SIP-based Context Distribution: Does Aggregation Pay Off?

Alisa Devlic
Appear Networks &
Royal Institute of Technology (KTH)
Kista, Sweden
devlic@kth.se

ABSTRACT

Context-aware applications need quickly access to current context information, in order to adapt their behavior *before* this context changes. To achieve this, the context distribution mechanism has to timely discover context sources that can provide a particular context type, then acquire and distribute context information from these sources to the applications that requested this type of information. This paper reviews the state-of-the-art context distribution mechanisms according to identified requirements, then introduces a resource list-based subscription/notification mechanism for context sharing. This SIP-based mechanism enables subscriptions to a resource list containing URIs of multiple context sources that can provide the same context type and delivery of *aggregated* notifications containing context updates from each of these sources. *Aggregation of context* is thought to be important as it reduces the network traffic between entities involved in context distribution. However, it introduces an additional delay due to waiting for context updates and their aggregation. To investigate if this aggregation actually pays off, we measured and compared the time needed by an application to receive context updates after subscribing to a particular *resource list* (using RLS) versus after subscribing to each of the *individual context sources* (using SIMPLE) for different numbers of context sources. Our results show that RLS aggregation outperforms the SIMPLE presence mechanism with **3** or more context sources, *regardless* of their context updates size. Database performance was identified as a major bottleneck during aggregation, hence we used in-memory tables & prepared statements, leading to up to 57% database time improvement, resulting in a reduction of the aggregation time by up to 34%. With this reduction and an increase in context size, we pushed the aggregation payoff threshold closer to 2 context sources.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network protocols—*Applications (SMTP, FTP, etc.)*; C.4 [**Performance of systems**]: design studies, measurement techniques, performance attributes

General Terms

Design, Performance, Measurement, Experimentation

Keywords

Context distribution, aggregation, RLS, SIMPLE, XCAP

1. MOTIVATION

In a context-aware system, mobile devices need to discover, collect, and provide available context information from their surroundings to the context-aware applications to enable them to adapt their behavior, *before* this context changes. This can be challenging since the *greater* the speed of a user's movement the *more frequently* a user's context changes, thus potentially leaving no time for the applications to adapt.

Due to the variety of available access networks, these devices can change their connectivity when they move to another location. Additionally, due to the device's specific processing and communication capabilities, users may want to employ various devices in different situations and can potentially switch to another device during a communication session. At the same time, users may want to be reachable and have transparent access to their desired services regardless of their current location, the device they are currently using, or the access network this device is currently connected to (i.e., their *context information*).

Context information is sensed from the device or the environment using some automated means (i.e., via sensors). Context sources retrieve raw context data from sensors and provide semantic markup to this data. By executing in the *same* device as the application that requests the desired context, context sources provide a **local** device's context. This context collection mechanism is called **context sensing**.

As context is often produced in *different* devices than where it is consumed, it needs to be distributed to the applications that have expressed interest in receiving this information. Such a mechanism is called **context distribution**, providing a **remote** device's context to these applications.

This discussion raises following questions:

- how to discover to which device(s) should the context information be delivered;
- how should the relevant context sources be discovered and activated, upon a user's arrival at a new location;
- how and when should the context sources that are no longer relevant be deactivated;
- how can users and their devices be uniquely identified in different networks;
- how to distribute context with different dynamics to applications that have specific requirements related to the latency, frequency of context updates, and the network traffic (that these updates can generate);
- how to support user, terminal, and session mobility.

These questions represent challenges that a context distribution mechanism needs to address. Some of these challenges are already known and widely adopted as criteria for context provisioning solutions [19, 4, 14]. A few others have been adopted from [2] and the rest of the challenges have been identified during design, implementation, and integration of our proposed distribution mechanism with the context sensing mechanism adopted from [13].

1.1 Requirements

Based upon the identified challenges we created a list of requirements that should be considered when selecting an existing or designing a new context distribution mechanism.

- **Support of the context model**. A context model represents a vocabulary that allows context-aware applications to deal with abstract terms rather than with raw, technical data. Therefore, the context distribution mechanism has to support the model with potentially rich semantics to enable knowledge sharing among network nodes running these applications.
- **Short response time**. We set the minimum time between arrival of two context updates at the application to be 1 second, since a user cannot be expected to react to updates arriving more frequently than once in a second. Given the one second of time between context updates reaching the application, the total waiting and the adaptation time have to be less than one second - or the adaptation will not be able to keep up with the context changes, without skipping some context. We set the context distribution time limit to 500 ms, leaving the remaining time to the application to adapt.
- **Timely discovery of context sources**. As users need to access context information from their surroundings while on the move, the appropriate context sources have to be discovered each time a user changes location. The context distribution mechanism needs to notify interested devices about discovered context sources quickly enough to allow them to retrieve and adapt to the current context, before they move to another location. The discovery time limit has been set to 15 ms in order to minimize the context distribution delay.
- **Transfer of context data over heterogeneous access networks**. Context information often needs to be transfered to devices connected to different access networks (such as WLAN, 3G, Bluetooth, etc). Therefore, context distribution should transfer context data from the device(s) in which context sources execute to the application device, independently of the access network(s) to which these devices are connected.
- **Unique naming and addressing**. Users and context sources have to be able to identify and locate each other in order to conduct context transfer, even if they reside in different networks. Therefore, context distribution has to support an abstract scheme of addressing users and context sources, which has to be independent of the underlying network and types of devices.
- **Support for user, terminal, and session mobility**. Since mobile users can employ a wide variety of devices in different situations, they can potentially switch between devices during a communication session. To support this, the context distribution needs to identify a device the user is currently using and enable the seamless delivery of context data potentially starting on one device and move to another device - without having to initiate a new session.
- **Activation and deactivation of remote context sensing**. As contexts in mobile environments may change frequently a problem arises if contexts are rapidly transferred to the users' devices, but this information is subsequently **not** used by applications. This can potentially lead to high resource consumption on the devices which host the context sources (in terms of power consumption, bandwith utilized, etc). In order to avoid the need for context sources to continuously provide their data to the system, the context distribution mechanism should activate context sensing on the remote device(s) when needed and deactivate it when this context is no longer required.
- **Distribution mode**. Some context changes frequently (such as location), while other context can change very slowly or not at all (such as user's profile). Moreover, various context-aware applications have different requirements for receiving context information (i.e., periodic pooling, on request, or receiving an update each time the context changes). In order to support frequent and infrequent context updates, **both** synchronous (request/response) and asynchronous (subscribe/notify) access to context should be provided.
- **Support for aggregation**. Context information (potentially of the same type) can be produced in multiple devices generating the need to support aggregation of context produced by these different context sources. Context aggregation is desirable as it reduces the amount of network traffic compared to that which would be caused by individual context updates [9].
- **Scalability**. The context distribution mechanism must scale well with the number of context sources, their rate of context updates, the number of context requestors, and their context requests/queries.
- **Robustness**. The context distribution mechanism should tolerate temporary disconnections of devices when devices move out of the network coverage or due to the network disruptions (such as handovers).
- **Security/privacy**. There is a need for controlled access to a user's context, when this user wants to share his/her sensitive data with different parties. Additionally, this data should be securely transferred such that the traffic contents cannot be inspected or modified by entities that are not involved in context distribution.
- **Deployment**. Existance of the underlying communication protocol and infrastructure in desktop, mobile, and server platforms is essential for context distribution mechanism to be widely supported.

