Decoupling Context-aware Services

Søren Debois
IT University of Copenhagen
debois@itu.dk

Arne John Glenstrup
IT University of Copenhagen
panic@itu.dk

Francesco Zanitti
IT University of Copenhagen
frza@itu.dk

***Abstract*—We present a novel software architecture for context-aware applications based on a distributed, non-monolithic, simple and extensible relational model for representing context; a service-oriented architecture for computing these relations in a decoupled, flexible fashion; and with data driven, event based communication providing the kind of fine grained dynamic service composition required in mobile and volatile environments. A prototype implementation is running a Bluetooth–sensor-based active map of users at our home university.**

I. Introduction

In this paper, we introduce a novel software architecture for context-aware services. Within the last 18 years, the construction of software architectures for context aware *applications* has been a hot topic [1], [2], [3]; context-aware *services* joined the fray within the last five [4], [5]. The architecture we propose (cf. Figure 2) has the same aims as many of these works: it seeks to promote code reusability by encouraging compositionality of system design. It distinguishes itself from these other approaches principally by being *data-driven*: Services are composed not by specifying which services invoke which other services, but rather by specifying which services need what context information. For example, three different services might provide the same kind of location data: one based on Bluetooth (BT), one on WiFi, and one on GPS. Services consuming this location data need not concern themselves with the origin of that location data, only the data itself.

This data-driven composition mechanism is of course not in itself novel; other architectures have separated context data from its computation. The novelty here is that we make a very fine-grained division of the computation of context into such decoupled services. As a consequence we get (a) very fine-grained re-usability of code: very small computations can be meaningfully re-used; and (b) a very neat mechanism for sensor-subsystem switch-over, as alluded to above.

Our architecture is intended for the scenario where a large area, say, a shopping mall, an airport, a train station, or a supermarket is equipped with physical sensors, such as temperature or location sensor, tracking the whereabouts of customers. The tracking could be based on the clients' mobile devices, RFID equipped shopping carts, or other positioning technologies provided by the merchants.

This work funded in part by the Danish Research Agency (grant no.: 2106-080046) and the IT University of Copenhagen (the Jingling Genies projects). Authors are listed alphabetically by last name.

A key point is that our architecture is *flexible* and *extensible* in the sense that several different merchants can deploy each their system, perhaps relying on different technologies, yet still allowing the context sensitive applications to operate seamlessly.

While this scenario is not new [6], it is recently of renewed interest: it is finally happening outside the laboratory. The advent of the ubiquitous hand-held mobile phone with BT, GPS, WiFi or RFID provides a cheap, non-intrusive means of zone-based tracking of individuals in public spaces [7], [8], [9], [10], just to mention some location tracking technologies. At the same time, institutions such as airports, train stations, supermarkets etc. are beginning to deploy the necessary infrastructure for context aware systems: tracking equipment installed to find customer movement patterns [11], [12], [13] conveniently doubles as a source of location information for context aware services.

Such environments—airports, supermarkets, etc.—are associated with particular activities, and knowledge about these activities is a source of context information different from physical, sensor-obtained data. This knowledge is stored in databases, relating, e.g., passengers with flights, flights with departure times and gates, etc. Examples of activities providing context information are people going to airports to take planes and to supermarkets to buy groceries.

In most cases the expected activity of users is to a degree predictable from existing databases. In airports, the activities are almost fully formalised: a database somewhere knows passengers' names, passports and flight numbers, as well as plane departure times, gate numbers, etc. In supermarkets, a database somewhere knows exactly what goods are available in the supermarket, where they are, and how fast those goods usually disappear in the course of a normal business day. If a supermarket in question has a frequent-shopper program, some database somewhere might also have a very precise idea about what particular individual shoppers tend to buy.

Thus, the present architecture targets a scenario where context awareness is based both on physical sensors and some well-defined activities we may get from existing databases—a scenario in fact being realized commercially right now.

The architecture proposed in the present paper is not only proposed, it is in fact implemented and available for download [14]. A prototype active map tracking people at our university is also available online [15].

978-1-61284-936-2/11/$26.00 ©2011 IEEE

II. Context-aware Services

We contend uncontroversially that *the* central challenge of context-aware computation consists of computing the actual context from whatever data is available. Software architectures for context-awareness distinguish themselves primarily by what way they assist that computation.

