way. Therefore, the three layer architecture shown in figure 2 is used:

Widgets. They have a mediator role between users and an environment containing sensors. By providing an uniform interface to the underlying hardware, details of the context-sensing mechanisms are hidden. Polling and subscribing mechanisms are available.

Aggregators. These are called meta-widgets by Salber et al.. They are an extension to widgets by providing all features of widgets plus aggregation mechanisms for context information.

Interpreters. In this architectural part low-level sensing data is combined in order to produce high level information e.g. location or identity. Enrichment or creation of new information is also possible. The following example is given by Dey et al. (Dey et al., 2001): To detect if a meeting is taking place in a conference room, a combination of a sound level, identity and locations sensor is necessary as a combined entity.

Communication is done via HTTP by all components of the architecture. The main focus of the Context Toolkit focus are context-aware applications like special mailing lists mailing only to subscribers being located in a specific building.

Although composition of contexts to a new context is possible, the Context Toolkit is less feasible than RestContext. Widgets as mediators are compiled entities that can not be changed without a new roll-out of a new compiled version. In RestContext a context is created by adding XMPP ids or URLs of other contexts to a list. This offers the possibility to create and change contexts in an easy way even while the system is running. Also the Context Toolkit is missing functionality to control sensors e.g. adjusting parameters in a measuring unit by the design of their widgets.

Hydrogen. Hydrogen (Hofer et al., 2003) uses a three layer architecture as presented in figure 3.

The bottom layer is the "Adaptor Layer" which is responsible for getting sensor value data and to communicate with the "Management Layer". It offers simultaneous access to the hardware through one context instance instead of direct sensor access that uses locking mechanisms. Sensors are directly connected to this layer. The "Management Layer" provides and retrieves contexts by interacting with the "Adaptor Layer". In the "Management Layer" the context server is located. It offers asynchronous (querying

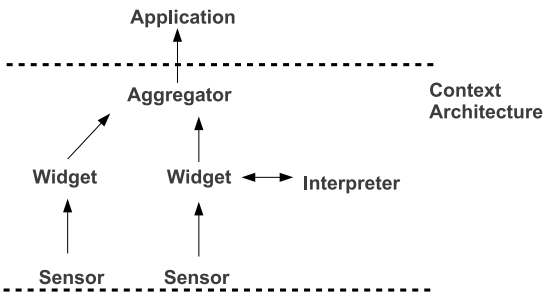

Figure 2: The Context Toolkit Architecture.

specific contexts) and pull-based interaction. Applications that use the context server in the ”Management Layer” are part of the ”Application Layer”. To be robust against network connection difficulties, all three layers are deployed on the sensor hardware. To exchange contexts with the environment, a peer-to-peer mechanism is used. For each type of context (located in the ”Adaptor Layer”) an extended version of a ”Context Object” class being the base class for all context objects has to be developed. This is done by implementing two abstract methods toXML() and fromXML() for non-Java applications to ensure interoperability.

Having located all three layers on the mobile device Hydrogon's approach is problematic in terms of data distributed systems like weather forecasting. Due to this fundamental design decision to focus on local mobile hardware, Hydrogen is lacking the possibility of different views on one sensor e.g. one view on a water level sensor that does updates of its values every hour and is free to use and another view that updates every minute but isn't for free.

CASS Middleware. The CASS-Middleware (Context-awareness sub-structure)(Fahy and Clarke, 2004) is a server based middle-ware with focus on support for ”higher-level context abstractions and is both flexible and extensible in its treatment of context” (Fahy and Clarke, 2004, p. 1).

