RI09008, 27 April 2009 Computer Science

IBM Research Report

A Scalable Middleware for Presence Virtualization and Federation

Arup Acharya, Xiping Wang
IBM Research,
T. J. Watson Research Center,
Hawthorne, NY, USA.

Nilanjan Banerjee, Dipanjan Chakraborty, Koustuv Dasgupta, Shachi Sharma
IBM Research,
India Research Lab,
New Delhi, India.

Abstract. Presence is a key enabling technology for developing rich context-aware applications. However, there is no general purpose presence infrastructure available today which is both flexible and highly scalable to cater to a wide range of end users (consumers) and applications. We have introduced earlier [12], the concept of *virtual presentities* (logical presentities created in response to applications or user queries to presence systems) and a middleware for the life-cycle management of the virtual presentities, that leverages the presence infrastructure in a programmatically flexible and scalable fashion. Given the exponential rise in the number of presence sources and the rapid emergence of sophisticated context-aware applications, the scalability of the middleware is critical to its efficiency and general acceptability. In this paper, we present the design to improve scalability further. A scalability measure, inherent to the system design, is the representation of the presence logic in applications and those used by end users as persistent queries on presence data, or the virtual presentities, for potential re-use among multiple applications and end users with identical requirements. This reduces the effective load on the base presence servers and thereby enabling it to support more queries. Another scalability measure is to enhance the life cycle management of the virtual presentities with the introduction of hierarchical data re-utilization strategy. Besides enhancing the scalability of the virtualization middleware, we have expanded its range of operation by federating multiple presence domains and thereby enabling applications and users making use of presence data from different heterogeneous, but independent presence domains. In this paper, we present the detailed design, implementation and performance evaluation of the enhanced middleware, highlighting the different measures adopted for scalability and present experimental results to corroborate them.
Keywords: Presence, context, virtualization, scalability, federation

1 Introduction

Presence technology was originally developed for communicating "online status" in instant messaging applications. Later, it graduated to become a key enabler of Web-based *content provider* (e.g., Google Talk^{TM}, Yahoo! Messenger ^{TM}or Skype^{TM}), *enterprise* (e.g., IBM Sametime^{TM}) 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). Thus, presence may be broadly described as a publish-subscribe system for context, that 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.*

1.1 Presence Technology: A Background

Current presence solutions are largely based on SIMPLE [4, 7] extensions to the base SIP signaling protocol (with Google Talk being a notable exception that utilizes the XMPP [3] protocol). The SIP-based presence model [4] defines a presentity, which is a combination of device, services, and personal information that depicts a user's presence status in the network and is identified by a single presentity SIP URI, as follows. This presence information is presented in the Presence Information Data Format (PIDF) [1], which has been defined as the reference format for all IETF presence implementation. Each PIDF document has presence data entries consisting of one or multiple tuples that contain a basic status, each identified by one or more attributes, and optionally an address and other data. PIDF, being based on XML is very extensible and allows the trivial addition of namespaces to the basic structure. Thus a wide range of PIDF extensions have been defined to model the context information of diverse entities [2].
The basic presence architecture consists of an application server called the Presence Server (**PS**) that 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" 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 the SIP URI of the presentities. 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 [5]) are considered. Moreover, the subscription logic over the content of individual URIs is restricted to a limited set of pre-defined "filter" operators specified in SIP standards [7] (e.g., alerts only on specified changes in the location value).

The problem: With the further emergence of advanced presence-based services, individual subscriptions or simple filter based logic is insufficient for the development of rich presence-driven context-aware applications. Consider the following two scenarios:

– Bob wants to hang out with a bunch of friends in the evening at a pub and wants to know which bar has the maximum number of the his friends who are his Instant Messenger (IM) buddies on that evening.
– Alice, an application developer is developing a smart call center application, which routes calls based on expertise and availability of the employees in an enterprise. For example, a call to Java Helpdesk gets automatically routed to one of the available Java experts located at a particular location.

The current practice of developing such application is to embed the application logic in vertical, independent application silos, where it's usually the developer's (or end user's) responsibility manually subscribe to the

relevant presence information available in various servers through domain specific gateways and execute the necessary logic on them as appropriate for the concerned application. For example, in the first scenario, Bob has to set appropriate filters on his IM buddies' and and then manually collate them to figure out which one of the bars has the maximum number of his buddies. Obviously, this is a tedious and time-consuming job. In the second scenario also, Alice needs to do the same thing in pulling the relevant data from different presence servers and embed the logic of finding online Java experts within her application and then integrate it with the call center application. Thus the queries are built within application silos vertically making it very difficult for other applications or user to use them directly or indirectly. We believe that this approach of developing presence-based application would present serious limitations to the deployment of a large-scale, scalable presence infrastructure for future converged applications. The limitations are characterized by the following requirements:
Scalability: With the advances in sensing, computation and communication, it is envisioned that there will be several orders of magnitude increase in the volume of presence data available. For example, there have been ongoing efforts [8] in using sensors on cell phones to infer variety of activities or events, both personal and social. Given that, presently there are almost 3 billion cell phones around the world, it is a very challenging task to manage and use this data efficiently in real-time. Hence, the presence infrastructure should be scalable enough to control both the network traffic (in terms of presence updates and notifications) and the server processing (in terms of both subscriptions and application based presence logic) loads. This scalability is critical for real-life scenarios such as 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).
Flexibility: The presence infrastructure should provide a flexible programmable interface to support a wide variety of presence queries required for the development of advanced presence based applications.
Federation: Furthermore, 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). Thus, future converged applications require the presence status from multiple sources/domains and we call the paradigm enabling such applications, *presence federation*. The current practice to solve this problem is to build domain specific gateways for presence federation.