It's important to note that for even the simplest of examples, such computation is non-trivial. To be concrete, suppose a context-aware application needs to know whether some passenger in an airport has passed through security. This is a simple boolean piece of information, obtainable by simply inspecting the current location of the passenger in question. Even so, in even the most naive of implementations, this computation will typically require (at least) two steps. First, the information that the device with MAC-id 0x2198f172 is currently in range of sensor 0x31 must be converted to the information that passenger Jones is in the Lufthansa First Class Lounge. Second, we must check whether this Lounge is in fact situated before or after security.

The opportunity for code re-use in the development of context-aware applications stems from the observation that within a given application domain—say, a specific airport—some of these computations will be almost universally useful. In the above example, the ability to map particular sensor inputs to particular passenger's high-level location would likely have such usefulness. In fact, also the information whether or not a passenger has passed security is conceivably interesting to variety of applications, even if it is not universally useful.

As usual for Software Oriented Architecture (SOA), we consider a context-aware service to be something which implements a reusable, distinct computation of context. The application in the above example then uses two services: one for the low-level sensor-to-passenger mapping, and one for the high-level location-to-security-state mapping.

However, put this way, there is no discernible difference between a *context aware service* and a *context aware procedure*. To maximise the potential reuse of context-aware services, we shall further decouple them in the following section.

III. Context-aware Service Interfaces

Let us call the mapping of low-level sensor-data to high-level location L, and the mapping of high-level location to security state H. Suppose that the low-level sensors it interfaces with is the Ekahau WiFi-positioning system [7]. In places where WiFi is not available, the airport can augment the location tracking with a BT-based positioning system. How do we modify the existing context-aware application to use also the new BT-based tracking system?

By use of the service H, the remainder of the application is already shielded from this change. But what should happen to H and L themselves? In a procedural world, H and L would be tightly coupled. For H to also accommodate the new location system we must implement a service L' taking BT-based positions to high-level locations, and we would have to modify H so that it always queried both L and L'.

But there is a better way! We know that the service H is only interested in high-level locations. It is not interested in other services. There is no need for H to know where its high-level location information comes from; whether it be L, L' or some third entity is absolutely irrelevant to H.

Thus we propose that services be further decoupled so that the composition is not formed by one service directly accessing another service, but rather by one service requiring particular data, no matter who provides it. In composition, the relevant identifier for a data-stream from a service is no longer its name, but rather the particular kind of data it produces and consumes.

However, this definition leaves us with a practical problem: how does a service specify the particular kind of data it produces or consumes?

IV. Relational Context Models & Relation Transformers

Describing what context-data is produced is of course a matter of selecting a context meta-model. Fortunately, the problem of describing classes of context models, of constructing a context meta-model, is by now fairly well studied, and has produced suggestions using various techniques, ranging from tuple-space [2] and object-orientation [4] to ontology-based [16], and even combinations (e.g. OO with ontology [17]).

The great strength of ontologies is that they come with description logics as declarative languages for specifying how context computation is done. However, the ontological meta-model invites a centralized approach, with one centralized database and reasoning engine. This centralization is inherently at odds with the breaking down of the computation into small, re-usable pieces that we are looking for.

However, we salvage the key abstraction from ontology-based context-models: at its core, an ontology defines how objects are related. So perhaps to describe context, it is sufficient to agree on the meaning of some set of relations? It seems so; any data relevant to context-aware applications would seem to immediately fit the relational model. For the above example, an ekahau relation relates WiFi MACs and sensor ID's; in particular, one data point could be

ekahau [wifimac=0x2198f172, wifisensor=0x31].

A different relation would indicate the high-level locations:

loc [passenger=Jones, area="Lufthansa First Class Lounge"]

Similarly, a relation relating BT MAC-addresses to other BT MAC-addresses might represent the information about which mobile phones are currently being detected by which BT sensor. A relation relating passenger names to flight numbers might represent the information about which passengers are booked on which flights. A relation relating flights, gate numbers and departure times might represent information about which flights are scheduled to depart from which gates and at which times, etc.

Incidentally, notice that the meta-model of relation is actually that of relational databases. This observation should, at

least on an intuitive level, justify our expectation that whatever context we wish to describe, it should be expressible in this relational model.

So, the interface of a service is the type of relations it consumes and produces. Both of the services L, L' produce the loc relation; the service H consumes that relation.

Do notice the pleasing abstractions here: the context is represented as some set of relations; computation on context is then essentially a function on relations, what we might think of as a relation transformer. The entire computation of high-level context from low-level context is performed by letting some set of such relation transformers interact.