Applications communicate with the CASS middleware and not directly with sensors. The CASS is a centralized service. Sensor nodes (located on the sensor hardware or computers directly attached to them) are connected, as depicted in figure 4, to a server-based middleware that can offer more memory and processor resources. CASS offers caching abilities for applications to reduce temporary connection issues. A ”knowledge base” contains rules to map sensor values to context-aware keys, e.g. weather sensor data indicating bad weather to recommend indoor activities. In the CASS middleware a ”sensor listener” is responsible for getting low-level sensor data from a pushing mechanism on the sensor node side. A ”change listener” is connected to the SensorListener to inform subscribed hand-held computers on

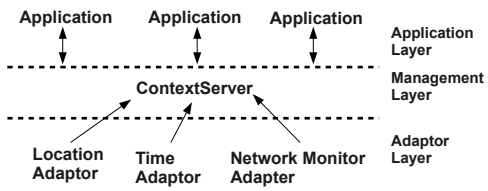

Figure 3: The Hydrogen Architecture.

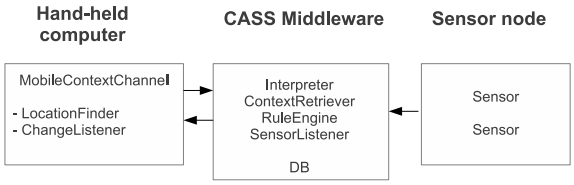

Figure 4: CASS Architecture.

changed contexts.

The CASS framework lacks the possibility of building up topologies of sensor and other middleware nodes as by design the CASS middleware is provided by a single server instance. In the presented version of the CASS middeleware only pushing of values from sensor nodes to a SensorListener instance is supported, pull mechanisms to retrieve data and control sensors are missing.

OpenWeather. A. Yanes suggests a peer-to-peer weather data protocol (OpenWeather) to provide a standardized way to transmit weather data (Yanes, 2011). OpenWeather is a protocol based on TCP using the protocol's features such as error and flow detection. Its aim is to break bottlenecks in scenarios of these days automated weather stations (AWS) that use proprietary or standard protocols like Server Message Block (SMB) (Microsoft Corporation, 2013) and FTP for the transfer of weather data to remote destinations. Problems are e.g. the missing real-time transfer of weather data, intermediary conversion steps before data transfers, performance issues regarding fetching, transmitting as wells as manipulating data and in general that no protocol takes advantage of the capabilities of AWS. However, OpenWeather “does not implement an aggregation technique between the nodes” (Yanes, 2011, p.87), although the author states that OpenWeather can be extended in such a way. Although OpenWeather presents the possibility of routing HTTP requests through a protocol bridge, OpenWeather is limited in terms of creating contexts using wide spread standard protocols, because the application domain is restricted to weather stations and not sensors in general. The objective of RestContext is to present a generic approach of generating and distributing contexts while using any kind of sensor and being a basis for specific adaptations.

4 ARCHITECTURE OF THE RestContext SERVICE

In this section we present a generic layout for a context service. We provide requirements followed by a

Table 1: Comparison of the presented projects and RestContext.

	Gaia Project	Context Toolkit	Hydrogen	CASS	RestContext
Requirements	Operating system for context aware working spaces	Support of context aware computing systems	Context awareness for devices with limited resources	Server based middleware that support context awareness	Efficient distribution and aggregation of sensor data
Lightweightness					
Extensibility					
Robustness		n.i.			
Distribution					
Interoperability					
Genericity	n.i.	n.i.		n.i.	
Two-way communication	n.i.				
Different Levels of Precision					
Interpretation of the context					

– not satisfied – partly satisfied – completely satisfied n.i. – no information

generic architecture and the underlying main concepts of the RestContext service. In section 5 we present details about the concrete implementation of a prototype of the RestContext architecture.

4.1 Requirements for a Context Service

We determined the following set of requirements for building a generic context service by analyzing the following context-aware systems from the literature: the CASS middleware (Fahy and Clarke, 2004), Hydrogen (Hofer et al., 2003), ContextToolkit (Salber et al., 1999). The first three requirements are from (Hofer et al., 2003). (Fahy and Clarke, 2004) share the requirement of robustness. The requirements distribution, interoperability, genericity and two-way communication were identified by evaluating the existing architectures. Altough not all systems identified these requirements explicitely. The Context Toolkit (Salber et al., 1999) is missing a dynamic approach for adding and removing elements. On the other hand the CASS middleware (Fahy and Clarke, 2004) does not offer the possibility to pull values from a sensor node.