1.2 Presence Virtualization and Federation

To mitigate the above issues, and effectively support more sophisticated use of presence, we have developed a *presence virtualization* solution [12] that provides a *programmable* abstraction by which applications can "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. Moreover, our virtualization solution also allows clients to *expose and share* the end results of their queries with other relevant clients; in effect, virtualization allows presence consumers to capture the persistent state of external queries as *virtual presentities* (presentities created in response to the queries), which become a seamless part of the presence infrastructure and are functionally indistinguishable from the 'raw' presentities.
By the virtue of the virtualization middleware Bob can add the particular query, described earlier, as a presence buddy in his IM client and can get the answer to the query at the click of a button whenever necessary. This approach has an additional benefit of application of user defined policies on the virtual presentities in the same way as they are applied to the regular presentities.
In case of the more generic queries such as the one in the second scenario, a catalogue of all the existing queries can be maintained and a developer, such as Alice, can manually inspect the catalogue to check whether the query of her interest exists already. If it exists then there is no need to create a new query and Alice can simply subscribe to the query to get the appropriate response. In case the query does not exist in the catalogue, the developer can find out which of the existing queries can partially meet her requirements along with raw presence information from other presence servers and can create a new query in the system. Of course there is this possibility that none of the existing queries is of any use to Alice and in that case all the required presence data needs to be pulled in from the presence servers. But, we envision that with more and more developers using this system an organically created interconnected set of queries would develop and a community oriented view of presence-based services will gradually emerge, which will enable query re-utilization at a large scale.
A detailed account of the design and implementation of this virtualization middleware has been reported in [12]. We will recapitulate the salient design points later in Section 3. The earlier work, however, realized the programmable virtualization middleware without much consideration into the scalability and federation aspects, which as mentioned earlier, are critical to the performance and wider acceptibility of the system. In this paper, we focus on these aspects and enhance the design for improved scalability and augment federation capability to the virtualization middleware to realize a highly scalable and flexible virtualization and federation middleware.

Key Contributions: The primary contribution of this paper is to take the concept of presence virtualization from our previous paper [12] and develop a middleware design that makes presence virtualization scalable.

we support scalability in three significant ways: (i) We had earlier developed 'query processing cell' (QPC) as a logical entity to couple related presence virtualized queries (transformation functions) and associated presentities. In this paper, we developed a comprehensive design to dynamically and automatically create a multiple of such QPCs based on end-user query loads/patterns. This ensures that the overall processing power of the system scales up with increasing load. (ii) We designed an hierarchical model of interconnecting QPCs so that presentity status available in one QPC is made available for subscription to other queries (that may be executed in a separate QPC). This reduces the subscription load on the PS by allowing subsribed data to be re-used across multiple QPC. Besides data re-use, our design allows re-use of tranformation functions across QPCs - when a new TF is instantiated, it can not only subscribe to presentity documents/state in other QPCs, but also to the output of other tranformation functions. (iii) A system-level evaluation of QPC implementation comparing multi-threaded design with single threaded design is carried out. Further, extensive experiments on the multiple QPCs with and without data reusability establishes the fact that increase in the level of data reusability in the QPCs design leads to enhanced performance of the system. We then extend the virtualization model to allow virtualization queries to be posed not only on the presentity data on the local presence server, but also on presentities that exist in a different presence system. We design and develop a specific model of presence federation for this purpose whereby an enterprise appears as a user on a consumer-facing presence system such as gtalk, so that presence of enetreprise clients can be pulled into a virtual presence system running within the enterprise.
The rest of the paper is organised as follows. Section 2 motivates the design considerations for scalability and federation. Section 3 recapitulates the presence virtualization system and presents the salient design enhancements done to the middleware for higher scalability and presence federation. Implementation of the system and the experimental results to validate the design considerations are presented in section 4. Section 5 presents a survey of related works in this area. Section 6 concludes the paper.

2 Motivation and Design Principles

In this section, we present the motivations behind the scalability design and federation capability of the middleware.
Enhancing Scalability: Let the set of presentities in a PS is denoted by P and $|P| = N$. Also, let there be M applications. So, there can be $|\mathcal{P}(P)| = 2^N$ unique combinations of the presentities that the applications can use, where $\mathcal{P}(P)$ is represents the power set of P. Now assuming that each application can uniformly use a group presentity selected from $\mathcal{P}(P)$, we can say that $M/2^N$ applications subscribe to each member of $\mathcal{P}(P)$ on an average. Now in $\mathcal{P}(P)$ each of the presentities in P can occur for atmost $2^{(N-1)}$ times. Hence, the total number of notifies sent out by the PS for each presentity in P is given by $(M \times 2^{(N-1)})/2^N = M/2$.

Now with virtualization middleware, each of the elements in $\mathcal{P}(P)$ can be reused by $M/2$ applications. Hence, the load on PS is automatically reduced by a factor of $M/2$. With the reuse of the presence data among the virtual presentities using a hierarchical organization, the load on PS can be reduced further so that the total number of notifies sent out by the PS is equal to $|P|$ or in other words there is at most one subscription per presentity in the PS.

Thus, a key design consideration for an efficient middleware is to push the application logic as close to the PS as possible (not necessarily to the PS), rather than pulling data into the application context. At the same time, it is impractical to assume that a PS would take this responsibility. The intermediate virtualization middleware does the trade-off and brings the query and data together. Users' queries are intercepted by this layer and responded to, rather than by the PS. Flip side is that this increases the computational load on the middleware and it requires a highly scalable design in terms of interdependent but independent computational units bolstered by additonal hardware appliances dedicated for specialized operations. Of course, we can add more server machines to scale this middleware as the query load increases.

In the following, we present the design principles that we adopted in our solution for increased scalability of the middleware. The first two corresponds to the middleware's scalable use of the base PS, whose performance is obviously critical to the overall scalability of the system and the third one corresponds to the computational scalability of the middleware.

