

# Programmable Presence Virtualization for Next-Generation Context-Based Applications

Arup Acharya\*, Nilanjan Banerjee†, Dipanjan Chakraborty†, Koustuv Dasgupta†,  
Archan Misra\*, Shachi Sharma†, Xiping Wang\* and Charles P. Wright\*

†IBM Research, India Research Lab, New Delhi, India

\*IBM Research, T. J. Watson Research Center, Hawthorne, NY, USA

**Abstract**—Presence, broadly defined as an event publish-notification infrastructure for converged applications, has emerged as a key mechanism for collecting and disseminating context attributes for next-generation services in both enterprise and provider domains. Current presence-based solutions and products lack in the ability to a) support flexible user-defined queries over dynamic presence data and b) derive composite presence from multiple provider domains. Accordingly, current uses of context are limited to individual domains/organizations and do not provide a programmable mechanism for rapid creation of context-aware services. This paper describes a presence virtualization architecture, where a Virtualized Presence Server receives customizable queries from multiple presence clients, retrieves the necessary data from the base presence servers, applies the required virtualization logic and notifies the presence clients. To support both query expressiveness and computational efficiency, virtualization queries are structured to separately identify both the XSLT-based transformation primitives and the presence sources over which the transformation occurs. For improved scalability, the proposed architecture offloads the XSLT-related processing to a high-performance XML processing engine. We describe our current implementation and present performance results that attest to the promise of this virtualization approach.

**Keywords:** Presence, virtualization, scalability, federation

## I. INTRODUCTION

While initially developed as a means for communicating the “online status” in instant messaging applications, *presence* has become a key enabler of Web-based *content provider* (e.g., Google Talk™, Yahoo! Messenger™ or Skype™), *enterprise* (e.g., IBM Sametime™) and *service provider/telco* (e.g., Push-to-talk) converged applications. Indeed, presence is rapidly evolving to become the *de-facto* method of representing and querying the context of an individual, both physical (e.g., a user’s location) and virtual (e.g., the status of avatars visiting my “island” in SecondLife). Moreover, presence is used to represent the dynamic attributes of not just individuals, but also *devices* (e.g., the battery level of a cellphone) and *abstract entities* (e.g., the number of attendees in a conference call). Presence may be broadly described as a publish-subscribe system for context, that currently enables a variety of products and applications (ranging from location tracking, to real-time discovery of available experts for collaboration, to business process-enablement). As such, presence embodies the first *practical, large-scale adoption of context-aware computing*.

With the proliferation of presence, an individual’s contextual state is increasingly fragmented across different applications

and provider domains; currently, presence-based applications operate in domain-specific silos, unaware of the individual’s presence status in other domains. Obfuscating these traditional barriers between communications service providers, enterprises and Internet content providers will, however, enable a significantly more unified and accurate view of an individual’s presence attributes across multiple domains. For example, an employee’s activity status cannot be accurately derived just from the enterprise-sanctioned Presence system (e.g., Sametime within IBM), as this infrastructure is unable to capture the fact that she may be using her cell-phone (from an external telco). More generally, future converged applications not only require the presence status from multiple sources/domains, but also effectively operate over *derived contextual attributes* by applying some processing logic over the raw presence information. For example, a call-center (Helpdesk) monitoring application may be interested in the percentage of call-center employees who are available, rather than the presence status of individual employees.

Current presence solutions are largely based on SIMPLE [5, 8] extensions to the base SIP signaling protocol (with Google Talk being a notable exception that utilizes the XMPP [4] protocol). In the SIP-based presence model, an application server called the Presence Server (**PS**) acts as the central repository for a specific domain (a specific organization or application) where presence information generated by SIP clients (via a PUBLISH message) belonging to that domain is matched against prior subscriptions issued (via a SUBSCRIBE message) by “watcher” [5] clients; the PS informs such watchers of changes in presence states (via a NOTIFY message). In the standard SIMPLE model, subscriptions and publishes are indexed using a single SIP URI, where each URI is typically associated with a unique entity (called *presentity*), such as a user or device. Consequently,

- subscribers can only specify an individual subscription over a single presentity (e.g., subscribe to the URI sip:alice@us.ibm.com); the URI restriction applies, even when group subscription mechanisms (such as the use of resource-lists [6]) are considered.
- the subscription logic over the content of individual URIs is restricted to a limited set of pre-defined “filter” operators specified in SIP standards [8] (e.g., alerts only on specified changes in the location value).

Accordingly, a monitoring application interested in the overall status of a call center Helpdesk must subscribe (via the presence server) to the status of each individual call center employee and perform the necessary aggregation locally. This approach not only wastes network resources, but also precludes multiple clients from being able to “reuse” the same computation.

We believe that these characteristics present serious limitations to the deployment of a large-scale, scalable presence infrastructure for future converged applications. In particular, to effectively support progressively more sophisticated uses of presence, we believe it is necessary to build a *presence virtualization layer*. The virtualization layer provides a *programmable* abstraction by which applications can easily obtain their desired collective “view” of presence by querying a server-side overlay, without focusing on the details of individual presentities. The base presence technologies do not provide a standardized and scalable mechanism for querying and *customizing* aggregate views over presence; at best, current products offer a means for pre-defined aggregation of the presence information of an explicitly identified user across several domains. While it may be tempting to view presence virtualization merely as another instance of generic “context aggregation”, practical presence aggregation and its use by multiple presence-enabled applications must factor in two key challenges that are not addressed by current approaches:

- *Query Flexibility*: Since virtualization is a common service spanning multiple presence applications, the client programming model must be expressive enough to support a wide variety of virtualization queries (for example, both a query that computes the percentage of available call-center employees, as well as another application that monitors the number of free taxicabs within a mile of a train station).
- *Scalability*: given the high volumes of presence updates and queries to be expected in tier-1 service provider and enterprise environments, the solution should control both the network traffic (in terms of presence updates and notifications) and the server processing (in terms of both subscriptions and the aggregation logic) loads. Scalable virtualization is critical, for example, to a telecom service provider that inject a unique set of presence attributes into a larger federated presence eco-system (e.g., a cellular provider supplying real-time location of an user to Yahoo, for use in location-aware advertising).