Lightweightness. The architecture has to deal with limited resources and low hardware requirements on the sensor hardware side.

Extensibility. New context elements should be added and removed easily to existing contexts. Connections to several remote sensors should be possible.

Robustness. Disconnections, temporary connection issues or other disruptions of a few sensor devices should not break the context service.

Distribution. Since collecting and managing sensor data of a large amount of sensors requires high resources on the server side, a distributed architecture with autonomic instances of a context-service is preferred.

Interoperability. Standard communication protocols should be supported to achieve uniform access by different applications to the context service.

Genericity. Context adding and removing ought to be done while running the system. The architecture should not be limited to special types of context.

Two-way Communication. Sensor data should be delivered both by push and pull mechanisms. Rudimentary sensor management can be provided by a managing instance on the context service side.

Different Levels of Precision. One sensor ought offer different levels of precision e.g. for various use-cases, pricing-models or security settings. For example the National Association of Securities Dealers Automated Quotations (NASDAQ) offers different pricing models regarding the time resolution of stock market prices. Another example is the former used selective availability[1] feature of

[1] http://www.gps.gov/systems/gps/modernization/sa/

the U.S. government that degraded the GPS signal “for national security reasons”.

A comparison of the support of the requirements by the Gaia project, the context toolkit, hydrogen, the CASS middleware and RestContext is presented in table 1.

4.2 Architecture

To achieve *lightweightness* of the architecture, communication with the sensor hardware takes place via the Extensible Messaging and Presence Protocol (XMPP) (Saint-Andre, 2011). For this protocol, many client-libraries are available[2]. They can also be used in embedded systems with low hardware resources. Because of these features, XMPP can be used for the communication between a context service and sensor hardware. XMPP is a protocol standardized by the Internet Engineering Task Force (IETF) that results in a non proprietary protocol stack. It is a mature mechanism to exchange messages (nearly) in real-time. Messages are based on XML, so standard toolkits can process them easily. XMPP has a broad industrial base (e.g. Google, Facebook). By using namespaces custom data messages can be created. There are various open source XMPP servers availabe[3] that can be used to build up a private network of clients which is not accessible to everyone.

Interoperability is achieved by the usage of the HTTP protocol on the node-side in a resource oriented way (REST). REST is an abbreviation for ”Representational State Transfer” and was introduced in the dissertation of Roy T. Fielding (Fielding, 2000). All communication between distributed context services as well as client applications to the context service is done through HTTP requests. Caching mechanisms (e.g. saving the last sensor value each) and status codes of the HTTP protocol help to achieve the requirement of *robustness* sufficiently.

Figure **??** gives an outline of the RestContext architecture: A *SensorView* is a layer on top of the direct sensor access mechanism. It is responsible for delivering sensor data to a RestContext instance and receiving XMPP commands. Commands are sent by chat-messages that follow the pattern `command parameters/data` . A SensorView registers to one or more RestContext instances by the `hello` command via the XMPP protocol. The parameter of the `hello` command is a `session-Id` , identifying the view on the sensor. On the one hand, it offers the possibility to have different views on one sensor e.g. one id for temperature with a precision of five degrees, another id for an accuracy to a tenth (requirement *Different levels of precision*). On the other hand it is a shared secret between the SensorView and a RestContext instance of trust to ban foreign access to the SensorView. Further parameters to the `hello` command are the type of the sensor and a meta data field for details and status about the sensor hardware. Each request to the SensorView instance has the shared secret as parameter. The sessionId can be updated by the `helloupdate` commmand.