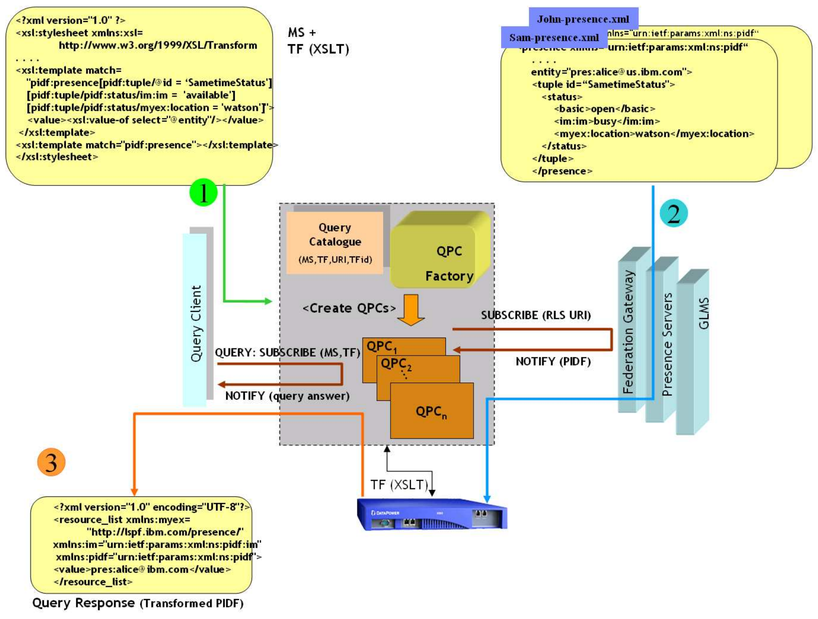
Re-use of Virtual Presentities: Virtual presentities do not directly reflect physical entities like people, devices etc., but they represent a computation on state of other physical presentities or recursively on other virtual presentities, and the computed state represents one execution of a query on the presentities, *e.g., 'which bar has most of my friends'*. This computed state is sent to all subscribers of this query. If any input to the query changes, e.g., a new friend enters a bar, the query re-executes and the updated answer is sent to all the subscribers. A virtual presentity can therefore be viewed as a continuously running or *persistent query*. Unlike physical presentities the virtual ones are unsuitable for stable storage. This is key to scaling since a virtual presentity's state change could be frequent and if they are represented as a physical presentity, their state change would need to be reflected as an update to a corresponding presence/XMLdocument, which would not scale. Also, in case of a failure, the virtual presentity can be re-initialized by re-installing the query on the relevant component presentites. By creating a interconnected network of virtual presentities we are, in effect, creating an event propagation system, where the raw events trigger the next level virtual presentities to be recomputed which then trigger further recomputation on higher-level presentities - thus, the effect of an event change is propagated only as far as it needs to. If we enable caching of query execution, then an update no longer propagates beyond a presentity which did not get changed due to query re-execution. The virtual presentities are in-memory presistent queries whuch can be shared across multiple consumers. As indicated by the PS load calculations, this has a direct implication to the PS load

as the number of subscriptions to base PS reduces with the increase in sharing of virtual presentities.
Presence Data Re-use among Virtual Presentities: Again as shown in the above calculations significant reduction of subscription load on base PS can be obtained when the virtual presentities re-use the presence data obtained from the PS among themselves. Thus, a virtual presentity can procure necessary data by internally subscribing to an exisiting virtual presentity that has already obtained the data from the PS. This directly reduces the number of subscriptions to the base PS and adds to the scalability of the system. The internal data re-usage can be further improved by creating a hierarchy of virtual presentities with the higher layer virtual presentities re-using the presence data of the lower layer virtual presentities through internal subscriptions.
Hardware Offloading: One of the most computationally demanding operation of the virtualization middleware is to evaluate the queries from XML-based presence documents. There are several general purpose hardware appliances available for processing XQuery or XSLT based queries on XML documents. When loaded with an XML document and a corresponding query, these appliances return the result of the query almost at wire speed, which is a few order of magnitude faster than any of the available software XML query processing engine. We have made use of one such hardware appliances to offload the large number of query computations that we envision for our middleware, thereby increasing the scalability of the overall system.
Presence Federation: Physical presentities may be stored in several different domains and the virtualization middleware needs to interface with these domains in order to execute a query that involves presentities from heterogeneous domains. This requires *presence federation* among multiple heterogeneous domains. Presence federation is achieved today either by building isolated domain specific gateways or embedding the federation logic within application logic. These approaches are clearly not flexible enough to support the application paradigm described before. Thus, there is clearly a need of a general purpose gateway, integrated with the virtualization middleware, that can federate multiple presence domains and seamlessly interface with them, so that virtualization could be oblivious to the heterogeneity of the underlying presence infrastructure.

3 Design

In this section, we review our basic design of the virtualization middleware followed by a detailed account of the design for enhanced scalability and presence federation.

Presence Virtualization Architecture - A Recap: We developed our system on the basic presence framework defined in SIMPLE [4] and SIP as the common message passing protocol between different components of our system. Although, any generic publish-subscribe system could have been used for our purpose, SIP and SIMPLE provided the following ready advantages: a) SIP is widely available at the communication end-points today, be it on cell phones or on desktops through different IM/voip clients. SIP has also been accepted as the standard signaling protocol for IMS, which is going to the key component in the next generation converged network systems; b) SIMPLE is already used to publish raw events. Therefore it makes sense to re-use the same interaction model to express queries in our system c) SIP has the flexibility to carry XML based payloads thus making it ideal for expressing general purpose queries.



```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl=
          http://www.w3.org/1999/XSL/Transform
. . . .
<xsl:template match=
   "pidf:presence[pidf:tuple/@id = 'SametimeStatus']
   [pidf:tuple/pidf:status/im:im =  'available']
   [pidf:tuple/pidf:status/myex:location = 'watson']">
   <value><xsl:value-of select="@entity"/></value>
 </xsl:template>
<xsl:template match="pidf:presence"></xsl:template>
</xsl:stylesheet>
```

```
nlns="urn:ietf:params:xml:ns:pidf"
urn:ietf:params:xml:ns:pidf"
    . . . .
    entity="pres:alice@us.ibm.com">
    <tuple id="SametimeStatus">
      <status>
         <basic>open</basic>
         <im:im>busy</im:im>
         <myex:location>watson</myex:location>
      </status>
    </tuple>
    </presence>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<resource_list xmlns:myex=
          "http://lspf.ibm.com/presence/"
xmlns:im="urn:ietf:params:xml:ns:pidf:im"
 xmlns:pidf="urn:ietf:params:xml:ns:pidf">
<value>pres:alice@ibm.com</value>
</resource_list>
```

Fig. 1. VPS Architecture

A Virtualized Presence Server (VPS) embodies the virtualization and federation middleware, which is responsible for accepting complex presence queries from clients and responding with the appropriate virtual presentity status. The basic design of the VPS is presented in [12]. Here we would review the salient design concepts and then describe the various scalability measures adopted to improve the overall performance. The two fundamental, and closely-coupled, aspects of virtualization are (i) the expression (i.e., in what structure and language) of the individual virtualization queries and (ii) the basic computational unit of presence virtualization.