Given this background, this paper presents our development of a novel Programmable Presence Virtualization solution, based on the fundamental ability to apply user-specific customized processing logic on a potentially large set of dynamically changing XML documents. The concept of presence virtualization is intimately linked with manipulation of XML streams, as the presence status for different objects is typically represented via XML-based schemas (such as, PIDF [2] format for SIP-based presence and presence format [4] for XMPP-based presence). Virtualization thus allows a presence client

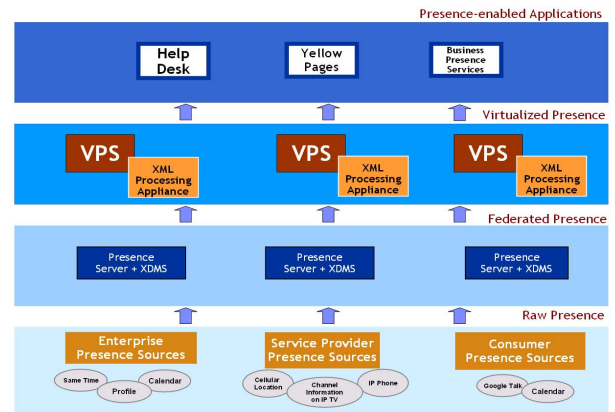


Fig. 1. High Level Presence Virtualization Architecture.

to “programmatically push” its application-specific logic, for deriving composite presence state (from the presence-related attributes of multiple individual presentities) onto the backend server infrastructure; this ability to “combine” the application of such logic from multiple clients promotes scalability by reducing both the subscription load on individual Presence Servers and the presence traffic load on the network. Moreover, our virtualization solution also allows clients to *expose and share* the end results of their transformations with other relevant clients; in effect, virtualization allows presence consumers to define *virtual presentities* (presentities created in response to external queries), which become a seamless part of the presence infrastructure and are functionally indistinguishable from the ‘raw’ presentities. The complex processing needed to support virtualization primitives at the backend infrastructure (which now not only deals with the basic publish-subscribe presence primitives, but must perform the added virtual presentity computations) can, however, become a serious *processing bottleneck*. We shall address this challenge by appropriate offloading of the more complex aspects of XML manipulation to a product-grade XML acceleration engine.

#### A. Key Contributions

The following are the key contributions of this paper:

- We motivate the importance of presence virtualization as a generic programmable framework for practical “context awareness” and then introduce the notion of a Virtualized Presence Server (VPS) that implements this virtualization logic, shielding clients from the diversity of underlying presence servers and protocols.
- We detail the implementation and design of the VPS, with special emphasis on a) how client queries are structured and specified, b) how virtual presentities and dynamically generated SIP URIs can be coupled with standard SIP Redirection mechanisms to allow presence clients to reuse the offloaded computation logic at different granularities, and (c) how the offloaded

computation may be efficiently performed at a VPS, through appropriate coordination of pipelined or parallelized XML transformations on a commercial XML acceleration engine.

- We present experimental results that demonstrate the feasibility of our current solution, and identify additional features requiring enhancements as part of ongoing investigations.

The rest of the paper is as follows. Section II summarizes the prior work related to scalable presence composition. Section III provides the high-level presence virtualization architecture. Section IV describes how the VPS operates with existing products, while Section V discusses the specifics of the VPS implementation. In Section VI, we present VPS performance results derived from our testbed. Finally, Section VII concludes the paper.

## II. RELATED WORK

The extensions to the SIP base protocol for supporting presence are specified in [5], while the most commonly used XML-based PIDF format for representing presence content is described in [2]. XMPP [4] provides an alternative messaging architecture for disseminating presence (e.g. in Google Talk) – however, the presence content is still encapsulated within XML streams.

A limited number of research prototypes have been previously suggested for offering advanced presence composition and services. YooHoo! [13] demonstrates a Web-service based solution towards presence composition (for each individual presentity), with presence clients issuing XQuery based composition requests to the backend server. Unlike our solution, there is no specific focus on achieving scalability via XML offload; moreover, the server treats each XQuery independently, without attempting to reuse common computational components. PASTA [18] described a rule-based correlation engine that augments a PS with the ability to derive higher-level presence attributes for a presentity from underlying data. Unlike our solution, PASTA does not allow for user-customized presence queries (only pre-specified rules are permitted) and applies customized logic (JAVA code) to presence data – thus lacking the potential scaling benefits obtained by our approach.

There have been a number of studies on context aware queries for pervasive and ubiquitous systems. Context query languages [15], [9] propose composing pervasive data, but does not provide a framework for optimal subscriptions to composite events in a pervasive system. Solar [14], provides a graph based abstraction for context aggregation and dissemination that enables application to subscribe to events corresponding to changes in contextual information in a flexible and scalable fashion. Solar, however, lacks programmability for aggregation and dissemination of contextual data. A flexible self-adaptable query service for getting contextual information from distributed database repository has been proposed in [12]. But, the query service does not provide the facility of subscription to a contextual query and its re-utilization

between multiple queries. Apart from these works, there have been a host of data aggregation and dissemination frameworks for evaluating contextual queries in pervasive and ubiquitous systems [1], [19]. Once again, none of these solve the problem of optimally answering persistent contextual queries through a user-oriented programming interface in a single framework.

Recent times have also witnessed a number of activities in the specific domain of presence aggregation [16] – both in terms of standardization as well as research prototypes. [17] proposes script-based aggregation of presence documents from multiple sources that can be individually controlled for every subscribed watcher. An extension to the standardized presence information data format facilitates secure publication of watcher-specific views on the users presence status. The aggregation process can be controlled by user-specific rule-sets that specify how concurring presence notifications from the contributing sources have to be combined into a single presence information document. Geopriv [11] defines filters in XML documents which limit location notification to events which are of relevance to the subscriber. These filters persist until they are changed with a replacement filter. The valueChanges filter event contains a string which is interpreted as an XPath<sup>1</sup> expression evaluated within the context of the location-info element of a PIDF-LO [10] document which would be generated by the notification. For example, given a logical PIDF-LO document, If the state, county, city, or postal code changes, then a notification is sent. Further, RPIDS [3] rpids expands the basic set of presence states (e.g. active, on-line, off-line, on the phone, in a meeting, out to lunch, etc.) with states that are applicable to the broad consumer market (e.g. steering to denote the user is driving). Rich presence aggregates user information from multiple devices, networks and applications to provide a more comprehensive and accurate view of user status. For example, on the phone is an aggregate of all a users voice devices: desktop, mobile, remote office and Voice over Internet Protocol (VoIP) terminals. Applications such as IM and calendar also provide important user status information, such as in a meeting.

While presence aggregation plays a critical role in the vision of all-pervasive presence enabled applications, we take it a step further by describing a virtualization layer that is capable of distributing this (aggregated) information through a well-defined user interface to a plethora of presence-enabled applications. The underlying query processing techniques further make this a scalable solution that can support a large number of applications (queries).