To support the requirement of *two-way communication* from section 4.1, a SensorView may push values by the `myvalues` command and react on pull requests by the `getcurrentvalues` command. To get current information about the sensor device the `getinfo` command is used.

The *RestContext* instance has two communication interfaces: The XMPP communication interface with SensorView instances and the communication interface via HTTP for communication with other RestContext instances and applications. There are two main tasks for a RestContext instance:

1. Offer communication abilities over HTTP from applications and RestContext instances to SensorView instances and other RestContext instances.
2. Manage a collection of contexts that consists of local or remote contexts and SensorViews. Local means that the SensorView instance or context is located on the RestContext instance in question. Remote contexts and SensorViews on the other hand are not located on the related RestContext instance and therefore have a different base URL.

RestContext owns a pool of registered SensorView instances accessible by an URL. Over the unique URL of the sensor other RestContexts may access the SensorView instance and both get data from the device and send requests (e.g. to update sensor value data) to it. For each SensorView instance, currently the last sensor value is saved in the RestContext instance. The technical implementation of a context is a collection of URLs. These URLs can be hyperlinks to local or remote SensorView instances. They can also be hyperlinks to local or remote contexts being part of RestContext instances that manage own collections. By using lists of links, different RestContexts can be composed in tree structures to fulfill the requirements of *distributed* contexts, *extensibility* and *genericity*. The link structure can return current values of all context elements by using recursive requests to all sub-hyperlinks. Also recursive updating of all data is possible. To prevent cycles in RestContexts that are linked together, each sub-request adds its