V. Data Driven Event Based Communication

At this point, the only remaining key idea of our architecture is the mechanism needed to connect services wishing to communicate on some relation, e.g., to connect the loc relation output of L, L' to the loc relation input of H. The architecture supports a basic push-based event publish-subscribe mechanism, on top of which other interaction patterns can be constructed if needed.

The routing of events is completely data-driven; a service subscribes to observe changes in a relation, and any events emitted changing the state of that relation are routed to the service. For instance, to subscribe to all changes in the ekahau relation, a service issues a subscription (pseudocode):

```
subscribe ekahau [wifimac=?, wifisensor=?]
```

To monitor only a specific area, the service fixes a value in the subscription:

```
subscribe ekahau [wifimac=?, wifisensor=0x31]
```

A subscription will not by itself make events appear; that requires services publishing that event. It is, however, acceptable for a service to subscribe to events not currently being published by anyone—they might appear later.

VI. Architecture Summary

To summarize, in our architecture, the computation of *what is* the current context is broken down into units of computation called services. Services consume small parts of the context-information and produce in turn new small parts. The parts themselves are represented as relations, and services never communicate directly: rather, they indicate their interest in particular relations. Notice that a service needs to know only the relations it is interested in; this makes fairly easy to add, remove and change services.There is no central place hosting the definition of context. This definition is itself dynamic, and changes with the available services. Actual communication between services rely on an event-distribution mechanism, on top of which one can build query-mechanisms. The interface between the services and the event-distribution sub-system is listed in the example listed in Table I (pseudocode, refer to [14] for actual interface).

We emphasize that in this architecture there is no centralized representation of the current context. The architecture only forces representation of a part of the context the instant that part must be transmitted as an update to a relation from one service to another. This is the *only* situation in which the architecture requires explicit representation of the context being manipulated. Individual services are of course free to preserve as much or as little of this information as they might like; one might for instance implement a location history-service by listening for and locally storing location-events, answering queries based on these stored events.

The architecture helps disseminating changes in relations among the different services, as long as they agree on the interface — i.e. the definition of the relation they are interested in or that they provide. This dissemination process happen at run-time, so it is possible to change it dynamically, e.g., to change a service implementation or to add relation providers.

Altogether, services are fully decoupled. It is not necessary to know in advance which service will provide required context information, nor to know which service will use provided context information. These dependencies will be resolved (if possible) in the moment a service is communicating a change in a relation.

VII. Prototype Implementation

A prototype implementation of the architecture has been implemented. In this section the key properties of the prototype are presented.

To avoid dependence on a specific programming language, we have employed language-agnostic technologies: the HTTP protocol, and the JSON [18] data format. Libraries for accessing these protocols exist for most programming languages, making our platform easily portable—currently, we have prototype support for Java, C#, Javascript and Python.

Dispatching events in an asynchronous fashion using the HTTP protocol is straightforward if the subscribing service is itself exposed as a web server, and this kind of communication is supported in our prototype. However, to overcome firewalls and NAT routers breaking the architecture, we leverage the Bayeux protocol [19], which offers a publish/subscribe mechanism over HTTP. As the Bayeux subscription mechanism is topic-based, we built a thin protocol on top of it that simulates content-based subscription [20]: first the service subscribes to a system channel, then it tells the server which event pattern it is interested in, and finally the server responds with an event channel. The service proceeds to subscribe to this latter channel, whence it will receive the contextual events.

We have observed two disadvantages of Bayeux. First, the protocol itself imposes noticeable network overhead: every time events are received, a new connection is made to the server. Second, different server implementations are not yet fully interoperable, requiring slight adaptations to work smoothly. Thus, in the future we will experiment with alternative interfaces, e.g., Web Sockets [21].

Regardless of transport technology, services specify event streams by providing to the server a *pattern*, a list of triples in the form of {`FieldName, Operator, Value`}. To check if a pattern matches a particular event, event and pattern values are compared using the operator; if all are satisfied,

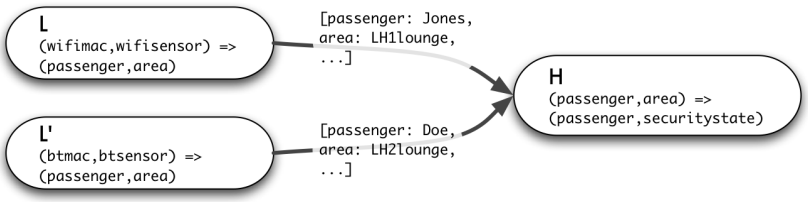

Fig. 1. Context model; services L and L' consume $wifimac, wifisensor$ and $btmac, btsensor$ relations, respectively, and both export $passenger, area$; this relation is consumed by H, which emits $passenger, securitystate$ relations