Presence virtualization may also be viewed as a form of event stream processing, with virtualization queries represented as a graph of operators operating over PIDF-based incoming XML data streams. Several middleware platforms for applying operator graphs [21] or arbitrary processing code [20] over incoming sensor streams have been recently proposed. Our virtualization effort differs from this body of work in that it is tailored to consider several unique features of

<sup>1</sup><http://www.w3.org/TR/xpath>

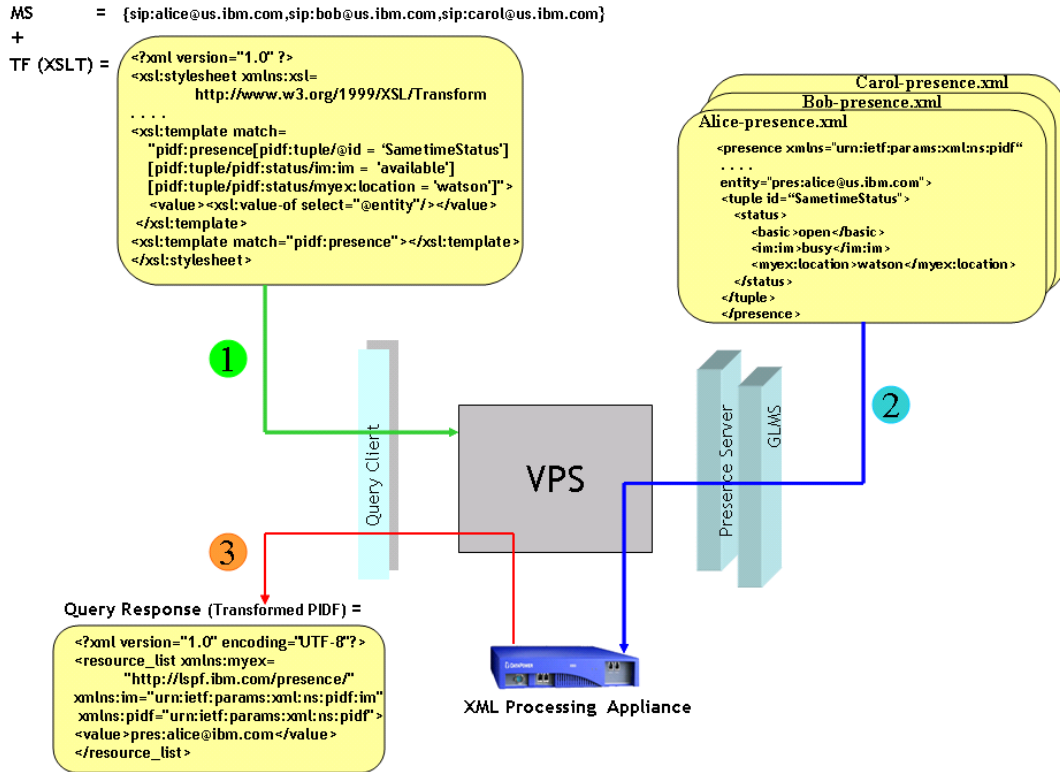


Fig. 2. XSLT-Based Query Specification for Presence Virtualization

the presence environment, such as the association of presence documents with specific URIs, the encapsulation of presence data in XML documents (hence, the use of XSLT operators) and the need to retrieve specific presence documents by subscribing to the PS (rather than assume that all presence events are proactively streamed to the VPS). Moreover, we focus explicitly on ensuring that the virtualization solution reuses the capabilities of the base SIP signaling protocol to the maximally possible extent, with a goal to minimizing changes to either existing presence clients or servers.

### III. BASIC DESIGN OF THE PRESENCE VIRTUALIZATION ARCHITECTURE

The goal of our presence virtualization work is to provide a semantically-useful abstraction over the underlying heterogeneous infrastructure; it is conceptually equivalent to the notion of “views” in database systems, which define custom abstractions over underlying physical tables. The conceptual presence virtualization framework is illustrated in Figure 1. The central element in the architecture is the *Virtualized Presence Server (VPS)*, which is responsible for accepting complex presence queries from clients and responding with the appropriate “virtual presentity” status. As shown in the figure, the virtualization layer consists of a set of VPS-es, which effectively shield the presence clients from the individual presentities managed by the underlying Presence Servers (PS). Each individual VPS may itself issue subscriptions to multiple underlying PS-es, potentially via the use of different presence

protocols. Each VPS supports multiple presence “queries”; to support efficiency, each VPS performs “query optimization” across the queries to essentially avoid redundant computations. To explain the details of our architecture, we must first explain our choices for two fundamental, and closely-coupled, aspects of virtualization:

- 1) How (i.e., in what structure and language) are the individual ‘virtualization queries’ expressed?
- 2) What is the basic unit of presence virtualization appropriate for the associated queries?

#### A. Choice of Virtualization Query Specification Format and Language

Each client wishing to avail of or instantiate a virtualized presentity on the VPS must specify its logic in a prescribed format that is both sufficiently expressive and permits efficient implementation in the VPS runtime. With the PS providing presence content in XML-based formats, it stands to reason that the manipulation logic will be based on one of the various XML manipulation languages (such as XSLT<sup>2</sup>).

To promote query expressiveness with efficient query reuse capabilities, we formulate each query as consisting of two distinct parts:

- A *Membership Set (MS)* part identifies the set of underlying presentities (either as an explicit list of individual pre-existing SIP URIs or via a group URI

<sup>2</sup><http://www.w3.org/TR/xslt>

corresponding to a resource list [6]) whose information is utilized to define different attributes of the virtualized presentity. In other words, the MS identifies the set of underlying presentities whose presence state is relevant to the posed query.

- A *Transformation Function* (TF) specifies a transformation (a sequence of operators) that is applied to the set of presence documents of the MS members to generate the response to the virtualization query.

Each virtualized presence query issued by a client is thus uniquely identified by the tuple (MS, TF). As an example, consider the virtualized query that seeks to return the subset of IBM buddies from (sip:alice@us.ibm.com, sip:bob@us.ibm.com, sip:carol@us.ibm.com) who are located in IBM Watson facility and available on IBM Sametime. In this case, the MS elements consists of the URIs {sip:alice@us.ibm.com, sip:bob@us.ibm.com, sip:carol@us.ibm.com}, while the TF consists of the logic that generates an XML attribute with the URIs of the buddies available on Sametime and in office. Clearly, multiple queries may be equal in either one or both elements of the tuple. As we shall see, this explicit separation of MS and TF components enables the VPS to efficiently exploit the commonality among MS elements of different queries.