[2]http://xmpp.org/xmpp-software/libraries/

[3]e.g. , http://www.ejabberd.im/

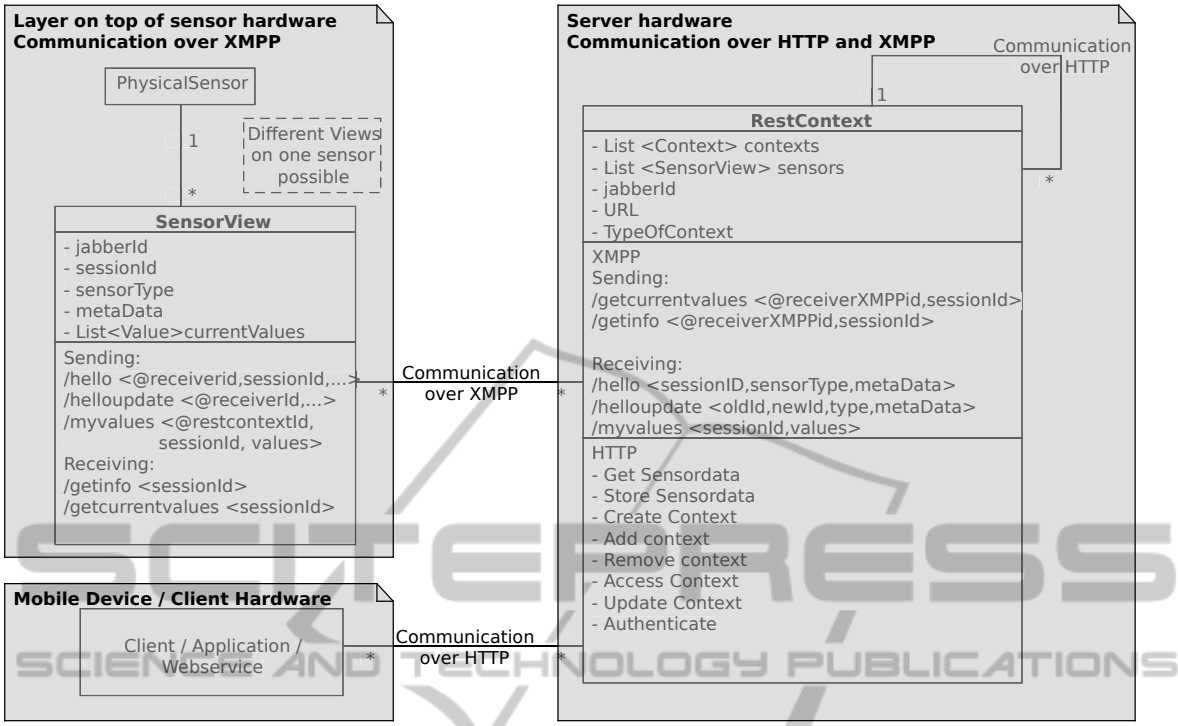

Figure 5: The RestContext Architecture.

URL to a list in the request body of the HTTP request. Each instance of RestContext checks the list to detect loops in the request chain. If a cycle occurs, instead of a value structure, an error message is returned.

Context-aware applications can access and modify the RestContext service by HTTP requests. Advantage of using the HTTP protocol are the widespread and robust HTTP-client implementations (e.g. Apache HttpComponents[4]). A robust implementation of an HTTP-client is available for nearly all devices, especially smart phones and embedded systems.

5 PROTOTYPE

In section 4.2 we outlined the architecture and main concepts of RestContext. In this section we present a concrete implementation of the context service as a prototype.

The URL interface structure of our RestContext implementation is presented in table 2. The implementation follows the RestContext architecture (presented in section 4), shown in figure 5.

Each SensorView instance is accessible by its own URL, can return values via the HTTP-GET command and also update values by HTTP-POST requests.

The update mechanism works as follows: A POST request is received by the RestContext instance. The instance sends an update request via XMPP to the corresponding sensor (with the known session-id parameter). While no updated values are received, a GET request to the update values resource returns a status information. Once the SensorView instance updated its values, the status information changes as demonstrated in figure 6. A context under a given {id} can be modified by POST requests to add items, DELETE requests for deleting items and GET requests to retrieve items.

All resources have a representation in the JSON data format. Each resource has various links to possible further actions e.g. by requesting a SensorView instance by an id. Links to the `updateinfo` , `updatevalues` and `values` resources are also returned.

The prototype has been implemented in the programming language Python by usage of the Python frameworks web.py[5] and jabberbot[6]. A modified version could also be deployed to the Google App Engine[7] (using their xmpp-library), so RestContext instances may be executed on a cloud infrastructure.

[4]http://hc.apache.org/

[5]http://webpy.org/

[6]http://thp.io/2007/pythonjabberbot/

[7]https://developers.google.com/appengine/

Table 2: URL interface structure of RestContext.

URL	Description
http://host/registeredsensor	List of all registered SensorView instances.
http://host/registeredsensor/{id}	URL of a SensorView with id {id}.
http://host/registeredsensor/{id}/values	Representation of current values.
http://host/registeredsensor/{id}/updateinfo	Sending request for sending updated status and meta information of the sensor over XMPP.
http://host/registeredsensor/{id}/updatevalues	Sending request to SensorView instance for updating values (pull-request) over XMPP.
http://host/information	Common information about the RestContext service instance like XMPP-id, uptime, status. etc.
http://host/context	List of all created contexts.
http://host/context/{id}	URL of context with id {id}.
http://host/context/{id}/elements	Managing structure for all elements associated with this context.
http://host/context/{id}/values	List of all cached values of the associated context.
http://host/context/{id}/updatevalues	Sending an update sensor data request to all elements associated with this context.
http://host/context/{id}/updateinfo	Sending an update sensor information request to all elements associated with this context.