In this paper, special attention will be put to support for **aggregation of context**. Context distribution delay will be measured with aggregation and compared with the time that would be needed by the context distribution to

deliver individual context updates from each of the context sources. This comparison will enable us to answer the question of whether aggregation of context from multiple context sources pays off and if so, is there a treshold in the number of context sources in order for aggregation to pay off.

1.2 State-of-the-art distribution mechanisms

This section briefly describes state-of-the-art frameworks that implement context distribution mechanisms and discusses to what degree these mechanisms meet the identified requirements. The described frameworks have been selected for review because of different methods and protocols that they use for context distribution.

1.2.1 Context toolkit

Context toolkit [5] provides programming abstractions to application developers that perform context acquisition, storage, and distribution functions, thus simplifying the development of context-aware applications. However, it lacks the *unique naming and addressing* support. In order to share context, communication entities have to provide the hostname or the IP address of the device they run on and the port number on which the device listens for communications.

Context toolkit *does not scale* well with the number or frequency of context updates [5], because all components need to constantly listen for context messages and a new thread has to be created to process each new message.

Context toolkit does not provide *user, terminal, or session mobility*. There is a limited support for *robustness* since temporary disconnections or network failures can disable the distribution until the device reconnects. When this happens, the widget can reconnect to its subscribers as it maintains a subscription log to keep track of all the subscribers, but it cannot recover the subscription state as it was before the connection went down nor can it retrieve lost notifications.

The transfer of context data over *heterogeneous networks* is not supported, unless a naming system similar to Domain Name System (DNS) or a discovery protocol such as Service Location Protocol (SLP) was implemented on top of the hierarchy of Discoverers to enable discovery of context sources accross different administrative domains.

1.2.2 JCAF

JCAF [3] is a context-aware framework in Java that handles management and distribution of context, leaving the application developers more time to focus on context modeling and implementation of application logic.

JCAF uses Java RMI to distribute context information and due to its dependence on a specific programming language our requirement for *deployment* is only partially fulfilled. JCAF supports both *distribution modes*, but in the synchronous mode there is a potential deadlock that can occur due to the blocking of the monitor execution until it reads the context from the sensor – if this sensor never answers. Therefore, this mode is not desirable, only partially fulfilling our requirement for *activation and deactivation of remote context sensing*. Additionally, JCAF lacks support for automatic *discovery of context sources*.

JCAF partially satisfies *short response time* and *scalability* requirements, because the performance of Java RMI is acceptable when transfering small or medium sized chunks of context data over high-speed data links, outperforming web services but being slower than TCP. However, when transfering larger chunks of data the RMI performance degrades rapidly, decreasing the difference in performance between RMI and web services. For comparison, RMI needs 5.1ms to transfer 10B of data versus 164.7ms to transfer 50KB [7].

JCAF does not support *user, terminal, and session mobility* nor does it provide *unique naming and addressing*. Thus, it can not provide seamless transfer of context data over *heterogeneous networks*. Additionally, JCAF can not cope with temporary disconnections of devices from the network. Therefore, the *robustness* requirement is not fulfilled.

1.2.3 Contory

Contory [16] is middleware specifically designed for context sensing, context processing, and context distribution on smart phones. It uses the following three strategies to execute these functions: (1) on the same mobile device where the applications run, (2) on remote devices, or (3) at nodes of the mobile ad-hoc network. These strategies are adaptive, which one is applied depends on dynamic operating conditions, such as sensor availability and resource consumption.

The requirement for *short response time* in Contory is not met. The time to receive a context item via Bluetooth is 31.83 ms; via Wi-Fi is 761.28 ms (for one hop) and 1442.5 ms (for two hops); and using UMTS this time is 1473 ms. All these times (except using Bluetooth) exceed our distribution delay limit (i.e., 500 ms). Moreover, the Bluetooth time noted above excluded the device and service discovery times, which are on average 13 seconds and 1.12 sec, respectively [16]. As a result no distribution of context data that requires Bluetooth device and service discovery can meet the timely discovery requirement – independent of the framework that it is used (including our own!).

Contory provides several ways of addressing different types of context source nodes and entities, but only in the ad hoc strategy, thus the naming and addressing of a node are coupled to the underlying network, therefore Contory does not meet the *unique naming and addressing* requirement. Furthermore, there is no *mobility* support in Contory.

Contory allows an application to assign a level of trust to the context source and to lock access to a particular context with a key that must be known by a requester. However, some security threats such as eavesdropping on intermediary links between the repositories and context sources, traffic analysis of queries, and the risk that a malicious user learns the key are not tackled, therefore the *security and privacy* requirement is only partially met.

By providing distributed context provisioning, Contory avoids a single point of failure. However, as we did not find any scalability results for this middleware, hence we concluded that it partially satisfies our *scalability* requirement.

1.2.4 JHPeer

JHPeer [21] is a hybrid peer-to-peer framework for context-based retrieval and distribution. It builds upon the JXTA platform by adding multiple super peers to address the problem of a single point of failure and to increase the distributed system's scalability. To the best of our knowledge there are no results regarding scalability of JHPeer, however some experiments have been performed on JXTA 2.0, showing that the response times to context queries do not depend so much on the peer group size, as on the query rate [8]. In particular, an increase in the average response time between groups of 8 and 32 peers is less than twofold for the same query rate.

	Context toolkit	JCAF	Contory	JHPeer	SIP-based
Support for context model	+	+	+	+	+
Timely discovery of context sources	+	-	+	-	-
Short response time	+/-	+/-	-	-	+
Transfer over heterogeneous networks	-	-	+	+	+
Unique naming and addressing	-	-	-	+	+
Support for user, terminal, and session mobility	-	-	-	-	+
Activation and deactivation of remote context sensing	+	+/-	+	+	-
Distribution mode	+	+	+	+	+
Support for aggregation	+	+	+	-	-
Scalability	-	+/-	+/-	-	+
Robustness	+/-	-	+	+/-	+
Security/privacy	+	+	+/-	+	+
Deployment	+	+/-	+	+	+

Table 1: Summary and comparison of context distribution mechanisms

At a rate of 8 queries per second, JXTA rendezvous peers could not respond in less than 15 ms, but instead in tens and hundreds of seconds, thus not satisfying the requirement for *timely discovery of context sources*. Despite its good general scalability properties, the performance seems to be good only at lower query rates of 2 and 4 queries per second, thus failing to meet our *scalability* requirement. Additionally, *aggregation of context* is not supported by this framework.