In our VPS solution, the TF component of the query is specified as an XSLT transformation over the XML contents of the presence documents. We choose XSLT due to both its expressiveness, and the advanced vendor-specific XSLT support provided by a variety of XML processing appliances. To maintain consistency with the base SIP signaling infrastructure, the queries are carried as XML bodies in the Payload field of SIP SUBSCRIBE messages; these SUBSCRIBE messages are routed to the VPS, which is then responsible for extracting and processing the encased query. Figure 2 illustrates the details of an XSLT-based query (and the response) corresponding to the virtualized query discussed in the example above.

### B. The Query Processing Cell (QPC)

To implement a scalable virtualization platform that can simultaneously support a large number of virtualization queries, we introduce the notion of a *Query Processing Cell* (QPC) as the fundamental unit of presence virtualization. A QPC is a software object that effectively represents a virtual presentity (with a dynamically assigned URI) defined by a specific membership set (MS) such that its presence status is an *aggregation* of the presence data of individual members. Multiple queries with identical MS, but distinct TF, specifications are mapped to the same QPC. Each of the TF components of queries mapped to a single QPC are then viewed as subscriber-specific filters over this presence document. As illustrated in Figure 3, a VPS can then be viewed as a collection of QPCs, whose creation, termination and inter-QPC coordination are orchestrated by the *QPC Factory*.

In the most common interaction model, a presence client specifies a query (in the BODY of a SIP SUBSCRIBE mes-

sage) that is addressed to a well-known “VPS URI” (i.e. that of the QPC Factory) and thus routed by a standard SIP Routing Proxy to the VPS. The QPC Factory acts as a container for creating and managing multiple individual QPCs, each representing a “virtual presentity” created by the VPS. The QPC Factory is also responsible for redirecting virtualization queries to the appropriate QPC and for maintaining life-cycles of QPCs (e.g., performing clean up of a QPC when it no longer has any valid client subscriptions). During the initialization of a QPC, the QPC Factory sets up a dynamic resource list URI (containing all the URIs in the MS) on a Group List Management Server (GLMS). A QPC uses this GLMS URI to efficiently retrieve the raw presence data from the PS (rather than create per-URI subscriptions), a point further described in Section IV.

An additional component of the VPS is the *Query Catalogue*, which contains the repository for currently running virtualization queries. By exposing the contents of this catalogue through a Web-based interface, the VPS allows clients to reuse existing queries and QPC objects. To do so, the QPC Factory maintains a unique tuple, i.e. [MS, TF, QPC URI, TFid], for each query in the Query Catalogue. On receiving the  $i$ th incoming query, represented by  $(MS_i, TF_i)$ , the QPC Factory first inspects the entries in the query catalogue to determine if a virtual presentity (URI) exists for an *identically* matching MS<sup>3</sup>. If a match does not exist, the QPC Factory creates a new QPC object (instantiated with a newly specified virtual presentity URI) and installs a GLMS group list (with the virtual presentity URI), containing the individual URIs of the Membership Set. If, on the other hand, a matching QPC exists, the QPC Factory simply issues a SIP REDIRECT message to the client, asking it to reissue its SUBSCRIBE message to the existing virtual presentity (QPC) URI.

Further, to improve system scalability, each QPC offloads some of the query computation (involving manipulation of XML-based presence content) to an XML processing appliance. Whenever the computed result changes, each QPC uses SIP NOTIFYs to inform the end clients of a new response to their query.

Internally, each QPC consists of the following components (Fig. 4):

- A *Presence Fetcher* that interacts with the Presence Server to setup subscriptions on the underlying Presence Server and obtain the presence documents of each of the members of the MS.
- A *Controller* that takes the different TF requests from all clients mapped to the same QPC, and interfaces with the XML processing appliance (to be described in Section IV) to efficiently apply the XSLT transformations to the aggregated presence data of the MS (obtained by the Presence Fetcher).
- A *Query Receiver* that manages the external subscriptions issued by the virtualization query clients – this consists of

<sup>3</sup>Optimizations that permit better reuse of QPCs are specified in the discussions of ongoing work in Section V-A

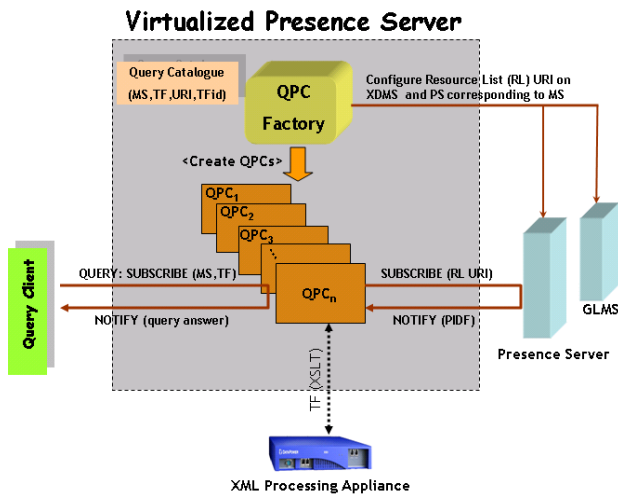


Fig. 3. Internal Architecture of a VPS, containing a single QPC Factory and multiple QPCs.

handling the SIP-based requests (SUBSCRIBEs) from the clients of this QPC, and for issuing NOTIFYs (containing the results of XSLT transforms) to the QPC's clients.

The operational details of each of these components will become clear in the next section, which describes the mechanisms by which a QPC interacts with the existing infrastructure to optimize its query processing.

#### IV. INTEGRATION OF VPS OPERATION WITH EXISTING ARCHITECTURE/PRODUCTS

Section III presented the fundamental design principles behind the VPS and its use of QPCs as units of computation. We now describe how the interactions between and with the various VPS components are designed to make optimal use of existing features and components of the SIP-based infrastructure that would be already deployed in an existing network. Figure 3 will be used to explain the component level interaction between the VPS, its individual QPCs and other functional components. In particular, our virtualization architecture uses the following three techniques to *a)* make efficient use of bulk subscriptions to the PS, *b)* maximize the reuse of query components among different clients and *c)* mitigate the processing overhead.

##### A. Interaction between QPC and Presence Server/GLMS:

GLMS is a component of the converged signaling network that provides the ability to efficiently store and managing resource lists (e.g. buddy lists). The resource lists are created, modified and deleted using the XML Configuration Access Protocol (XCAP).