Since the presence documents are expressed in XML (PIDF), the manipulation logic is expressed in the queries using an XML manipulation languages *viz.* XSLT. To promote query expressiveness with efficient query reuse capabilities, each query is composed 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 corresponding to a resource list [5]) whose presence state is utilized to define different attributes of the virtualized presentity. For example, in case of the second scenario described in Section 2, the MS will consist of the indexes (SIP URI) to all the presentities corresponding to all the Java experts in the enterprise.
- A *Transformation Function* (TF) specifies a transformation (a sequence of operators) expressed in XSLT that is applied to the set of PIDF documents of the MS members to generate the response to the virtualization query. Thus, the query posed by the call center application will have an XSLT which will return the list of all the available Java experts at a particular location.

Each virtualized presence query issued by a client is thus uniquely identified by the tuple (MS, TF). Figure 1 illustrates the details of an XSLT–based query (and the response) corresponding to the virtualized query discussed in the example above.

To implement a scalable virtualization platform that can simultaneously support a large number of virtualization queries, we have developed 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 1, a VPS can then be viewed as a collection of QPCs, whose creation, termination and inter-QPC coordination are orchestrated by the *QPC Factory*.

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 2 (i.e. steps 1, 2, an 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

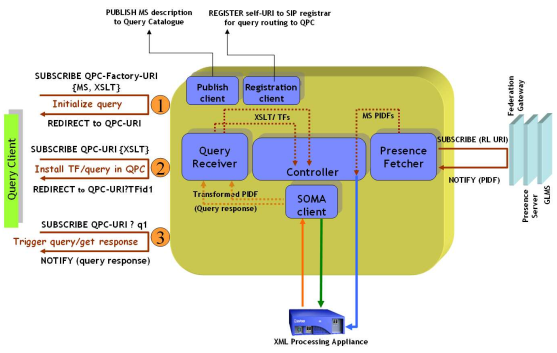


Fig. 2. Internals of QPC with the interactions between a query client and QPC/QPC Factory

"sip:qpcURI?TFid" URI; else, the QPC Controller installs 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 components, the client can simply send its subscribe directly to the corresponding "sip:qpcURI?TFid" URI). Whenever the computed result of a query changes, each QPC uses SIP NOTIFYs to inform the end clients of a new response to their query.

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).

Internally, each QPC consists of the following components (Fig. 2):

– 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 efficiently apply the XSLT transformations to the aggregated presence data of the MS.
– A *Query Receiver* that manages the external subscriptions issued by the virtualization query clients – this consists of 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.

3.1 Multi-QPC Scenario and Data Re-use

QPCs are the basic computational unit in VPS design. Each QPC has its own MS on which it operates. In this subsection, we discuss the scenarios when and how these basic units can be re-used when there are multiple QPC running in the VPS. There are three possible cases:

1. If an incoming new query has all members of its MS same as that of existing QPC's MS i.e. the incoming MS is equivalent to an existing MS.
2. If an incoming new query has all members of its MS in an existing QPC's MS i.e. the incoming MS is subset of existing MS.
3. If an incoming new query has all or some members of its MS present in more than one QPCs i.e. the incoming MS has some members present in existing QPCs but it is neither a equivalent set nor a subset of existing MS.

QPCFactory on receiving a new request determines through catalogue lookup if there exists a QPC having MS equals to (case 1 above) or superset of incoming MS (case 2 above). If it finds such a QPC then QPCFactory redirects client to existing QPC. This design also facilitates to re-use an installed the TF inside QPC. On receiving a request, QPC checks if the same TF is already installed into it or not. If the TF exists in QPC, it simply redirects client to TF URI. Hence, the QPC design permits clients to re-use both the data and transformation logic and thus helping in providing scalability into the VPS.
The QPC also supports identity transformation. If a client sends a request with identity transformation, the QPC returns the aggregated presence document of all members of its MS. This feature of QPC design is useful for supporting case 3 which we discuss in next sub section.

3.2 Hierarchical Model of QPCs

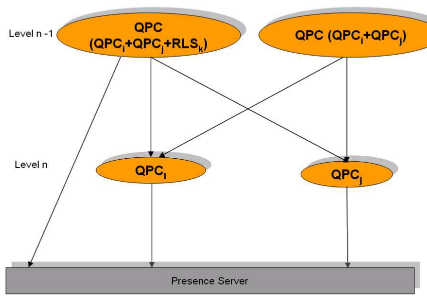


Fig. 3. QPC hierarchy

With multiple QPCs using common presence data, it is possible for the QPCs to internally share the presence data by building a hierarchy of QPCs in the VPS. Let there be N presentities publishing to

PS. Let $\{P = P_1, P_2, \ldots P_N\}$ be the set of all presentities. At any point of time, there are M QPCs running inside VPS such that $Q = \{QPC_1, QPC_2, \ldots QPC_M\}$ is the set of all running QPCs and for each $QPC_i \in Q$, there exists a $MS_i \subseteq P$ and a transformation function TF_i. The members of Q are organized in a hierarchy so that a QPC either gathers its presence data from one or more QPCs or from PS directly or from both QPCs and PS. This is depicted in Figure 3 where QPC at level $n-1$ fetches presence information of its members from QPC at level n and from PS. Let a new query comes to VPS with membership set MS_0 and transformation function TF_0. The problem at hand then is to place the corresponding new QPC, *viz.* QPC_0 in the hierarchy H in the VPS. As shown on Table 1, there are 4 possibilities. The first row is all about reusing the existing QPCs while for the second row QPCFactory determines the list of membership sets from which MS_0 can be composed from. Let QPC_n have membership set $MS_n = \{P_1, P_2, \ldots P_K\}$ and some members of $\hat{MS_0}$ belongs to MS_n, where $\hat{MS_0} \subseteq MS_0$, then QPC_0 sends SUBSCRIBE to QPC_n for $\hat{MS_0}$ where $\hat{MS_0} = \{P_1, P_2, \ldots P_{K1}\}, K1 < K$, with an identity transformation function and QPC_n sends back aggregated presence document composed of the members of $\hat{MS_0}$ in NOTIFY message to QPC_0.