The evaluation results of JHPeer in [21] showed that it takes on average 800 ms to get a response to a context query, after publishing 15000 advertisements to a peer group. This does not satisfy our *short response time* requirement.

JXTA supports both distribution modes, however only the unidirectional asynchronous mode (which is enabled by default in JXTA) provides an easy fault recovery in case of peers disconnections from the network or network failure, thus only partially fulfiling our *robustness* requirement.

As JXTA does not support the use of multiple network interfaces by each peer or the assignment of more than one address to the same interface, the *user, terminal, and session mobility* are not supported.

1.2.5 SIP-based distribution

Goertz, Ackermann, and Steinmetz proposed to implement context sharing during call setup based on SIP [6]. Their idea was to enable the calling party to acquire the callee's current context in order to decide whether or not to initiate a call. They proposed two methods for context sharing: (1) a direct query/response mechanism between two SIP user agents and (2) a subscription/response mechanism, where a caller's user agent subscribes for changes in the callee's context at the SIP proxy. The proxy intercepts the incoming messages which query for the callee's context and replies on the callee's behalf.

For a query/response mechanism two methods were proposed: (1) extending the existing SIP OPTIONS mesage with a *Context* header and (2) defining a new SIP CONTEXT message. Both methods return the user's current context in the body of the 200 OK response message. The benefits and drawbacks of using both of these methods are explained in [6]. SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE) [1] has been proposed as the subscription/notification mechanism.

Since the described framework does not implement any additional functions on top of SIP and SIMPLE methods, it does not meet requirements for *aggregation* and *timely discovery of context sources*. In SIMPLE watchers (i.e., applications) and presentities (i.e., sensors) do not need to be online at the same time to exchange context, thus preventing implementation of the *activation and deactivation of remote context sensing*. Using the modified OPTIONS or CONTEXT method for this purpose could delay the 200 OK response until the context is sensed, causing after 500 ms retransmissions of this request as specified in SIP standard [18]. Delays longer than 500 ms could occur when context is sensed from a wireless sensor network (as described in [10]), instead from a sensor that is built in or connected with a high-speed data link to a user's device. Additionally, for each new context update another SIP OPTIONS or CONTEXT request has to be sent. Therefore, our *activation and deactivation of context sensing* requirement is not met.

1.2.6 Summary

We can see from Table 1 that none of the existing mechanisms fully meets the identified requirements. SIP-based context distribution satisfies these requirements to the highest degree (i.e., 10 out of 13 requirements). However, it lacks the ability to discover context sources, activate/deactivate remote context sensing, and support aggregation of context data. In order to preserve the advantages of SIP/SIMPLE mechanism while adding the missing capabilities, we propose a resource list-based subscription/notification mechanism for context sharing, described in the following section.

2. PROPOSED MECHANISM

The proposed context distribution mechanism enables subscriptions to a resource list containing URIs of multiple context sources that can provide the same context type and the delivery of aggregated notifications containing context updates from each of these sources. Figure 1 shows the infrastructure required by the proposed mechanism.

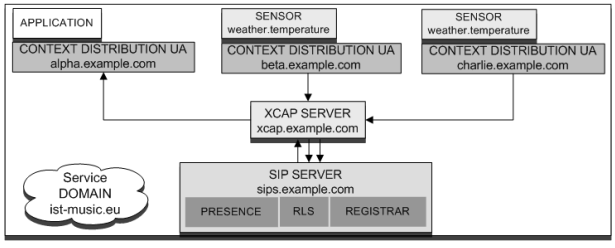


Figure 1: Distribution service using RLS and XCAP-based operations

Resource lists are stored as XML documents at an XML Configuration Access Protocol (XCAP) server, which provides a repository that can be accessed by multiple entities [17]. The SIP registrar receives registration requests and associates the user's SIP URI with one or more SIP user agents. Presence server provides the subscription/notification mechanism, while a resource list server (RLS) processes subscriptions for resource list URIs and aggregates notifications. Context distribution user agents run as SIP endpoints on three nodes connected to the Internet, providing context to and requesting context from the server side of the this infrastructure. These nodes run an application or a sensor/context source along with the context distribution agent.

Discovery of context sources is provided in both *synchronous* and *asynchronous* manners. In *synchronous* mode, the context distribution user agent (UA) retrieves from the XCAP server the resource list associated with the desired context type. If the obtained resource list is empty, the *asynchronous* mode enables the context distribution UA to subscribe to *changes* in the resource list document. These changes can occur when a context source that can provide the requested context type becomes available and *adds* its entry to the resource list, or is turned off or disconnected from the network – hence its entry is *removed* from the resource list. Our measurements (described in Section 3.2) show that it takes 4.7 ms to synchronously discover 150 sensors providing a particular context type and 2.7 ms to discover a change in the resource list and notify watcher(s), thus meeting our requirement for *timely discovery of context sources*.

The proposed mechanism allows *activation and deactivation of remote context sources* upon receiving the individual subscription/unsubscription request from the RLS, resulting in starting/terminating the sensed context updates.

Applications can select a subset of the available context sources to subscribe to (based on the *quality* of information their sensors provide) in order to receive aggregated notifications containing these sensors' context updates.

To provide the described functions, some modifications in the RLS and XCAP standards were needed:

- We introduced the concepts of *public* and *private* resource lists to distinguish between discovery of *all available* context sources and distribution of context from only the *selected* context sources to the application that chose to subscribe to these sources.

- We implemented support for the MUSIC context model [15] (in the body of SIP NOTIFY messages) that models real world entities (e.g., User, Room, Device) and their context information using the *entity* and *scope* terms. This method of context modeling is suitable for composing public resource lists as SIP URIs in the form: *sip:<entity>.<scope>@<domain>*, thus allowing easy querying for some entity's context information. As a private resource list is applicable to the specific user, its URIs is composed as: *sip:<username>@<domain>*. Context sources of one node have URIs of the form: *sip:<username>@<domain>;metadataId=<entity>.<scope>* that are inserted as entries in the resource lists.

- We created customized authorization mechanisms for access control and management of public resource lists to allow context sources to *add* or *remove* their entries to/from the public resource list, which can be read by everyone. In private resource lists, we use the same authorization mechanism as in the default authorization policy [17] that allows watchers to read, write, or modify their own private resource lists.