To enable more efficient specification of subscriptions to the Presence Server (PS), the MS component of a query is configured as a resource-list in GLMS. The VPS (in specific, the QPC Factory) is responsible for interacting with the GLMS to associate a dynamic resource-list URI with the URIs of the presentities addressed by the query, and for interacting

with a Presence Server (using standard SUBSCRIBE-NOTIFY messages) to obtain the presence data for this set of presentities. The GLMS also supports an internal presence service, through which an external module can *subscribe* to changes in status of group lists (e.g. addition/deletion of elements to a list, deletion of list). The Presence Server (PS) exploits the above-mentioned functionality of GLMS to accept SIP-based subscriptions to resource-lists. In particular the PS, upon receiving a resource-list (MS) SUBSCRIBE message from a client, uses XCAP to retrieve the list of elements from GLMS and then subscribes to GLMS to be notified whenever there are modifications to the resource list. Internally, it subscribes to the presentities in the MS, gets notified of any changes in the presence documents of an MS member, and sends back any changes inside a NOTIFY message to the client (i.e. the VPS). The presence document inside the NOTIFY is an *aggregated* (PIDF) document containing the individual presence data of each MS member. The Presence Fetcher within the QPC subscribes to this resource-list URI to obtain aggregated presence information of MS, rather than maintain per-URI subscriptions.

##### B. Virtualization Query Routing to QPCs:

By appropriate use of standard SIP URI qualifiers and session redirection, the VPS allows different clients to interact with it in three different ways, without requiring *any* modifications to the client-side SIP stack.

Figure 4 (i.e. steps 1, 2, and 3 therein) shows the SIP-based interaction between a query client and the QPC (QPC Factory):

- A query client can issue its query (a SIP SUBSCRIBE with a (MS, TF) tuple in the body of the message) addressed to the QPC Factory URI. If a QPC corresponding to the MS exists, the client will be redirected to the QPC URI; else, a new QPC object will be created on-demand by the QPC Factory (with a dynamically allocated URI from the URI space managed by the QPC Factory), and the query client will be redirected to this new URI.
- The (MS,TF) query is then routed by the query router to the Query Receiver module of the QPC. To promote reuse, each TF being currently supported by the client is identified by a “query component” label (a “?id” suffix appended to the URI for the QPC). As before, if the TF exists, the query client is again redirected to the “sip:qpcURI?TFid” URI; else, the QPC Controller creates the corresponding TF transformation logic (on the XML processing Engine), generates a new “TFid” and then redirects the client to this URI.
- The (MS,TF) query addressed to a “sip:qpcURI?TFid” URI is then managed by the Query Receiver module of the QPC.

The Query Catalog entries expose the existing (MS, TF, qpcURI, TFid) bindings to the external world; accordingly, virtualization clients are able to reuse existing components on the VPS by directing their query to different URIs (e.g., if there is an existing query with identical MS and TF

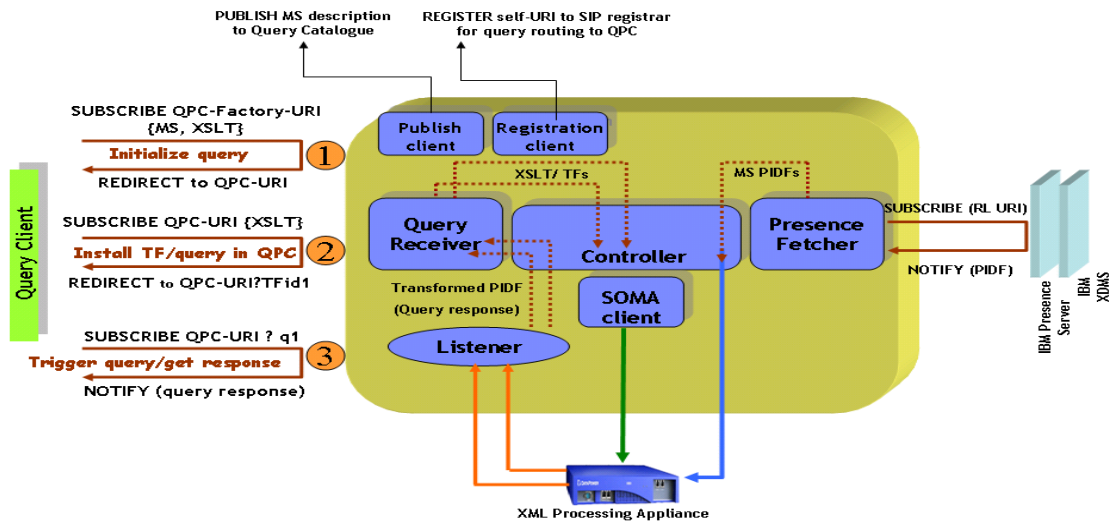


Fig. 4. Internals of QPC with the interactions between a query client and QPC/QPC Factory

components, the client can simply send its subscribe directly to the corresponding “sip:qpcURI?TFid” URI).

### C. Interaction between QPC and XML processing appliance:

XSLT-processing on a collection of XML documents can incur considerable processing overhead; to build a scalable XSLT-based presence virtualization platform, it is thus imperative to improve the execution of the queries.

To implement a high-performance virtualization solution, our VPS offloads the bulk of the XML transformation and processing logic to a “wire-speed” XML processing appliance (referred to as XML engine). The QPC interfaces to the XML processor through a Web-services based interface. Each QPC in effect installs a distinct firewall service on the XML engine; each firewall policy is identified by a specific (name, port) combination. To support multiple XSLT-based TFs emanating from query clients transforming a common MS, the QPC installs multiple TFs onto the firewall policy that can be applied in parallel (logically) on the arriving MS data. The QPC additionally spawns a HTTP listener for the response of each TF (XSLT) from the XML engine.

Recall that, the Presence Fetcher is initialized to receive an aggregated presence document as part of a NOTIFY, each time the presence information of any MS member changes. On receiving the NOTIFY, the QPC ships this merged XML document to the XML engine and receives a response (as a transformed PIDF document) from the firewall policy. The Query Receiver then transmits this transformed PIDF (corresponding to the output of the corresponding TF filter applied to the virtual presentity) via NOTIFY messages to the client. Figures 3 and 4 illustrate the specific interactions between the QPC and the XML processing appliance.