5.1 Use Case

Figure 7 shows a use-case scenario for the RestContext service. Based on this use-case we will demonstrate our implementation of the RestContext architecture. The use-case offers meteorological services for the federal state Baden-Wuerttemberg with two counties. Each county has locations, shown by location circles in the figure. There exist sensors that belong to geographical regions. There is also a storm forecasting service with its own sensors, a sensor registered at the *location 2* RestContext instance and a sensor *temperature 4* that has registered to two RestContext instances. All circles with no background color run instances of the RestContext service and are reachable by the dummy URL `http://nameofcircle/` and XMPP-Id `xmpp:nameofcircle` for example `http://location1/` resp. `xmpp:location1` . Sensors are all filled circles and addressable by the XMPP-ID `xmpp:nameofsensor` .

Setting Up the Infrastructure.

- `xmpp:temperature1` registers to `xmpp:location1` with a message *hello 123 temperature somemetadata*.
- All other instances of SensorView are doing the same to their corresponding RestContext service. The sensor `temperature4` registers to both RestContext instances `xmpp:stormforecast` and `xmpp:location3` .
- A context (e.g. `http://stormfc/context/1/`) on the storm forecast node adds sensor temperature 3 by adding the url `http://location2/registeredsensor/temperature3` to one or many of its contexts.
- In order to build up a topology, a context on node county1 (e.g. `http://county1/context/2`) adds the context of location 1 (e.g. `http://location1/context/3`) that refers to temperature 1 (`xmpp:temp1`). The next level of the topology is constructed by adding the context url (e.g. `http://county1/context/2`) of county 1 to a context url of `http://bawue/context/id` .
- If the storm forecast context doens not require the sensor temperature 3 any more, it erases the url of the context refering to temperature 3 (e.g `http://location2/registeredsensor/temperature3`) from its list of elements (e.g. `http:// stormfc/context1/`).

Requesting the Infrastructure.

- If you are interested in the current temperature value of *location 1*, you request the url: `http://location1/context/ {1}/values`
- The sensor *temperature 1* pushes current values to the RestContext location1 periodically.
- To update sensor values aperiodically of the sensor *temperature 1*, you do a POST request to