Table 1. Cases for QPC and TF re-utilization

	$TF_0 = TF_n$	$TF_0 \neq TF_n$
$MS_0 = MS_n$	QPC re-utlization	QPC re-utlization with TF installation
$MS_0 \neq MS_n$	Creation of QPC_0 and installation of TF	Creation of QPC_0 and installation of TF

Creating Optimal Hierarchies The VPS generates two types of load: (i)Load on presence server: Each QPC inside VPS subscribes to presence server (one or more) for presentities in its membership set (either through separate URI for each presentity or through group list URI). This generates SUBSCRIBE-NOTIFY load between VPS and presence server. (ii) Load on QPC: In hierarchical design of QPCs, a QPC may subscribe to one or more other QPCs for presence data of all or some members of its membership set. QPC load comprises of this internal subscribe-notify load and external notifys QPC receives from presence server and sends out to its clients.

The load on presence server and the load on QPCs due to internal subscriptions are complementary to each other. The load on presence server is more expensive as it corresponds to signaling load in the SIP network. The main consideration in building optimal hierarchy is to balance these two loads.

Greedy Set Cover Algorithm (GSC): The objective of greedy algorithm is to use minimal number of QPC for internal subscription, thus subscribing to QPCs with maximally matching MS, which in turn boils down to the problem of finding minimal cardinality set cover over a given set. GSC, however, results in skewed subscription load distribution within H, where a few QPC handles the majority of internal subscriptions. To distribute the load efficiently, we propose a weighted set cover approach as follows.

Weighted Set Cover Algorithm (WSC): We define weight or flow of a query processing cell as sum of its inflow and outflow, i.e., $flow = Inflow + Outflow$, where the inflow of a QPC is defined as the total number of NOTIFYs it receives for all its presentities (i.e. its membership set members) and the outflow of a QPC is sum of number of NOTIFYs it sends to other QPCs. Thus, $Inflow = \sum_{i=1}^{n} U_i$, where n is the cardinality of MS of QPC and U_i is the updates for presentity P_i and $Outflow = \sum_{i=1}^{n} S_i U_i$, where S_i is the count of SUBSCRIBEs QPC receives for a presentity P_i and U_i is the updates for presentity P_i. The algorithm is depicted in Figure 1.

```
**Input**: $Q, MS_0$
**Output**: $C, MS_0^{out}$
$C = \phi$; $MS_0 = \phi$;
Sort $Q$ in ascending order of weight of each element;
while $Q \neq \phi or MS_0 \neq \phi$ do
    Remove first element $Q_i$ of $Q$;
    Extract $MS_i$ of $Q_i$;
    if $MS_0$ contains some elements of $MS_i$ then
        Add $Q_i$ to $C$;
        $MS_0 = MS_0 \ MS_i$;
    end
end
if $Q = \phi$ and $MS_0 \neq \phi$ then
    $MS_0^{out} = MS_0$;
end
```

Algorithm 1: The Weighted Set Cover Algorithm

The new QPC_0 sends SUBSCRIBE to all members of C generated by WSC and to PS for members of MS_0^{out}. The QPC_0 is inserted into hierarchy H.

Table 2. Standard deviation of load on QPCs

Number of Requests	50	100	1000	10000
GSC	2435.864117	2766.630256	12230.03768	34805.36056
WSC	1378.266869	1878.277492	7529.176063	24713.21174

Extensive simulation experiments have shown that the WSC performs better than GSC in terms of load distribution among QPCs with increasing number of queries submitted to the system. Table 3.4 shows the standard deviation of the load across all QPCs with increasing number of query requests for the two algorithms, where load on a QPC is measured in terms of the total flow through it. The results are for Pareto publish rate and when the MS for each QPC is selected from a universe of presentities following Pareto distribution. The reason for choosing Pareto distribution is based on the fact that some of the presentities are much more familiar and hence will be used more frequently compared to others. As the results show the difference in standard deviation of QPC load gets amplified with the increase in the number of query requests. Figure 3.4 illustrates this fact for 10000 query requests and for Pareto scale parameter = 1.5.

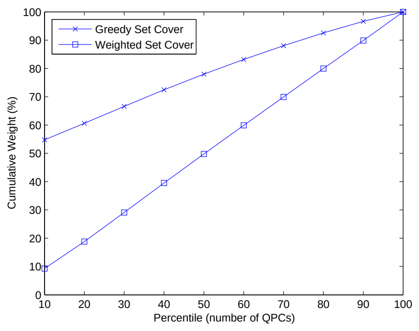


Fig. 4. Simulation Results: Distribution of QPC Load for Hierarchical Algorithms

3.3 Interaction between QPC and XML processing appliance:

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. The XML processing appliance works in loopback mode. In the lookback model, a new firewall service is created for every new XSLT for XML transform processing and the transform result is directly sent back to the client through an HTTP response.
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 the 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 2 illustrate the specific interactions between the QPC and the XML processing appliance.

3.4 Presence Federation

Presence federation is mostly an engineering problem which often reduces to an administrative one. Different types of federation models have been

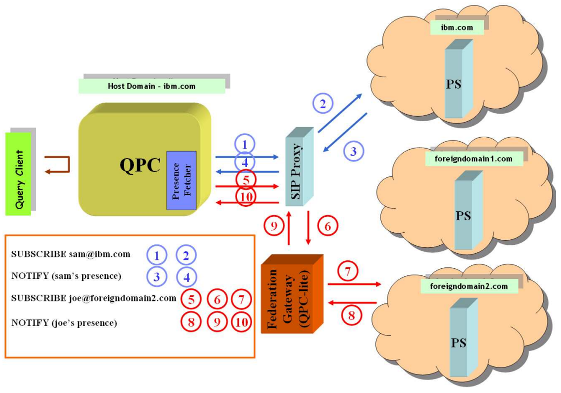

ibm.com
PS
Host Domain - ibm.com
QPC
Presence Fetcher
Query Client
SIP Proxy
foreigndomain1.com
PS
foreigndomain2.com
PS
Federation Gateway (QPC-lite)
SUBSCRIBE sam@ibm.com 1 2
NOTIFY (sam's presence) 3 4
SUBSCRIBE joe@foreigndomain2.com 5 6 7
NOTIFY (joe's presence) 8 9 10

Fig. 5. Presence Federation

proposed depending on the constituent domains. There could be server-to-server federation or there could be domain specific gateways between two heterogeneous domains, exchanging presence information. The former model is subject to a lot of issues such as business relationship, administrative policies, protocol heterogeneity, etc., while the latter, although easier to implement has limited applicability. Here we propose a general purpose model where an application specific federation gateway is integrated with the virtualization middleware, thereby increasing the scope of the applications developed over the virtualization and federation middleware.