- We provide a partial notification of changes in the public resource lists using the xcap-diff SIP event package [20]. This notification contains the context source entry that has been *added* to or *removed* from a public resource list and the patch which when applied, enables watchers to modify their private resource lists accordingly and perform a re-subscribe action.

2.1 Testbed: Software

Our testbed is shown in Figure 2. We extended OpenSIPS [11] and OpenXCAP [12] with the modifications described above and implemented the context distribution service in Java using the JAIN SIP stack.

The server side of the infrastructure is deployed on a single node, because the OpenSIPS in combination with the non integrated XCAP server could not provide fast enough partial notifications. Such an integrated mode was also proposed by the OpenXCAP implementors [12]. As a result, the SIP registrar, RLS, and XCAP server share some database tables that store subscribers and resource lists. Additionally, the XCAP server invokes a function in the SIP proxy server to notify watchers about changes in a public resource list, using Inter-process communication. The context distribution service communicates with the SIP proxy server using SIP/SIMPLE and with the XCAP server using HTTP GET, PUT, and DELETE methods.

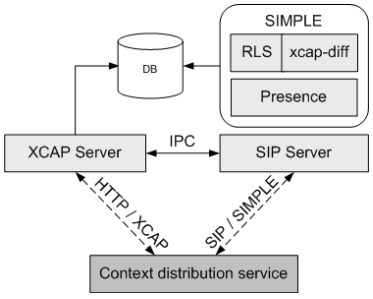


Figure 2: Testbed

From configurable parameters we used in OpenSIPS the maximum buffer size of 256 KB, the RLS timer (for waking up) of 1 second, and each context distribution sensor published its sensor's context updates once in 3 seconds.

2.2 Testbed: Hardware

Experiments were performed on three computers in an unloaded, isolated 100 Mbps wired local area network. The hardware configuration of each node is presented in Table 2. A client and sensor load generator were used to generate multiple contextities and watchers in order to load the server side infrastructure with SIP and XCAP operations while measuring these servers' performance locally on the server machine. Load generators were implemented in Java on top of the context distribution service.

A client load generator simulated 1 watcher that registers at the SIP server, queries the XCAP server for the public resource list, and uploads entries from the public resource

list into its private resource list. Next, it subscribes to receive aggregated context from the sensors in this private resource list and for changes in the availability of these sensors. A sensor load generator simulated various number of contextities that were created as separate threads and registered at the SIP server at the startup. Upon receiving subscription requests, each thread processed the corresponding request, activated the corresponding context sources to publish their context events, and sent context from these events as individual NOTIFYs back to the SIP server performing aggregation. An aggregated NOTIFY was received by the client load generator at the end of each aggregation period.

	SIP & XCAP server	Sensor generator	Client generator
Device	Fujitsu Siemens Celsius M420	Dell XPS M1530	Acer Aspire 5021
OS	Ubuntu 8.04.2	MS Windows Vista Ultimate	MS Windows XP Professional
CPU	2x Intel Pentium 4 @ 2.60Ghz	Intel Core 2 Duo CPU T8300@2.4GHz	AMD Turion 64 Mobile Technology ML-28@1.6GHz
RAM	1GB	3GB	1GB
NIC	Intel 82547EI Gigabit Ethernet Controller	Marvell Yukon 88E8040 PCI-E Fast Ethernet Controller	Realtek RTL 8169/8110 Family Gigabit Ethernet

Table 2: Hardware used in the testbed

3. PERFORMANCE EVALUATION

3.1 Measurements description

The purpose of this performance evaluation was to: **(1)** assess the latency of different context distribution activities, **(2)** measure the time needed by RLS to aggregate context updates and send an aggregated NOTIFY to the watcher, and **(3)** compare this aggregation time with the time needed to deliver the same number of NOTIFYs using SIMPLE, in order to determine if it pays off to perform context aggregation and for what number of sensors.

To achieve the **first goal**, we measured the average response time that is needed to: (1) register a context source URI at the SIP registrar and XCAP server, (2) discover available context sources providing the desired <entity, scope> pair in synchronous and asynchronous manner, and (3) upload the private resource list at the XCAP server, aggregate the latest context updates & send the aggregated NOTIFY to the application that subscribed to a given context. We used Wireshark for performing these measurements.

For the **second goal**, we inserted the timing logs into the RLS module of the OpenSIPS code and measured the aggregation time with time stamp counter, in order to obtain high precision time logs. We measured the aggregation time with one client and for different number of sensors (ranging from 1 to the maximum number of sensors whose context could be aggregated in the single notification, due to the limit of the UDP packet size (i.e., 64 kB)).

To determine the number of sensors for which it pays off to perform aggregation, we measured the time needed by SIMPLE to deliver individual NOTIFYs and compared it to the previously obtained RLS aggregation time.

3.2 Evaluation

The pie chart in Figure 3 illustrates duration of different activities performed by the proposed distribution mechanism, using a single sensor. It can be observed that context distribution consumed the largest portion of the time.

Note that all the measurements results are represented by medians to best represent the data set and avoid the effects of outliers. Each measurement was repeated 100 times to minimize the random error.

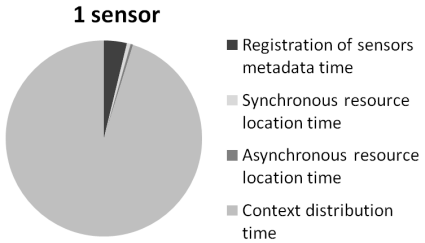


Figure 3: Activities performed by context distribution mechanism (1 sensor)

Looking at the individual context distribution activities for a single sensor (shown in Figure 4), we can observe that the largest portion of time is taken by the RLS aggregation process. RLS wakes up every second to aggregate updated NOTIFYs. These NOTIFYs are sent from another computer at random times with the respect to the clock of the machine where the RLS is running and these clocks are not correlated. When a single sensor is used, the average time to wake up and aggregate NOTIFYs is 493.3ms out of 527.6ms consumed by all context distribution activities. **This** motivated us to further investigate the cost of RLS aggregation.