Example	Description
publish ekahau[wifimac, zone]	Declare intent to publish ekahau events, carrying data-fields wifimac and zone.
notify ekahau[wifimac=0x2f, wifisensor=0x31]	Publish ekahau event carrying the indicated data.
subscribe ekahau[wifimac, wifisensor=0x31]	Request notification of future ekahau events for which wifisensor is 0x31, wifimac arbitrary. Such filtering is optional. Used with callback below.
receive ekahau [wifimac=x, zone=y]	Callback. Receive notification of ekahau event; x and y will be filled in.

TABLE I
SERVICE/DISTRIBUTION INTERFACE

the event is delivered. An event is a key/value tuple, without special attributes; to simulate type of events, each event is tagged with the field `type`. Supported operators are equality, containment, size comparison and an undefined operator, requiring the field not to be present in the event. This very basic language, with its obviously limited expressiveness, turns out to be sufficient for our sample applications. In the future, we expect to expand the available operators, and to support disjoint patterns.

For the initial prototype implementation, subscription and dispatch are handled by a centralized server, making deployment simple. Note that we centralized the dissemination of the events, not the representation of context — that resides with individual services, if at all, so it is already distributed. The main disadvantage is that all communication must be handled by a single machine, limiting system scalability. For supporting larger wide-area installations, we envision moving to a fully distributed event-distribution mechanism as future work. More precisely, we will leave the server/services communication as-is, implementing the distribution only at the server level. A simple possible solution is to enrich the runtime with some system messages that makes a server aware of the presence of other servers, and using the very same event distribution system we already have, maintain their state. This way, when a service subscribes to an event stream, the same subscription is replicated on all the servers that provides a compatible stream.

VIII. EXTENDED EXAMPLE

Figure 2 shows an outline of a web application displaying simply users' locations running at http://tiger.itu.dk:8000/ITUitter/.

If the user has, say, a BT capable phone, its location can be detected by BT sensors, providing crude zone based tracking. When the phone moves between zones, an event updating the BT-interface/sensor relation is emitted on the hub. Any services subscribing to such events will receive it; in particular, a conversion service maintaining a user/area relation will receive it and emit a user/area update event. It bases its conversion from BT interface to user on information received

Fig. 2. Outline of a web application for displaying an active map of users' locations.

via events from services that maintain interface/device and device/user relations. The web application subscribes to these events, and updates browser displays accordingly.

The flexibility and extensibility of this architecture is demonstrated by integrating WiFi-based trilateration simply by connecting a WiFi event adapter to the genie hub, and adding a WiFi location converter.

Also note that service discovery and service composition has been relegated to transformer services and replaced with the problem of simply listening for the right events, viz. the BT/WiFi location converter services above. The Web App application itself need not concern itself with exactly how the events it is listening for are synthesized, not which services are involved in that synthesization. Moreover, if only BT-tracking were available in some locations and only WiFi in other, the Web App would receive events arising from either subsystem as appropriate, oblivious to the difference. Thus, service discovery and composition happen completely

dynamically, and *transparently* for the application.

It is important to note that this architecture is not constrained to mere location-based applications. For instance, our university has extensive databases on course membership, course room allocations etc. As we hinted above, activity of individuals at the university can to a large degree be approximated from data in these databases; moreover, the database representation of such information fits snugly into the relational context meta-model of our framework.

IX. Related Work

Research into software architectures for context-aware applications has been a hot research topic for the last decade. Presently, we contrast our proposed architecture to four existing ones: The seminal Context Toolkit [2], Solar [22], [23], SOCAM [24], and SLCA [25]. There are of course many more such architectures; these four were chosen for their influence and familiarity, and because it is instructive to observe the differences to our architecture.

The **Context Toolkit** resembles our architecture in that it affords code reuse through compositionality, and in that this is achieved by breaking the computation of *what is* context into distinct pieces, called *widgets*. The key difference with our work is that composition in the former is object oriented, whereas for us it is data driven. A context widget explicitly decides *which other widgets* to use, whereas our services only decide *what information* they need. In a sense, the present work refines the compositionality of the Context Toolkit by separating the notions of *context information* and *code computing context information*, notions which coincide in the widget-abstraction. This decoupling refinement offers the key advantages of increased code re-use through increased modularity (cf. Sections III and VIII); and a dynamic and transparent context-source switch-over technique (cf. Section IV).