## V. VPS AND QPC IMPLEMENTATION DETAILS

We next discuss implementation details of the virtualization architecture. The current implementation of the Virtual Presence Server consists of a single QPC Factory with multiple

QPCs that are created, managed and destroyed on-demand. Each QPC is instantiated with (a) a resource-list URI that denotes the Membership Set (MS) of the virtual query, and (b) one (or more) Transformation Functions (TFs) to be applied to the presence documents of the MS members. The VPS is implemented in JAVA using IBM JAVA Version 5.0. Query clients interact with QPC Factory through the west-bound interface of the VPS, providing the Membership set (list of presentity URIs) and the Transformation function (TF) in the payload of a SIP SUBSCRIBE message. The QPC Factory redirects the client, using SIP RFC 3261 [7] semantics, to the URI of a newly created QPC or a *pre-existing* QPC.

Each QPC consists of a Query Receiver that is responsible for the subsequent interactions of the query client with the QPC. The Query Receiver is implemented using JAIN-SIP 1.2<sup>4</sup> and supports the requisite functionalities for redirection (RFC 3261 [7]) and subscription management (RFC 3265 [5]). Note that, a client subscribes to an installed transformation function by passing the identifier of the transform along with a SUBSCRIBE message. The QPC registers the client as a “watcher” on the TF. Subsequently, the QPC sends out any new response received from the XML engine to watchers subscribed to the corresponding TFid. The body of the NOTIFY message to the query client contains the new response.

The implemented VPS infrastructure consists of vendor-specific implementations of GLMS and Presence Server to operate with the QPCs. The Presence Fetcher in the QPC sits at the east-bound interface (implemented using JAIN-SIP 1.2) and subscribes to all presentities in the Membership Set. The Presence Server manages these subscriptions and sends an initial aggregated NOTIFY to the Presence Fetcher corresponding to the MSet. Subsequent NOTIFYs contain presence updates of individual members. The Presence Fetcher

<sup>4</sup><https://jain-sip.dev.java.net/>

parses the NOTIFY and extracts the presence document for each presentity. It then merges these documents into a well-formed XML document to be shipped to the XML appliance.

Each QPC has a south-bound interface that communicates with the firewall service installed at the XML appliance. Each firewall (one for each TF/XSLT) is responsible for handling XML transformations on incoming data and sending back the transformed responses to the appropriate HTTP listener of the QPC. These responses are then picked up by the Query Receiver that notifies the corresponding “watcher” clients.

Finally, all QPC objects (and their internal TF transformations) are maintained as “soft-state” in accordance with the base SIP protocols[5, 7]. This implies that each subscription has a specified duration and must be periodically refreshed. The QPC Factory and QPC objects maintain the timers to perform the necessary cleanup. In particular, a QPC object has the ability to self-destroy when the number of active subscriptions for its MS drops to zero; at this point, the QPC Factory removes the QPC, releases the virtual presentity URI and removes the corresponding entry created in the GLMS. Similarly, the QPC Factory and QPC coordinate to ensure that the Query Catalogue is always kept updated, as individual TF subscriptions for a specific QPC expire.

#### A. Ongoing Enhancements for scalability

We are currently investigating several possible architectural enhancements to the base VPS design detailed in this paper to support scalability for large-scale deployment of VPS. In particular, our ongoing work is focusing on the following two very interesting aspects of presence virtualization:

- *MS-Reuse and Hierarchical QPCs*: Currently, a new query  $q$ , denoted as  $(MS_q, TF_q)$  is matched to an existing QPC  $E$  (denoted by  $(MS_E, TF_E)$ ) only if the membership sets are identical, i.e.,  $MS_q = MS_E$ . For better reuse of MS-es (which will alleviate the subscription load on the underlying PS), other forms of full or partial matching may be used. For example, the query  $q$  can be mapped to the existing QPC  $E$  if its MS is *completely contained* in  $MS_E$ , i.e.,  $MS_q \subset MS_E$ . Of course, the  $TF_q$  has to be modified a bit (by the VPS) to essentially restrict the application of TF on only the appropriate subset of  $MS_E$  (i.e., to avoid the application of the  $TF$  operator on the presentities in  $MS_E$  that are not part of  $MS_q$ ). More interestingly, the QPC Factory can also exploit *partial membership* of QPCs; for example, it is possible that  $MS_C \subset \{MS_A \cup MS_B\}$ , but is not wholly contained in either. In this case, the QPC for  $MS_C$  may simply aggregate the data from the QPC objects for  $MS_A$  and  $MS_B$ , in effect creating a QPC *hierarchy*, without creating a new subscription on either the GLMS or the PS. We have developed an efficient maximal set-matching based heuristic to enable the dynamic creation and maintenance of such QPC hierarchies, and are currently evaluating its performance for realistic workloads.

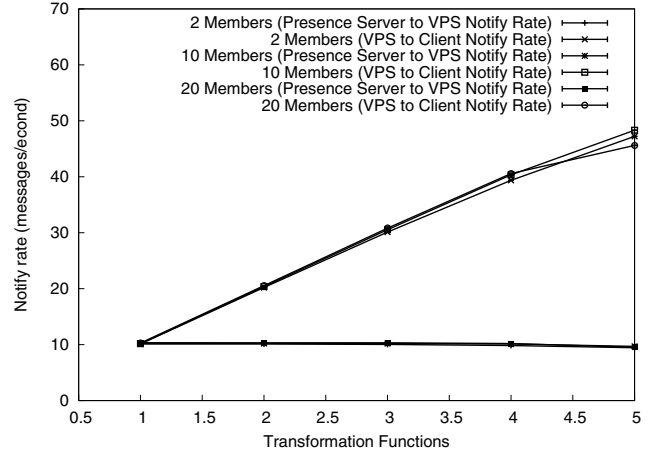


Fig. 5. Input and Output NOTIFY loads for the VPS

- *QPC coordination across multiple VPS-es*: The current QPC design focuses on scalable design of a single VPS. Our algorithms for creating and maintaining QPCs must be extended to consider the generic virtualization architecture, consisting of federated VPS-es. In particular, the query distribution algorithms may either partition (so that a particular  $MS$  subscription exists only on a single VPS) or replicate (different queries with the same  $MS$  are routed to different VPS-es) QPCs. We are currently developing such federated query routing algorithms that factor in the processing load of each individual VPS (implicitly preferring the use of a lightly-loaded VPS) and the possibility of hierarchical aggregation on a single VPS (implicitly preferring the use of a VPS where lower-layer QPCs may be locally exploited).

## VI. TESTBED PERFORMANCE RESULTS

In this section, we present the performance results observed with the small-scale deployment of our VPS implementation on a laboratory testbed. Our principle objective here is to demonstrate the functioning of our VPS implementation and gain some *initial* understanding of how VPS performance of a single server is affected by factors like the MS cardinality (the number of presentities in MS) and number of TFs (i.e., when more than one transformation logic is applied to the same MS). We would also like to demonstrate the use of the XML processing appliance to sustain the throughput of XSLT-based transformations. In addition to these limited ‘proof-of-concept’ tests, we are working to perform a medium-scale deployment of our solution in a tier-1 telecom service provider’s test environment—when completed, this deployment will enable us to better identify potential implementation bottlenecks and iteratively refine our implementation.