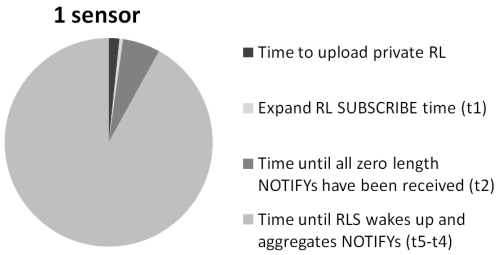


Figure 4: Context distribution activities (1 sensor)

To learn the details of this RLS aggregation, we measured the aggregation time by inserting timing logs into the OpenSIPS code. The aggregation time was measured with one client and various numbers of sensors (ranging from 1 to 166). The length of context data carried in the body of each individual notification was 43 bytes, resulting in aggregation of maximally 166 of such context updates into the UDP buffer (the buffer was extended from 8 to 64Kbytes). The measurements and linear fits to these data are shown in Figure 5. This graph also shows the time needed by SIMPLE to deliver the same number of individual NOTIFYs. It can be observed that it takes 10.2 ms to aggregate 166 NOTIFYs. Using SIMPLE, it would require 157 ms to deliver NOTIFYs from 166 sensors.

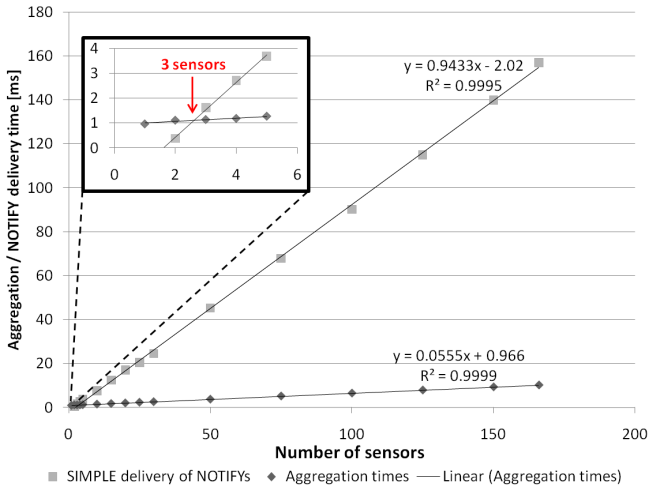
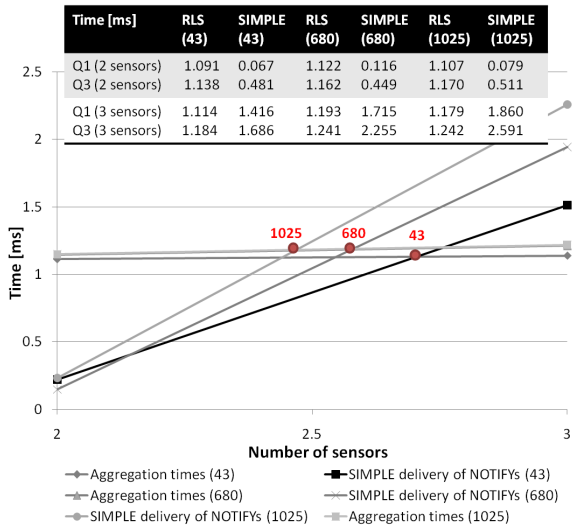


Figure 5: Comparison of SIMPLE delivery time of NOTIFYs and RLS aggregation time



Time [ms]	RLS (43)	SIMPLE (43)	RLS (680)	SIMPLE (680)	RLS (1025)	SIMPLE (1025)
Q1 (2 sensors)	1.091	0.067	1.122	0.116	1.107	0.079
Q3 (2 sensors)	1.138	0.481	1.162	0.449	1.170	0.511
Q1 (3 sensors)	1.114	1.416	1.193	1.715	1.179	1.860
Q3 (3 sensors)	1.184	1.686	1.241	2.255	1.242	2.591

Figure 6: Comparison of SIMPLE and RLS using 2 and 3 sensors generating different context sizes

At the crossing of these curves, we see the number of sensors for which RLS and SIMPLE notification times become equal, and to the right of this point RLS aggregation performs better than SIMPLE. The enlarged view of the data near the origin enables us to conclude that for **three or more sensors** it pays off to perform aggregation. This is an interesting result, because prior to making any measurements, we expected this treshold to be significantly larger.

Next, we performed the same measurements with a context size of 680 bytes, in order to determine the aggregation time behavior when the context size increases. The maximum number of context updates which the 64 Kbytes UDP buffer could hold was 62 and the time to aggregate 62 updates was *half* the time required to aggregate 166 updates of 43 bytes length (i.e., 5.13 vs 10.2 ms). However, after increasing the content length from 43 to 680 bytes, the RLS aggregation and the SIMPLE notification delivery time increased at a similar pace (119.3% and 115.7% with respect to the corresponding times at 43 bytes length). As a result, the number of sensors for which the aggregation outperforms the SIMPLE remained the same (i.e., 3 sensors).

In order to get an indication for how many sensors aggregation pays off with a further increase in the size of context updates, we repeated RLS and SIMPLE measurements with context sources generating updates of 1025 bytes. Note that 1025 bytes was the maximum size of context carried in 3 NOTIFYs that we could send without changing configuration parameters. Figure 6 indicates that the cross over point in the number of sensors slightly decreases towards 2, which tells us that RLS becomes more efficient than the SIMPLE when increasing context length. First (Q1) and third (Q3) quartiles in Figure 6 indicate dispersion for each data set.

To find out which processes affects the aggregation time and to what extent, we measured the duration of aggregation activities as a function of the number of sensors for the case of 166 sensors, each with 43 bytes context length. Figure 7 illustrates that the *most* of the aggregation time is consumed by *database (DB) operations* (i.e., 59%), in particular 24% by the query and 35% by the update database operations.

The database operations are used as follows: when the RLS wakes up, it *queries* the RLS presentity table for the updated context and after selecting and copying the records for aggregation, it *updates* these records with a NOT UPDATED flag, initializing them for the next round of context updates and aggregation. The rest of the aggregation time is spent on aggregation of context data into the NOTIFY multipart body and sending the aggregated NOTIFY to the watcher (i.e., 37% and 4% of the total aggregation time).

Next, we wanted to know how an increase in context size affects individual aggregation activities, thus we measured the duration of these activities during aggregation of 50 sensors updates, each with 43 bytes and 680 bytes length. Figure 8 presents these measurement results, showing that the database operations time increased the most (i.e., for 300 μs), followed by the increase in time to aggregate context data (i.e., 162 μs). The least increase in time (i.e., 92 μs) had sending of an aggregated notification to a watcher.