Solar is a distributed and event-based architecture that is data-driven, but in a different sense than the present work. It is superficially very similar to the present effort. The key difference—Solar's impressive array of applications and much greater maturity aside—is the notion of "data-driven". The present notion of *service* is almost the same as a Solar *operator*; both consume a number of input event streams and produce a number of output event streams. However Solar conflates the notions of *event stream* and *event stream provider*, conflating context information with its producer. Thus the examples of code reuse and dynamic composition from Section III are not immediately realisable in Solar.

The notion of *data-driven* employed by Solar is instead one in which *subscriptions* can be data dependent. That is, an operator may specify that it is dependent on some event stream identified by the data currently carried by some other operator. For instance, an operator might specify that it wants video data from a camera selected by the computation of some operator.

This data-driven event stream selection is the primary dynamic composition mechanism of Solar. This mechanism is almost but not quite strong enough to realise the examples of dynamic and transparent context-source switchover of Section IV: while it can certainly do the switchover, it will be necessary to construct an operator emitting events whenever the switchover is necessary, partly defeating transparency. Moreover, it is easy to find examples where two operators could meaningfully simultaneously inject events into the same event stream, e.g., partially overlapping sensors sub-systems.

Next, **SOCAM**, the Service-Oriented Context-Aware Middleware, resembles the present work. One, it proposes a bridge between the notions of *service* and the notion of *context-aware*. Two, it proposes that the services comprising a system communicate using a shared context meta-model. Three, it proposes that this meta-model be used for indirection; that instead of services contacting context information producers directly, they request specific information, whence the requested information is automatically routed their way.

However, the notion of *context-aware service* proposed by SOCAM encourages larger pieces of functionality than in the present framework; in particular, SOCAM services are, with two exceptions, expected only to *use* context information, not take part in the *production* of that information. Communication using a shared meta-model and the indirection that services request information, not other services, is of course the same mechanism as in the present work. However, SOCAM's meta-model is a highly structured ontology model, where as ours are but flat relations. Moreover, whereas matching a service' request to the available information streams is in our case the simple matter of comparing stream types, SOCAM relies on a specialised Service Locating Service interacting with a centralised knowledge-base, rule-base and reasoning engine.

In short, whereas we envision context-aware services as small, lightweight processes participating in computing the context represented relationally, SOCAM envisions context-aware services as business-service-sized pieces of functionality using highly structured context information to locate each other.

Finally, **SLCA**, the Service Lightweight Component Architecture. The present work resembles SLCA in that both emphasises event-based decoupling and light-weight composition of services. However, the approaches are not really comparable: Whereas the present architecture uses events to decouple services via a context meta-model, SLCA uses a high-performance service orchestration mecahnism. So composition in SLCA is service-oriented: it is one of services; whereas in the present work composition is data-driven: it is one of event streams.

X. Conclusion and Future Work

In this paper, we have proposed an event based, data driven and SOA for context aware applications. The architecture emphasizes code re-usability and fine grained dynamic context adaptation.

Starting from the observation that context-aware applications must compute *what is* the context, the framework encourages the programmer to split that computation into small

parts, each part performed by a service. Services are strongly decoupled, communicating only by requesting or producing specific context information. A producing service does not know exactly who are its consumers; conversely, a consuming service does not know exactly who are its producers. This decoupling yields strong code re-usability and the opportunity for highly dynamic run-time adaptation. The prime example of the latter is the transparent switch-over between location systems, afforded simply by the fact that the high-level service does not need to care whether location events originate with, say, a WiFi location subsystem or a BT location subsystem.

a) Future Work.: A key assumption for the present architecture is that of end users' possession of mobile devices. With that assumption, we have the opportunity to consider not only how to collect and aggregate context-information, but also how to interact with the user utilising that mobile device. A common solution exploits that the user's device tends to be internet enabled, whence such interaction can be simply web based. However, that would miss out on an opportunity for the architecture to provide (semi-) transparent switch-over between interaction devices—say, the user's mobile devices when there are no other options, public displays when they are available, etc. Instead of handling the mere passive data acquisition, we are pursuing a way to permits the services to have a *conversation*, in a broad sense, with the user, by using some physical actuator near the user (it might be her mobile device, a loudspeaker, or some other interaction device).