To demonstrate our ‘proof-of-concept’ implementation, we focus on two performance metrics:

- The *VPS NOTIFY throughput*, roughly defined as the number of NOTIFY messages (per second) generated and delivered to virtualization end clients.

- The *virtualization end-to-end latency*, defined as the interval between the PUBLISHing of a change in the presence status of a presentity and the corresponding delivery of the NOTIFY of the query response to the relevant virtualization clients. In our implementation, this delay will consist of several components (besides the network transport delay for each message), including i) the delay in the issuance of the NOTIFY from the PS, ii) the latency incurred by the QPC Presence Fetcher in redirecting the contents of an aggregated NOTIFY to the XML appliance, iii) the XSLT-processing latency in the XML appliance and iv) the latency incurred by the QPC Listener and Query Receiver in sending the transformed output in a NOTIFY to the virtualization clients.

We deployed a single VPS on a server with Intel(R) Xeon(TM) CPU 3.40GHz with 5GB of memory, running Red Hat Enterprise Linux AS release 4 and IBM Java 5.0. The PS was also deployment on a separated server with a similar configuration, but with enhanced system memory of 7GB. To simulate the creation of QPCs and subsequent installation of the TFs, we have implemented a watcher client that generates queries to execute the three-step subscription procedure with the VPS illustrated in Figure 4. The watcher client can be configured to create multiple QPCs and install multiple TFs within a QPC. In addition to the watcher client, we have also developed a publish load generator that randomly (with a specified frequency) changes the presence state of the presentities constituting the MSes. Both the watcher client and the publish load generator have been implemented with SIPp<sup>5</sup>, an Open Source light-weight traffic generator for SIP, and are deployed on different machines, but on the same local network as the VPS, PS and the XML processing appliance.

We present here the performance results obtained with three different tests:

- MS Cardinality: 2; Number of TFs: 1 – 5; Load: 10 publishes/sec
- MS Cardinality: 10; Number of TFs: 1 – 5; Load: 10 publishes/sec
- MS Cardinality: 20; Number of TFs: 1 – 5; Load: 10 publishes/sec

Note that the load is expressed as the number of PUBLISH messages generated per second by the publish load generator. Each presentity publishes two different types of dynamic presence information, viz. Yahoo! IM status and location. The publish load generator keeps on toggling between these presence attributes, resulted in cascaded responses from the relevant ‘downstream’ TFs. In the experiments presented here, each TF installed on a QPC is associated with a single unique watcher client.

Figure 5 presents, for varying TF and MS cardinality, (i) the rate at which NOTIFYs are received by the VPS (from the PS) and (ii) the rate at which NOTIFYs (corresponding to virtualization responses) are sent out by VPS to the clients. As expected, the throughput of VPS NOTIFYs linearly increases

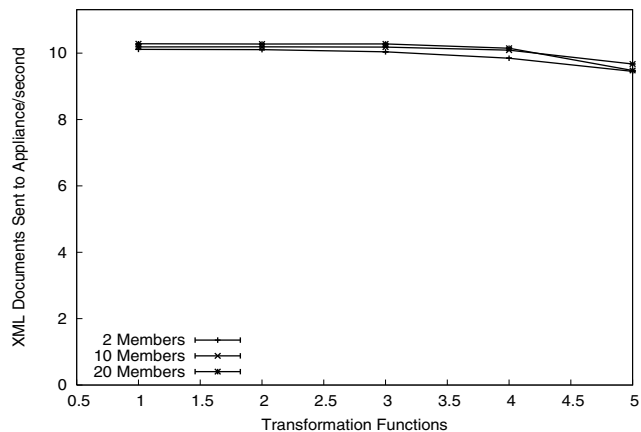


Fig. 7. Input Load on XML Processing Appliance

with number of TFs for a fixed value of MS cardinality and is a multiple of the input NOTIFY rate. Figure 6(a) and 6(b) present the CDF of the end-to-end latency for the cases where MS equals 2 and 5 respectively. We can see that the delay increases slightly with an increase in the number of members in MS; this is due to the potentially larger size of the XML-based PIDF that must be transported from the VPS to the PS, and then transformed by the XML appliance. In both cases, the end-to-end latency is more sensitive to the number of TFs per QPC. However, in general, the 95<sup>th</sup> percentile of the latency is lower than 1 second; since this includes various network-layer delays, we can infer that the architecture is capable of supporting presence virtualization at low-to-moderate loads with acceptable latencies. (For large notification loads, we shall have to employ the federated VPS architecture currently under investigation).

To further understand the performance of our implementation, Figure 7 plots the input XML transformation load (in terms of the number of XML documents input to the XSLT processor) observed on the XML processing appliance, while Figure 8 plots the corresponding XML output load (again, in terms of the number of documents generated as the output of XSL transforms). The two figures clearly demonstrate that the XML appliance is able to easily support the required transformation rates (of approx. 50/second), effectively helping to lower the processing bottleneck in our implementation.

## VII. CONCLUSIONS AND FUTURE WORK

Presence virtualization is a key enabler for the anticipated commercial deployment of next-generation, context-aware, converged applications. We present a novel middleware architecture for presence virtualization that allows applications to consume and compose real-time presence from various sources, specify their computation needs using XSLT-based transformations on the presence data, and compute responses to these queries using scalable XML processing technology. We are currently implementing several extensions to the base virtualization architecture, with a goal to further improve the virtualization efficiency and better exploit a federation of Virtualized Presence Servers.