Since we identified database operations to be the major *bottleneck* during the context aggregation, we repeated the

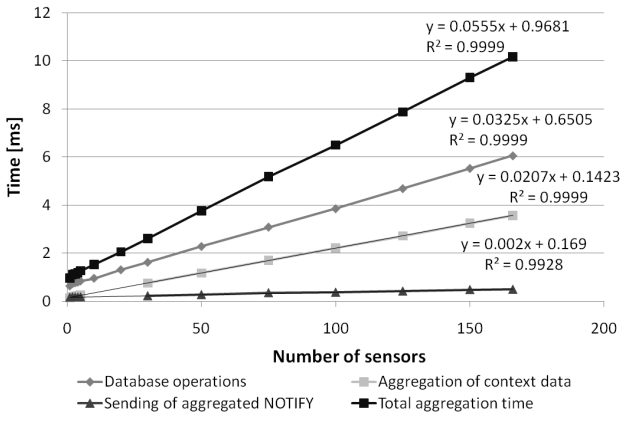


Figure 7: Duration of aggregation activities when increasing the number of sensors

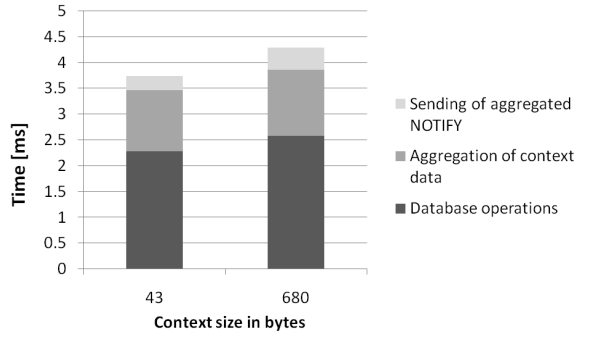


Figure 8: Duration of aggregation activities when increasing the size of context updates

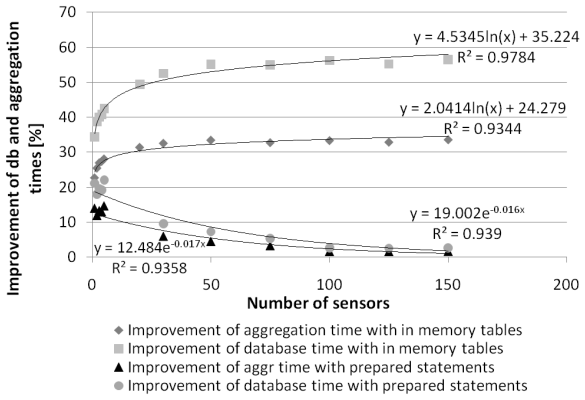


Figure 9: Reduction of DB and aggregation times

same experiments using the *prepared statements* and then using the *in-memory tables & prepared statements*.

The use of *prepared statements* reduced the database time for about 152 μs. Although this 152 μs might be a significant improvement for 5 sensors or less (i.e., 20%), for 100 or more sensors this improvement is only 3% (see Figure 9).

To measure the duration of these same operations using the *in-memory tables and prepared statements*, we had to recreate the RLS presentity table as an in-memory table. However, because in-memory tables do not support the BLOB type, we replaced it with the VARBINARY type with maximum sizes of 43 and 680 bytes. To check if there is any difference in performance when using these two datatypes, we measured the time to query and update the context state from/to the MySQL table using both datatypes. Table 3 shows that using the VARCHAR datatype instead of BLOB does not degrade the database performance results.

Average time to perform	43 bytes	680 bytes
UPDATE operation using BLOB	757.4 μs	959.6 μs
UPDATE operation using VARCHAR	718.6 μs	916.2 μs
SELECT operation using BLOB	805.6 μs	773,7 μs
SELECT operation using VARCHAR	778.1 μs	784.8 μs

Table 3: Comparison of query and update operation times using BLOB and VARCHAR datatypes

The use of in-memory table and prepared statements logarithmically reduced the database times by up to 57 %, resulting in reduction of aggregation time up to 34% (see Figure 9). This database time reduction is efficient for supporting a large number of sensors (>50), however it is insufficient to move our treshold of 3 sensors to 2 sensors. Therefore, as part of future work, it should be examined if further increases of context lengths when used together with in-memory tables, can shift this boundary to 2 sensors, thus making an even stronger case for using context aggregation in *any* context-aware application.

3.3 The benefit of aggregation

We fit curves for RLS aggregation time and SIMPLE delivery time of NOTIFYs as linear functions of the number of sensors. To the right of the intersection of these curves (as shown in Figure 5) we can see the number of sensors for which RLS aggregation performs better than SIMPLE. In order to quantify the effectiveness of aggregation when compared to the SIMPLE delivery of individual NOTIFYs, we define **the benefit of aggregation** as the ratio of the SIMPLE delivery time and the RLS aggregation time that specifies how many times RLS aggregation is faster than the SIMPLE delivery of individual NOTIFYs.

Using the fixed parameters (i.e., the frequency of context updates once every three seconds, aggregation time period of one second, the UDP buffer size of 64kB, and the context updates size of 43 bytes), we found that the RLS aggregation pays off for 3 or more sensors (i.e., that the benefit of aggregation equals to 1 for 3 sensors and increases with an increase in the number of sensors). However, we would like to derive how would the benefit of aggregation change due to any of the factors affecting it. To do this, we created an analytic model of the relationship of the parameters on which RLS aggregation time and SIMPLE delivery time depend.

Abstracting the fitting curves functions for the generic scenario, we can express the RLS aggregation time (t_A) as a function of the number of aggregated context updates in a single aggregated NOTIFY within an observed aggregation period t (i.e., $N_{AGG}(t)$). Similarly, the SIMPLE delivery time (t_S) can be represented as a function of the number of individual NOTIFYs generated by the SIP server during the same period t (i.e., N(t)). The resulting functions are:

$$t_A(N_{AGG}(t)) = a * N_{AGG}(t) + b \quad [s] \tag{1}$$

$$t_S(N(t)) = c * N(t) + d \quad [s] \tag{2}$$

In the above, T_{AGG} denotes an aggregation time period, t represents any time period identified by $n*T_{AGG}$, $n \in \mathbb{N}$, and parameters a, b, c, d are empirically derived from the measurements, depending on the size of context updates. As these parameter fits were derived from measurements where the size of all updates was equal, we need to assume (in both scenarios) that all context updates are of the same size.

For this model we assumed that each sensor can send its context update with some frequency f_i (where i denotes the sensor identifier, $i \in \mathbb{N}$). The same frequencies are used in the RLS and SIMPLE scenario, to be able to compare the RLS aggregation time and SIMPLE delivery time. For reasons of simplicity, this model assumes that all sensors start sending their context updates within the first aggregation period.