In our model, the federation gateway creates a proxy account in the foreign domain and adds the presentities of interest in that domain as trusted members. This enables the gateway to procure the presence information of the presentities and seamlessly make them available to the virtualization and federation middleware. Some commercial IM service providers, such as Google^{TM}, provides APIs to programmatically subscribe to the presence status of a buddy. We leverage this feature to implement the gateway that federates the VPS domain and the Google^{TM}domain. Thus the federation gateway creates an user account with Google^{TM}and adds the users whose presence status is relevant to the VPS. Of course, the user also needs to add the federation gateway user as a buddy. This model is very useful for developing applications such as the one for a courier company who wants to deliver package only after ensuring customers availability at home. The application can add the customers as buddies in any commercially available IM application and check on their availability through the virtualization and federation middleware. The customers are also motivated to add the application as their buddy simply because they can expect better service from the courier service provider.

The mode of interaction between the VPS and the federation gateway is the same as it is between two QPCs in the hierarchical QPC model. The gateway essentially implements a lighter version of QPC (QPC-lite), which on the front-end accepts SUBSCRIBE and sends NOTIFYs like the Query Receiver and in the back-end interfaces with the foreign domain in a domain specific way. The proposed model of presence federation is depicted in Figure 5. A SUBSCRIBE to a foreign domain from the Presence Fetcher of a QPC is re-directed to the federation gateway, which then handles the seamless translation of SUBSCRIBE and NOTIFYs between the host and the foreign domain.

4 Implementation and Results

The first version of VPS implementation was described in [12]. Since then, several enhancements were done to the system for improvement in overall performance including scalability. The VPS is still implemented with IBM Java Version 5.0, but all the SIP based interaction have been implemented using the product grade SIP Stack instead of the open source JAIN SIP. A major change in the implementation of QPC has been the development of all internal components as independent threads. This enables more efficient handling of the asynchronous events that comes to the VPS for processing and thereby improves the overall performance of VPS. The interaction of VPS with the XML processing appliance is also changed to support the loopback mode of the appliance with an objective of leveraging maximum performance benefit from the hardware appliance. The rest of the components like GLMS, PS are vendor specific implementations of open standards.
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 query client that generates queries to execute the three-step subscription procedure with the VPS illustrated in Figure 2. For test purpose a single query is used, which yields the list of currently available buddies at a particular location. The query client can be configured to create multiple QPCs and install multiple TFs within a QPC. The query client forms MS by selecting presentities from a universe of presentities following Pareto distribution. Also, it is possible for the query client to include buddies from different domains such as `gmail.com` in the MS of the query through the middleware. The federation gateway in the middleware for interfacing with Google^{TM}is implemented using Smack APIs[1], an Open Source XMPP (Jabber) client library in Java for instant messaging and presence. In addition to the query 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 query client and the publish load generator have been implemented using IBM SIP Stack and are deployed on different machines, but on the same local network as the VPS, PS and the XML processing appliance.

[1] Smack API - http://www.igniterealtime.org/projects/smack/index.jsp

Table 3. Comparison of Presence Server Subscription Load

Number of Requests	Without data Reuse	With partial data reuse	With Hierarchy
10	67	58	18
50	288	188	34
100	576	357	45

4.1 Experimental Results

In this section, we present several experimental results to substantiate the design decisions taken to improve the scalability of the system. For all the experiments the load offered to the system 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. The queries submitted to the system are also generated at an uniform rate. The results are categorized into four stages of development of the system as follows.

Single-threaded QPC: Figure 6 (a) and (b) present the CDF of the VPS latency for the $|MS| = 5, 40$. The number of TF is increased from 1 to 10. The publish rate or load to the system is 10 publications/second and a total of 3000 publishes are sent. For equivalent values of the critical parameters such as $|MS|$ and load, the VPS delay is found to be much lower than what was reported in [12]. Figure 6(c) shows the rate at which the NOTIFYs are received at by the VPS (input rate) and the rate at which the NOTIFYs (corresponding to virtualization responses) are sent out by the VPS to the query clients (output rate). We observed significant performance improvement in terms of throughput over the results reported in [12]. In this case, we could sustain much higher values for MS (upto 20) and TF (upto 10). For even higher values of $MS(= 40)$ the throughput starts dropping.

Multi-threaded QPC: We carried out the same experiments as above with the multi-threaded implementation of the QPC and the corresponding results are shown in Figure 6 (c), (d) and (e). Here we observed similar performance for throughput, but significant improvement in terms of VPS latency, particularly for higher $|MS|(40)$ and number of $TF(10)$ values.

Multiple QPC Experiments: The next set of experiments was done with multiple QPC scenario, which forms the basis of data and query re-utilization within VPS, which is a key factor for overall scalability. These experiments were carried out with the following parameters: cardinality of the presentity universe was fixed to 100; publish rate was 10 publications/sec; total number of 3000 publishes were sent; the query generation rate was 12 requests/min and the Pareto scale parameter for selecting presentities in MS in query client was fixed to 1.5. The three experimental scenarios were: (i) Without data re-utlization or set matching between MSets – QPCFactory simply creates a new QPC for every incoming re-