<sup>5</sup><http://sipp.sourceforge.net>

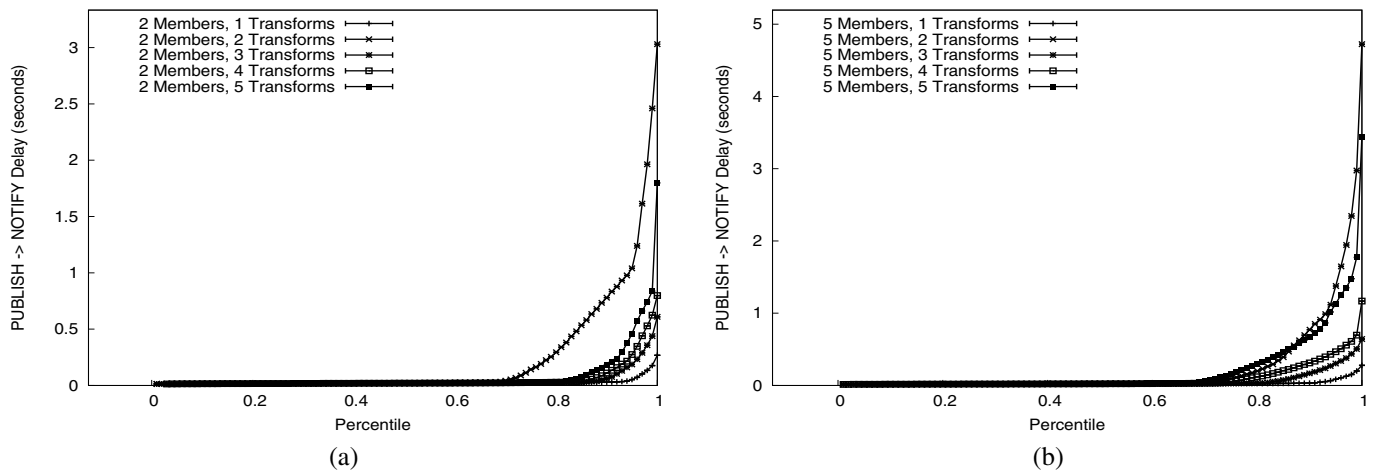


Fig. 6. (a)End-to-End Latency (MS = 2) (b) End-to-End Latency (MS = 5)

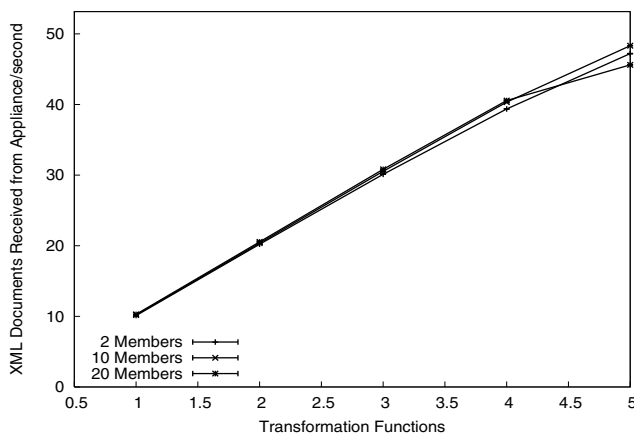


Fig. 8. Output Throughput of XML Processing Appliance

Future work in the area of presence virtualization must also address the challenges of security and privacy, that rely on the increasingly pervasive use of presence-based contextual data in multiple enterprise and provider domains. Also, to support virtualization requests from battery-constrained pervasive devices, we need to enhance the query specification primitives to allow specification of tradeoffs between the frequency/accuracy of virtualization responses and the overhead of virtual presenty notifications.

#### REFERENCES

[1] H. L. Truong, L. Juszczak, A. Manzoor and S. Dustdar, "ESCAPE - An Adaptive Framework for Managing and Providing Context Information in Emergency Situations", *Proceedings of EuroSSC*, pp: 207-222, 2007.  
 [2] H. Sugano, et al, "Presence Information Data Format (PIDF)", *In RFC 3863*, August 2004.  
 [3] H. Schulzrinne, "RPIDS Rich Presence Information Data Format for Presence Based on the Session Initiation Protocol (SIP)", *Internet-Draft - draft-schulzrinne-simple-rpids-02.ps*, Columbia U., February 2003.  
 [4] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", *In RFC 3921, IETF*, October 2004.  
 [5] A. Roach, "Session Initiation Protocol (SIP)-Specific Event Notification", *In RFC 3265, IETF*, June 2002.

[6] A. Roach, B. Campbell and J. Rosenberg, "A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists", *In RFC 4662, IETF*, August 2006.  
 [7] J. Rosenberg, et al, "SIP: Session Initiation Protocol", *In RFC 3261, IETF*, June 2002.  
 [8] J. Rosenberg, "A presence event package for the session initiation protocol (SIP)", *In RFC 3856, IETF*, August 2004.  
 [9] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, M. Valla, C. Fra, N. Paspallis, and G. A. Papadopoulos, "A Context Query Language for Pervasive Computing Environments", *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, 2008.  
 [10] J. Peterson, "A Presence-based GEOPRIV Location Object Format", *In RFC 4119*, December 2005.  
 [11] R. Mahy and B. Rosen, "A Document Format for Filtering and Reporting Location Notifications in the Presence Information Document Format Location Object (PIDF-LO)", *Internet-Draft - draft-ietf-geopriv-loc-filters-03*, Neustar, November 2008.  
 [12] H. Grine and S. Lecomte, "Self-adaptation of a query service using reconfigurable components", *Proceedings of IEEE 24th International Conference on Data Engineering Workshop (ICDEW)*, pp: 145-151, 2008.  
 [13] M. Fernandez, N. Onose, R. Hull and J. Simeon, "YooHoo! Building a Presence Service with XQuery and WSDL", *Proc. ACM SIGMOD International Conference on Management of Data*, June 2004.  
 [14] G. Chen and D. Kotz, "Context Aggregation and Dissemination in Ubiquitous Computing Systems", *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, 2002.  
 [15] N. H. Cohen, H. Lei, P. Castro, J. S. Davis II and A. Purakayastha, "Composing Pervasive Data Using iQL", *Proceedings of Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2002.  
 [16] O. Bergmann, "A framework for aggregation of presence information based on user provisioned rules", *Ph.D. Thesis*, 2007.  
 [17] O. Bergmann, J. Ott and D. Kutscher, "A Script-based Approach to Distributed Presence Aggregation", *Proceedings of the IEEE WirelessCom Conference*, USA, June 2005.  
 [18] E. Belinsky, et al, "PASTA: Deriving Rich Presence for Converged Telecommunications Network Applications", *Proc. IEEE COMSWARE*, January 2007.  
 [19] J.E. Bardram, "The JAVA context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications," *Proceedings of PERVASIVE 2005, LNCS Springer*, Heidelberg, 2005.  
 [20] L. Amini, et al, "SPC: A Distributed, Scalable Platform for Data Mining", *SIGKDD 2006 Workshop on Data Mining Standards, Services, and Platforms*, August 2006.  
 [21] D. Abadi, et al, "Aurora: A New Model and Architecture for Data Stream Management", *VLDB Journal*, vol. 2, no. 2, pp. 120-139, August 2003.