$N_{AGG}(t)$ can be calculated as the sum of the frequencies at which updates from a given sensor arrive (f_i) multiplied by a given aggregation time period ($t=n*T_{AGG}$, $n \in \mathbb{N}$):

$$N_{AGG}(t) = f_1 * t + \cdots + f_N * t = \sum_{i=1}^{N} f_i * t = n * T_{AGG} * \sum_{i=1}^{N} f_i \quad (3)$$

Additionally, an aggregated NOTIFY size has to be smaller than or equal to the UDP buffer size:

$$agg\_notify\_size = \sum_{i=1}^{N} m\_part\_size_i + rlmi\_size \approx N * m\_part\_size + rlmi\_size \leq UDP\_buffer\_size \quad (4)$$

In the above equation m_part_size and rlmi_size denote the size of the multipart/related content, an XML document providing meta information about each resource in the list (including the context data from each sensor), and the size of Resource List Meta-Information (RLMI) document, which is a root document of the multipart/related body, respectively.

In SIMPLE, N(t) can be expressed as a mean rate at which the SIP server generates individual NOTIFYs (λ) multiplied by the observed aggregation period (t):

$$N(t) = \lambda * t = \frac{t}{t_n} \quad (5)$$

where t_n represents the mean inter-arrival time of individual NOTIFYs generated by the SIP server.

In a specific case where all sensors send their updates with the same frequency f (as was the case in our measurements):

$$N_{AGG}(t) = f * t \quad (6)$$

Note that each received context update from a given sensor overwrites the earlier received value in the database. Therefore, if a sensor sends more than one update within an aggregation time period, only the latest context update from this sensor will be aggregated. Thus, in this case, we will calculate f_i* t as 1 in the equation (3) for N_{AGG}(t):

$$f_i * t = 1 \; update, i \in \mathbb{N} \quad (7)$$

In order for an aggregated NOTIFY to contain context updates from each sensor every aggregation period, the frequency at which sensors should send their context updates has to be greater than the frequency of aggregation:

$$N_{AGG}(t) = N \quad \text{if} \quad \forall i \; f_i > 1/T_{AGG} > f_{AGG}, i \in \mathbb{N} \quad (8)$$

Considering the assumption that all sensors start sending their context updates within a first aggregation period, if one or more frequencies of context updates (f_i) are lower than the frequency of aggregation (i.e., $f_i < f_{AGG}$), then in order to calculate N_{AGG}(t), we have to consider the following:

- if $0<f_i*t<1$, then no update was received during this period of t seconds $\rightarrow$ $f_i*t=0$.
- if $f_i*t \in \mathbb{N}$, then a context update from a corresponding sensor has been received and 1 should be added to $N_{AGG}(t) \rightarrow f_i*t=1$.
- if $f_i*t>1$, $f_i*t \in \mathbb{R}$, and if it can be expressed as $x+y/T_i$, where $x,y \in \mathbb{N}$ and $y<T_i$, then f_i*t will be equal to 1 only if $x*T_i$ lies in the interval ($t-T_{AGG}$, t) and 0 otherwise. Note that the round brackets mean that this interval excludes the points $t-T_{AGG}$ and t, because when context updates arrive in t, f_i*t is a natural number, while they cannot arrive at $t-T_{AGG}$ because this belongs to the previous $(n-1)^{st}$ aggregation interval.

After inserting $t=n*T_{AGG}$ into $f_i*t = x+\frac{y}{T_i}$, we get:

$$x = \frac{n * T_{AGG} - y}{T_i}, n \in \mathbb{N} \quad (9)$$

As the length (Δ) of the interval ($t-T_{AGG}$, t)=(($n-1$)*T_{AGG}, n*T_{AGG}) is T_{AGG}, we can write:

$$n * T_{AGG} - y = (n-1) * T_{AGG} + k, k \in [1, \Delta - 1] \quad (10)$$

Note that k starts from 1 and finishes with Δ-1 because the interval does **not** include $(n-1)*T_{AGG}$ or $n*T_{AGG}$.

From (10) we obtain that:

$$x = \frac{T_{AGG} * (n+1) + k}{T_i}, y = T_{AGG} - k, k \in [1, \Delta - 1] \quad (11)$$

We can conclude that, if for a given frequency f_i and the observed aggregation period t the condition (11) is satisfied, a context update from a corresponding sensor has been received and 1 should be added to N_{AGG}(t).

Now that we have a way of calculating f_i*t for all the frequencies of arriving context updates, we can express the RLS aggregation time as a function of the aggregation period t (i.e., t_A(t)). After inserting (3) into (1) we obtain:

$$t_A(t) = a * \sum_{i=1}^{N} f_i * t + b = a * n * T_{AGG} * \sum_{i=1}^{N} f_i + b \quad (12)$$

Similarly, the SIMPLE delivery time can be represented as a function of the aggregation period t (i.e., t_S(t)):

$$t_S(t) = c * \lambda * t + d = c * \frac{n * T_{AGG}}{t_n} + d \quad (13)$$

In both equations (12) and (13) $t=n*T_{AGG}$, $n \in \mathbb{N}$.

In order to find the number of sensors for which the RLS aggregation outperforms SIMPLE, the RLS aggregation time has to be less than or equal to the SIMPLE delivery time.

$$t_A(t) \leq t_S(t) \quad (14)$$

We define the benefit of aggregation in a specific aggregation period B_A(t) as the ratio of t_S(t) and t_A(t), which according to (15) has to be greater than or equal to 1:

$$B_A(t) = \frac{t_S(t)}{t_A(t)} \geq 1 \quad (15)$$

B_A(t) can be interpreted as follows: when t_S/t_A is equal to 1, the benefit of aggregation is the lowest because the time to aggregate NOTIFYs is the same as the time needed by SIMPLE to deliver the same number of individual NOTIFYs. Consequently, as a result of this relation we obtain the minimum number of sensors for which the RLS aggregation

pays off. In other cases, when t_S/t_A is greater than 1, the benefit of aggregation increases as the time to perform aggregation is t_S/t_A times shorter than the time to deliver the same number of individual NOTIFYs, thus performing aggregation pays off to a greater extent.

Inserting (12) and (13) into (14) results in:

$$a * n * T_{AGG} * \sum_{i=1}^{N} f_i + b \leq c * \frac{n * T_{AGG}}{t_n} + d \qquad (16)$$

In order to calculate the minimum number of sensors for which it pays off to perform aggregation, N(t) has to be equal to $N_{AGG}(t)$, thus the rate at which NOTIFYs are generated (λ) must be equal to the sum of the frequencies at which sensors send their context updates:

$$\lambda = \frac{1}{t_n} = \sum_{i=1}^{N} f_i \qquad (17)$$

Finally, expressing $B_A(t)$ using (16) defines a relationship between the benefit of aggregation and the different factors affecting it:

$$B_A(n * T_{AGG}) = \frac{c * \frac{n * T_{AGG}}{t_n} + d}{a * n * T_{AGG} * \sum_{i=1}^{N} f_i + b} \geq 1 \qquad (18)$$

From (18) it can be concluded that the benefit of aggregation increases with a significantly smaller mean inter-arrival time of individual NOTIFYs generated by the SIP server (t_n) and a significantly smaller sum of the frequencies of individual context updates that are sent by sensors – both when compared to the selected period of aggregation ($t=n*T_{AGG}$).