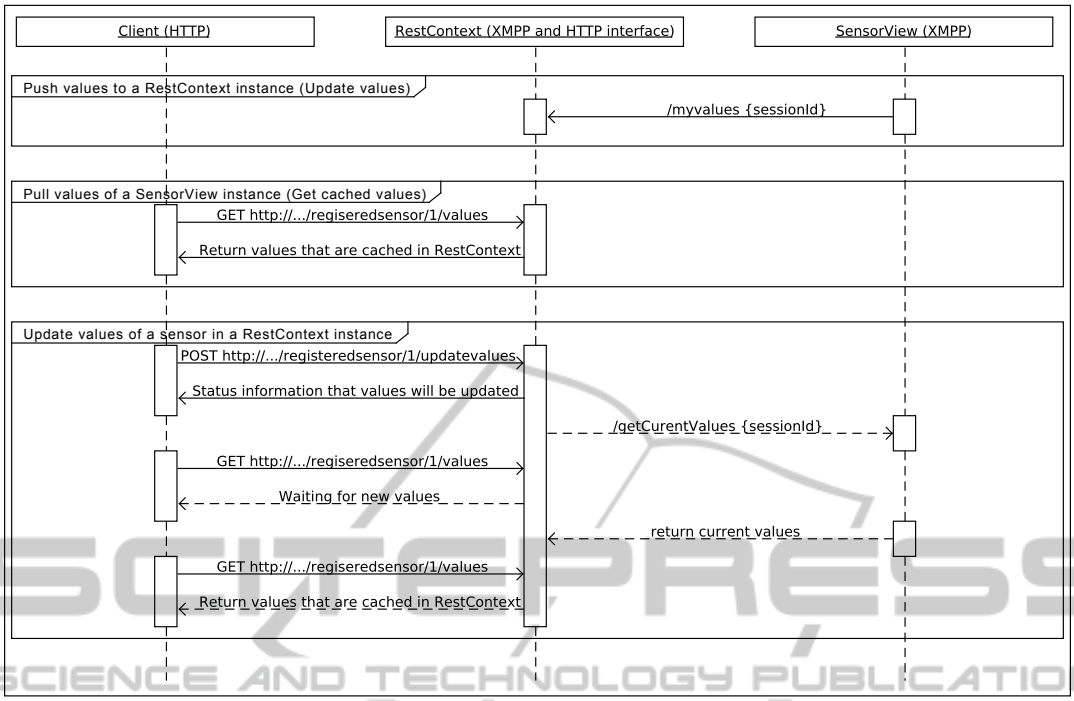

Figure 6: Sequence diagram of communication between a client and a RestContext and SensorView instance.

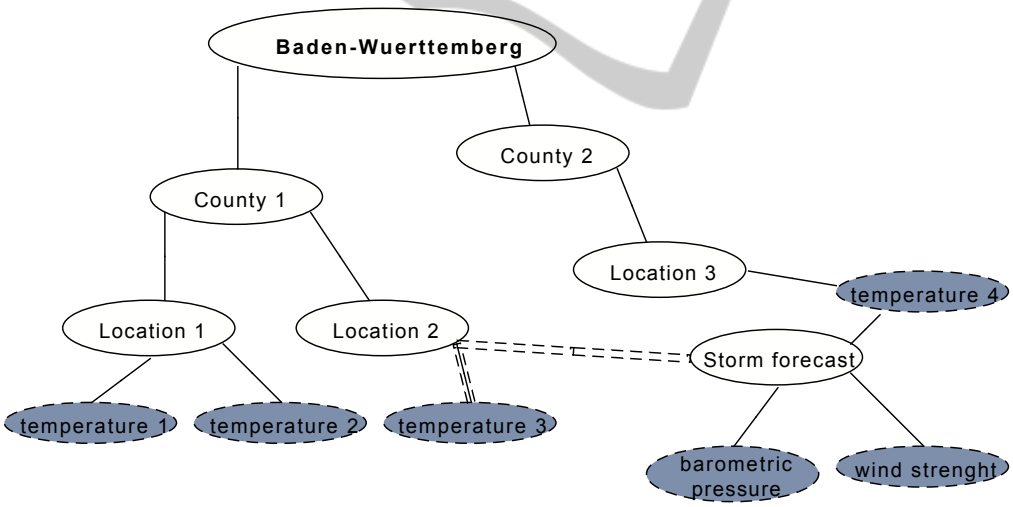

Figure 7: Use-case scenario for the RestContext service.

the url: `http://location1/registeredsensor/{temperature1-id}/updatevalues`

- To get all temperature values of the country (getting all four temperature sensors) a GET request to the following URL is sufficient: `http://country/context/{1}/values`
- A request to the storm forecast service for values does a further GET request to the *location 2* RestContext service instance. Four sensor values (*barometric pressure, wind strength, temperature 3 and temperature 4*) are returned.
- Modifications (adding, changing or removing contexts resp. sensors) are implemented by changing the list of the resource elements of context *id* (URL: `http://host/context/{id}/elements`) .

6 CONCLUSIONS AND FUTURE WORK

In section 3 we presented some middleware and context frameworks that inspired the work on the RestContext service. Especially the Context Toolkit architecture with its widget and aggregate layers proposed the separation of an instance running on or nearby the sensor hardware and a second layer with sufficient hardware resources on the internet.

Our RestContext service is an architecture for distributed contexts in order to deal with a high number and types of sensors and different settings of contexts. Based on standard communication protocols it offers interoperability between different operating systems. Due to its resource oriented design, it can be extended to further communication activities to and from the sensor devices like controlling and managing the sensor hardware by resources. However, RestContext is missing event based notifications such as the Context Toolkit. Also RestContext is static in the way of defining contexts. A definition of a context that is self-changing based on its environment needs further work e.g. , a context that contains nodes of elements that are within an area. RestContext is currently in a prototype state and missing the functionality to remove registered SensorView instances from one or many RestContext instances. Currently, we do not address advanced authentication and authorization mechanisms as presented by Sonnenbichler (Sonnenbichler and Geyer-Schulz, 2012) for distributed requests across multiple RestContext instances.