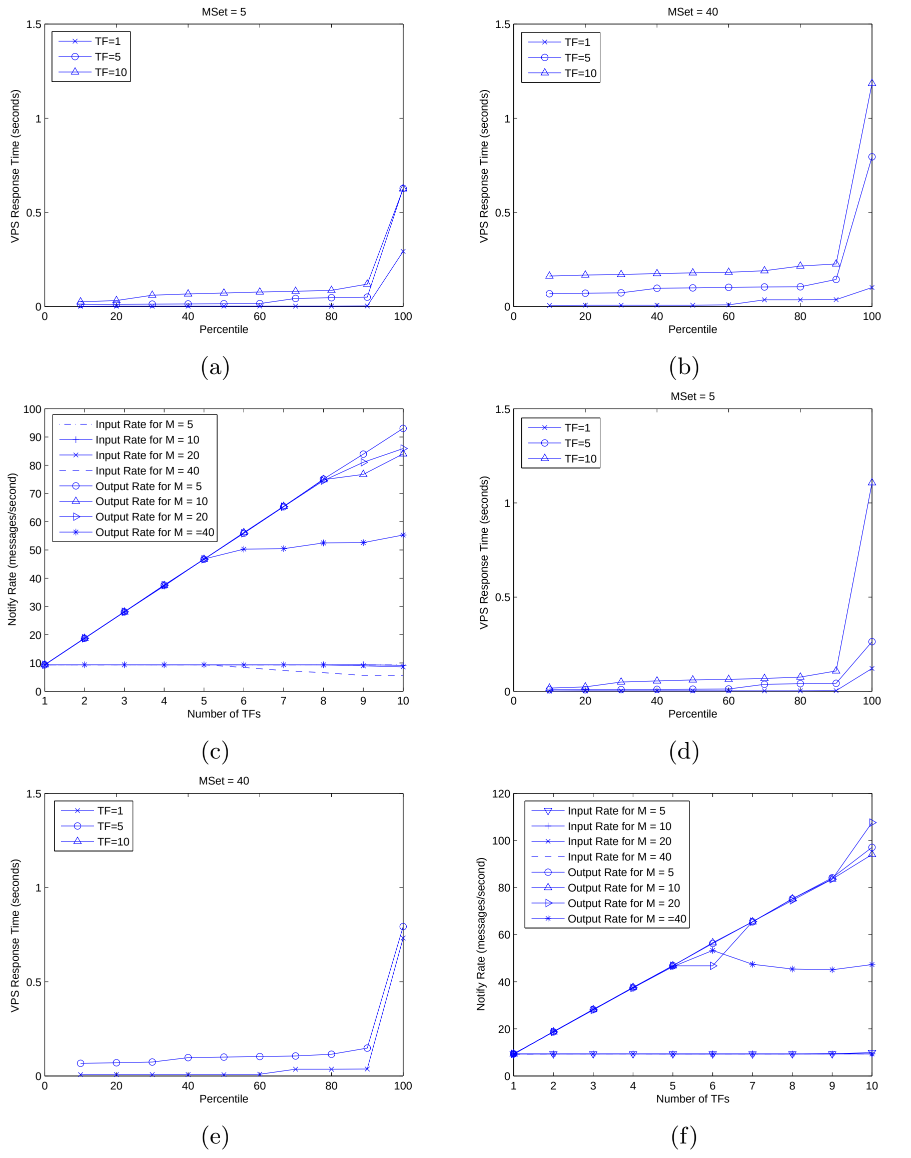


Fig. 6. Single threaded QPC: [VPS Latency for (a)($|MS| = 5$) (b) ($|MS| = 40$) and (c) Throughput]; [Multithreaded QPC: VPS Latency for (c)($|MS| = 5$) (d) ($|MS| = 40$) and (e) Throughput];

quest; (ii) With partial data re-utilization – QPCFactory on receipt of a request checks whether the incoming MS or its superset exists in the

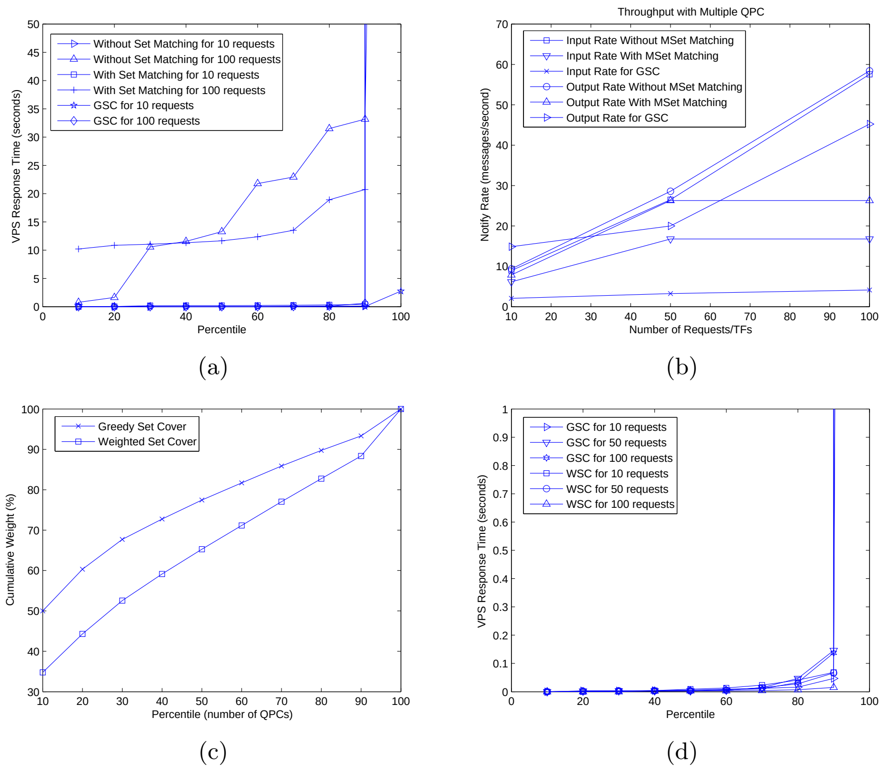


Fig. 7. Multi QPC: [(a) VPS Latency (b) Throughput]; (c) Experimental Results: Distribution of QPC Load for Hierarchical Algorithms; (d) Experimental Results: VPS Latency with GSC and WSC;

system and if QPCFactory finds such a QPC, it redirects the client to that existing QPC; (iii) With full data re-utilization – QPCFactory on receiving a new request runs set cover algorithm to find the collection of QPCs which contains all or some member presentities of the incoming MS. In this case, we have chosen GSC algorithm as described in Section 3.2 to build QPC hierarchy with internal subscriptions between the QPCs.
Figure 7 (a) shows the processing delay in VPS in aforementioned cases. The delay is found to be consistently decrease with the increase in data re-utilization. Thus the delay was observed to be the lowest for GSC and maximum for the case without data re-utlization. This is because data re-utilization saves the PS access time for each QPC resulting in an overall
Figure 7 (b) compares the notify rate in the above mentioned three cases. The most interesting observation from the results is that the ratio of output to input rate increase drastically with the increase in data

Table 4. Comparison of GSC and WSC

Algorithm	Number of Requests	Number of QPCs Created	Hierarchy Height	Presentities Used	QPC-QPC Edges Edges
GSC	10	8	3	18	15
WSC	10	8	4	18	15
GSC	50	24	4	34	56
WSC	50	24	6	34	47
GSC	100	45	4	45	119
WSC	100	45	7	45	144

re-utilization. This is because, with the increase in data re-utilization, the input notification rate decreases and at the same time, because of ready data availability larger number of queries gets evaluated resulting in higher output rate.
Table 3 shows the subscription load on PS in the above three cases. As expected, the maximum load on the PS is when there is no data re-utilization. Maximum re-utilization of data with hierarchy corresponds to the minimum number of subscriptions to the PS.