Note that the accuracy of B_A depends on how good the fits of $t_S(t)$ and $t_A(t)$ are through the given data points. As to calculate the value of t_A in (18) only fixed values are used and a variable on which t_S depends is t_n, therefore it is important to calculate t_n from the SIMPLE notification delivery time that has the lowest residual value. The values for t_n are 0.905 ms (in the case of 43 bytes) and 0.922 ms (in the case of 680 bytes), resulting in B_A equal to 0.97 (in the case of 43 bytes) and 0.9 (in the case of 680 bytes).

As in our measurements all sensors were sending context updates with the same frequency f (of one update every 3 seconds), thus we can derive from (17) the minimum number of sensors for which the RLS aggregation pays off (N_{min}):

$$N_{min} = \frac{1}{f * t_n} \qquad (19)$$

Using the same values for t_n as in the calculation of B_A, we obtain N_{min}(43 bytes)=3.32 and N_{min}(680 bytes)=3.25. This result analytically proves that the minimum number of sensors for which the RLS aggregation pays off is 3 and that with an increase in the size of context updates, this number decreases. Note, however, that t_n increases with an increasing number of generated context updates, because some PUBLISH messages are waiting in a queue while other messages are being processed by the SIP server, resulting in slower generation of individual NOTIFYs. Therefore, by calculating t_n value from SIMPLE delivery times of 150 notifications, we would obtain lower N_{min} values (i.e., N_{min}(43 bytes)=3.2 and N_{min}(680 bytes)=2.8).

4. CONCLUSIONS

This article discussed the existing context distribution mechanisms according to identified requirements and proposed a resource list-based subscription/notification mechanism for context sharing. The important feature of this mechanism is the aggregation of context, as it can reduce the amount of network traffic in comparison to individual context updates. This paper started with the question: Does aggregation pay off? Our measurements of the proposed distribution mechanism show that the answer is "yes for 3 or more sensors". Our experiments give an indication that this boundary becomes closer to 2 sensors with an increase of size of context updates and the use of in-memory tables & prepared statements. Therefore, as part of our future work we plan to conduct more experiments to be able to confirm this indication.

We also plan to repeat the aggregation measurements with sensors publishing context updates of different sizes at different frequencies, and observe how this impacts the aggregation time and the number of NOTIFYs being aggregated.

5. ACKNOWLEDGMENTS

We would like to acknowledge the partial financial support given to this research by the European Union (EU FP6 MUSIC-IST project, contract number 35166). We would also like to thank Prof. Gerald Q. Maguire Jr., for the fruitful discussions & comments during this research work and to one of the anonymous reviewers for many valuable comments.

6. REFERENCES

[1] SIP for Instant Messaging and Presence Leveraging Extensions (simple) RFCs. *Available at http://www.voip-telephony.org/rfc/simple*, last accessed in September 2010.

[2] C. Angeles. *Distribution of Context Information using the Session Initiation Protocol (SIP)*. Royal Institute of Technology, Stockholm, Sweden, December 2008.

[3] J. E. Bardram. The Java Context-Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In *3rd International Conference on Pervasive Computing (Pervasive 2005)*, pages 98–115. LNCS, May 2005.

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

[5] A. K. Dey. Providing Architectural Support for Building Context-Aware Applications. Technical report, Georgia Institute of Technology, November 2000.

[6] M. Görtz, R. Ackermann, and R. Steinmetz. Enhanced SIP Communication Services by Context Sharing. In *Proceedings of the 30th Euromicro Conference*, pages 272–279. Rennes, France, September 2004.

[7] H. Gümüşkaya, A. V. Gürel, and M. V. Nural. Architectures for Small Mobile Communication Devices and Performance Analyses. In *Proc. of the 1st International Conference on Applications of Digital Information and Web Technologies (ICADIWT)*, pages 342–347. Ostrava, Czech Republic, August 2008.

[8] E. Halepovic and R. Deters. The JXTA Performance Model and Evaluation. *Future Generation Computer Systems*, 21(3):377–390, March 2005.

[9] A. Houri, T. Rang, and E. Aoki. Problem Statement for SIP/SIMPLE. *IETF Internet draft*, October 2006.

[10] J. Y. Khan, M. R. Yuce, and F. Karami. Performance Evaluation of a Wireless Body Area Sensor Network for Remote Patient Monitoring. In *Proc. of the 30th Annual International Conference of the IEEE Enginering in Medicine and Biology Society (IEEE EMBC08)*, pages 1266–1269. Ostrava, Czech Republic, August 2008.

[11] OpenSIPS. Open source implementation of a SIP server. *http://www.opensips.org*, last accessed in September 2010.

[12] OpenXCAP. Free XCAP server for SIP SIMPLE (RFC 4825). *http://openxcap.org*, last accessed in September 2010.

[13] N. Paspallis, R. Rouvoy, P. Barone, G. A. Papadopoulos, F. Eliassen, and A. Mamelli. A Pluggable and Reconfigurable Architecture for a Context-aware Enabling Middleware System. In *Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA08)*, pages 553–570. LNCS, November 2008.

[14] D. Pavel and D. Trossen. Context Provisioning and SIP events. In *MobiSys 2004 Workshop on Context Awareness*. Boston, Massachusetts, USA, June 2004.

[15] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, J. Lorenzo, M. Valla, C. Fra, N. Paspallis, and G. A. Papadopoulos. A Comprehensive Context Modeling Framework for Pervasive Computing Systems. In *Proceedings of 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 281–295. LNCS, June 2008.

[16] O. Riva. Contory: A Middleware for the Provisioning of Context Information on Smart Phones. In *Middleware 2006*, pages 219–239.

[17] J. Rosenberg. The Extensible Markup Language (XML) Configuration Access Protocol (XCAP). *IETF RFC 4825*, May 2007.

[18] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. *IETF RFC 3261*, June 2002.

[19] B. N. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *1st International Workshop on Mobile Computing Systems and Applications*, pages 85–90. Santa Cruz, CA, USA, December 1994.

[20] J. Urpalainen and D. Willis. An Extensible Markup Language (XML) Configuration Access Protocol (XCAP Diff Event Package. *IETF RFC 5875*, May 2010.

[21] K. F. Yeung and Y. Yang. Mobile Information Retrieval in a Hybrid Peer-to-Peer Environment. In *Proc. of the 6th International Conference on Mobile Technology, Applications, and Systems*. Nice, France, September 2009.