The present architecture poses some hard questions about privacy. In the current prototype, users have none: they are tracked at the discretion of service providers. Envisioning that third-party or otherwise not completely trusted applications are allowed on the architecture, it becomes vital that users or other parties can control which services can access what location data. Such control is complicated by the decoupling inherent in our architecture, because data-flow from sensors to application can change unbeknownst to that application; in particular, the application has no knowledge of which intermediate services are used. A dual security issue is authentication, knowing that context-information produced is in fact genuine.

The examples in the present paper, and the examples we have otherwise studied in the implementation, there have been no impedance mismatch between low-level sensor-data and high-level context information. For instance, in this paper, bluetooth mac-addresses and sensor-terminals map straightforwardly to users and zone-based location. Conceivably, in more advanced settings, and for more advanced applications, this correspondence between low-level sensor-data and high-level context is less clear. Low-level decisions about sampling, framing, etc., must intuitively follow from requirements of the top-level application. We are confident that such decisions are naturally representable by introducing suitable event-transformers, but have yet to study a non-trivial case in practice.

Finally, we would like to support distributed event processing, and enrich the subscription pattern language.

REFERENCES

[1] Want, R., Schilit, B., Adams, N., Gold, R., Petersen, K., Goldberg, D., Ellis, J., Weiser, M.: An overview of the PARCTAB ubiquitous computing experiment. IEEE Pers. Comm. **2** (1995) 28–43

[2] Salber, D., Dey, A.K., Abowd, G.D.: The Context Toolkit: Aiding the development of context-enabled applications. In: CHI 1999. (1999)

[3] Ranganathan, A., Campbell, R.H.: An infrastructure for context-awareness based on first order logic. Pers. UbiComp **7** (2003) 353–364

[4] Bardram, J.E.: The java context awareness framework (JCAF) - a service infrastructure and programming framework for context-aware applications. In: Pervasive'05. Number 3468 in LNCS, Springer (2005) 98–115

[5] Tigli, J.Y., Lavirotte, S., Rey, G., Hourdin, V., Riveill, M.: Lightweight Service Oriented Architecture for Pervasive Computing. IJCSI **4** (2009) 1–9

[6] Asthana, A., Crauatts, M., Krzyzanowski, P.: An indoor wireless system for personalized shopping assistance. In: WMCSA '94. (1994) 69–74

[7] Ekahau Real-time Location System: http://www.ekahau.com (2010)

[8] ZONITH Bluetooth based indoor positioning: http://www.zonith.com (2010)

[9] Bruno, R., Delmastro, F.: Design and Analysis of a Bluetooth-Based Indoor Localization System. In: Pers. Wireless Comm. (2003) 711–725

[10] BlipZones: http://www.blipsystems.com (2010)

[11] SPOPOS Project: http://www.spopos.dk (2010)

[12] Hansen, J.P., Alapetite, A., Andersen, H.B., Malmborg, L., Thommesen, J.: Location-Based Services and Privacy in Airports. In: INTERACT. (2009) 168–181

[13] Frank, T.: Airport device follows fliers' phones. USA TODAY, March 23 (2010)

[14] Debois, S., Glenstrup, A.J., Zanitti, F.: SIC Framework Download (2010) http://www.itu.dk/people/frza/infobus/.

[15] Glenstrup, A.J.: ITUitter (2010) http://tiger.itu.dk:8000/ITUitter/.

[16] Baader, F., Calvanese, D., McGuiness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook. 2nd edn. CUP (2007)

[17] Lee, D., Meier, R.: A hybrid approach to context modelling in large-scale pervasive computing environments. In: COMSWARE'09, ACM (2009) 1–12

[18] Introducing JSON: http://www.json.org (2010)

[19] The CometD Project: The Bayeux protocol (2010) http://cometd.org.

[20] Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM CSUR **35** (2003) 131

[21] The Web Apps Working Group: The websocket API (2010)

[22] Chen, G., Li, M., Kotz, D.: Data-centric middleware for context-aware pervasive computing. Perv. & Mob. Comp. **4** (2008) 216–253

[23] Kotz, D., Chen, G.: Context aggregation and dissemination in ubiquitous computing systems. In: WMCSA'02. (2002) 105–114

[24] Gu, T., Pung, H.K., Zhang, D.Q.: A service-oriented middleware for building context-aware services. J. Net. & Comp. Apps. **28** (2005) 1–18

[25] Tigli, J.Y., Lavirotte, S., Rey, G., Hourdin, V., Riveill, M.: Lightweight Service Oriented Architecture for Pervasive Computing. I. J. of C. S. Issues **4** (2009) 1–9