Hierarchy Experiments: We compare the performance of GSC and WSC algorithms with respect to the following performance metrices: (i) distribution of load on QPCs inside VPS, and (ii) delay in processing Notifys (which is a function of publish rate of presentity and height of the hierarchy). All these measurements are taken with following set of parameters: presentity universe consists of 100 presentities; publish rate is constant to 2 publications/sec; total number of publishes 2000 publishes are sent; query generation rate is fixed to 12 requests/min and total of 100 requests are submitted to the VPS. In these experiments, both the publish load generator and query client ran simultaneously. This is an essential requirement for WSC algorithm as it is dependent on the load on QPCs which depends on number of notifys handled by it.
The distribution of load on QPCs for GSC and WSC is shown in Figure 7(c). With GSC, 10% of QPCs are contributes to almost 50% of the VPS load, whereas with WSC, 10% of QPCs contribute to approximately 35% of the VPS load. This implies WSC results in more uniform distribution of load when compared to GSC. However, WSC may result in longer hierarchy as opposed to GSC (refer to Table 4). Despite this, as shown in Figure 7(d) the VPS processing delay was found to be almost similar for the two algorithms. Thus, WSC is preferrable when a better distribution of load is required among the

5 Related Work

It may be tempting at times to find paralles of presence virtualization in the areas of context-aware queries for pervasive and ubiquitous systems, presence aggregation systems or event processing systems. However, as discussed earlier [12], when compared with the existing body of work

(references in [12]), presence virtualization and federation middleware presents several differentials from the above mentioned existing research work. None of these works provides a comprehensive, scalable middleware framework for answering persistent contextual queries over presence data through a flexible programming interface. Individual comparisons with the existing works in these areas can be found in [12].
Recently there have been some work on "Invisible or Deep Web" [13] where the key problem has been identified in the processing of dynamic user-generated queries on large volume of real-time information such as stock quotes, weather information, flight information etc. Even the very advanced search engines available today for the regular Web are incapable of answering such queries primarily because the web crawlers are usually designed to handle static or semi-static web pages and are not equipped to handle very dynamic real-time information. There are currently technolgies available to subscribe to real-time information in the Web (e.g., RSS), but they do not allow subscription to user-generated queries. Our virtualization and federation layer essentially solves these problems and thus can provide a middleware based solution for enabling user-specified scalable searching in the Invisible Web.
Several models for presence federation have been proposed. The models can be categorized into the following two broad categories *viz.* (i) Single protocol system and (ii) Multi-protocol system. Each of these two categories can be further classified into (i) intra-domain federation and (ii) inter-domain federation. Intra-domain federation with single protocol is more of an administrative problem than anything else [10]. Inter-domain federation with single protocol has been studied mostly for XMPP and various flavors of server-to-server federation have been proposed [11]. However, the server-to-server federation works for pair-wise relations between the servers and the connections between them also are established on an one-to-one basis. In the multi-protocol scenario, a protocol translation function in addition to the server-to-server federation functionality is required.

6 Conclusions

In this paper we presented the scalability design the presence virtualization middleware proposed earlier and validated the design with extensive experimental results. We also added presence federation capability to the middleware, expanding its scope of operations across heterogenous presence domains. Some of issues that we would like to explore in the future are as follows. Privacy and security of sensitive presence information is very critical to the success of any presence based application and we would like to integrate these functionalities with the virtualization and federation middleware. Fault tolerence of the middleware is another issue of our research interest. The base PS is another bottleneck in the system and we are weighing the possibility of scaling PS through a farm of PSs. Similarly, scaling the hardware offloading functionality in our architecture with multiple federated hardware appliances is also in our consideration. Finally, we want to explore the possibility of smarter presence updates from the source for overall scalability of the middleware.

References

1. H. Sugano, et al, “Presence Information Data Format (PIDF)”, *In RFC 3863*, August 2004.
2. 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.
3. P. Saint-Andre, “Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence”, *In RFC 3921, IETF*, October 2004.
4. A. Roach, “Session Initiation Protocol (SIP)-Specific Event Notification”,*In RFC 3265, IETF,* June 2002.
5. A. Roach, B. Campbell and J. Rosenberg, “A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists”, *In RFC 4662, IETF*, August 2006.
6. J. Rosenberg, et al, “SIP: Session Initiation Protocol”, *In RFC 3261, IETF*, June 2002.
7. J. Rosenberg, “A presence event package for the session initiation protocol (SIP)”, *In RFC 3856, IETF*, August 2004.
8. A. T. Campbell, et. al, ”The Rise of People-Centric Sensing”, *IEEE Internet Computing*, Page(s) 12-21, 2008.
9. J. Rosenberg, “A Data Model for Presence,” *In RFC 4479*, July 2006.
10. J. Rosenberg, et. al., “Models for Intra-Domain Presence and Instant Messaging (IM) Bridging”, *draft-ietf-simple-intradomain-federation-02*, November, 2008.
11. “The XMPP Cloud: Building a Presence-Enabled Infrastructure for Real-Time Communication,” *Jabber White Paper*, Jabber Inc.
12. A. Acharya, et. al. “Programmable Presence Virtualization for Next-Generation Context-Based Applications”, Percom, March 2009.
13. M. K. Bergman, “The deep web: Surfacing hidden value.” *Technical report*, BrightPlanet LLC, Dec. 2000.