Future work includes the development of a context inference mechanism as introduced by the CASS framework (Fahy and Clarke, 2004) and a persistence mechanism to archive sensor data e.g. by extending the values resource structure. Automatic matching, conversion and merging of values e.g. automatic conversion of degree Fahrenheit to Celsius could be achieved by combining ontologies with RestContext. Aggregation techniques like average out values (e.g. air pressure) of various SensorView instances are desirable as well. Also a prioritization of nodes, e.g. to prefer nodes from local hardware rather than external information should be implemented in future versions of RestContext. Another improvement would be an automatic registration of a SensorView instance to one or many RestContext instances. This automatic registration could draw on concepts used by the Dynamic Host Configuration Protocol (DHCP) (Droms, 1997).

Further research is required to allow anonymous communication between RestContext and SensorView instances. This objective is useful for settings where the identity of the sensor should be hidden. Reasons may be privacy or security concerns e.g. in the field of traffic jam prediction where only the presence of a car on a route is important, not its identity itself. One attempt for privacy concerns might use the possibility of the XMPP protocol to allow anonymous login to XMPP servers. Asynchronous communication handling like notifications to applications working with RestContext instances can be developed by using the upcoming WebSocket protocol described in the RFC 6455 (Fette and Melnikov, 2011).

REFERENCES

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277.

Bourbaki, N. (1998). *General Topology: Chapters 1 – 4*. Springer, Berlin/Heidelberg, Germany. Originally published as Éléments de Mathématique, Topologie Générale 1–4.

Chen, H. (2004). *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD thesis, University of Maryland, Baltimore, USA.

Dey, A. and Abowd, G. (2000). Towards a better understanding of context and context-awareness. In *Proceedings of the Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems*, The Hague, Netherlands.

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

Droms, R. (1997). Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard). http://www.ietf.org/rfc/rfc2131.txt.

Fahy, P. and Clarke, S. (2004). CASS –a middleware for mobile context-aware applications. In *Workshop on Context Awareness, MobiSys*, Boston, USA.

Fette, I. and Melnikov, A. (2011). The WebSocket Protocol. RFC 6455 (Proposed Standard). http://www.ietf.org/rfc/rfc6455.txt.

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, USA.

Geyer-Schulz, A., Ovelgönne, M., and Sonnenbichler, A. C. (2011). A social location-based emergency service to eliminate the bystander effect. In *E-Business and Telecommunications*, Communications in Computer and Information Science, pages 112 – 130. Springer, Berlin/Heidelberg, Germany.

Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., and Retschitzegger, W. (2003). Context-awareness on mobile devices - the hydrogen approach. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, Hawaii, USA. IEEE.

Microsoft Corporation (2013). [ms-smb]: Server message block (smb) protocol. http://msdn.microsoft.com/en-us/library/cc246231.aspx. last accessed: 2013-03-01.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). A middleware infrastructure for active spaces. *Pervasive Computing*, 1(4):74–83.

Russell, S. J., Norvig, P., and Russell, S. J. (1999). *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, USA, 2 edition.

Saint-Andre, P. (2011). Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard). http://www.ietf.org/rfc/rfc6120.txt.

Salber, D., Dey, A. K., and Abowd, G. D. (1999). The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '99, pages 434–441, Pittsburgh, USA. ACM.

Sonnenbichler, A. and Geyer-Schulz, A. (2012). ADQL: A flexible access definition and query language to define access control models. In Samarati, P., editor, *Proceedings of the International Conference of Security and Cryptography*, Rome, Italy.

Yanes, A. (2011). Openweather: a peer-to-peer weather data transmission protocol. *CoRR*, abs/1111.0337.