A Semantic P2P Framework for Building Context-aware Applications in Multiple Smart Spaces

Tao Gu [a] , Hung Keng Pung [b] , Daqing Zhang [a]

[a] Institute for Infocomm Research, 21 Heng Mui Keng Terrace, Singapore

[b] National University of Singapore, 3 Science Drive 2, Singapore
tgu@i2r.a-star.edu.sg; punghk@comp.nus.edu.sg; daqing@i2r.a-star.edu.sg

Abstract. Context information has emerged as an important resource to enable autonomy and flexibility of ubiquitous applications. The widespread use of context information necessitates an efficient lookup service in a wide-area network over multiple smart spaces. In this paper, we propose a context lookup framework based on a semantic peer-to-peer network to support the building of context-aware applications in multiple smart spaces. Collaborative context-aware applications that utilize different context information in multiple smart spaces can be easily built by invoking a pull or push service provided by our framework. We describe the design of our system, demonstrate the development process of context-aware applications, and report the measurements obtained from our prototype.

Keywords: Context lookup, semantic peer-to-peer network, context-aware applications, multiple smart spaces.

1 Introduction

The recent convergence of ubiquitous computing and context-aware computing has seen a considerable rise in interest in various context-aware applications. These applications exploit various aspects of the contextual environment to offer services, present information, tailor application behavior and trigger adaptation, based to the changing context.

Context information gathered from various sensor systems is the basis for enmeshing ubiquitous computing into our daily lives and exhibiting the autonomy of applications. Storing and acquiring such information in a single smart space can be easily handled by a centralized database server. The server can provide fast response to a lookup query. However, handling large-scale context information over multiple smart spaces requires an appropriate context lookup architecture. Distributing database servers to multiple smart spaces in a wide-area network does provide a scalable and reliable solution. However, this approach requires a significant investment on servers, the bandwidth costs of storing and updating context information, and administration restrictions.

Emerging Peer-to-Peer (P2P) approaches have been proposed to overcome some of these obstacles, and providing potential solutions for a large-scale distributed lookup system. This paper proposes a semantic P2P framework to support storing and acquiring context information in multiple smart spaces. In this framework, context data is stored in a context producer where it was generated. Each context producer is only responsible for managing its local context data that may be acquired from the sensors attached. For an efficient context lookup, we design and implement a semantic P2P overlay network in which context data is organized and retrieved according to their semantics. In this network, context producers are arranged in such a way that those with semantically similar data are grouped together so that a context query can be routed efficiently. Many existing or potential collaborative context-aware applications can use our framework, especially those collaborative context-aware applications over multiple smart spaces. For examples, Family Intercom [5] – an advanced communication application between multiple smart homes, is able to identify the caller and the recipient and mediate the initiation of the audio conversation. In health care applications, a tele-monitoring application tracks a patient wherever he/she goes or a tele-medical record application can provide anywhere availability of personal medical history.

The rest of the paper is organized as follows. We describe the system architecture and its details in Section 2. We then present collaborative context-aware applications in Section 3, and report the results obtained from our prototype in Section 4. We discuss related work in Section 5. Finally, we conclude the work in Section 6.

2 System Architecture

2.1 Overview

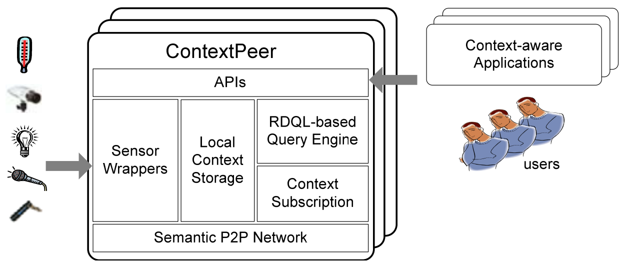

Fig. 1. The architecture of ContextPeers

Our framework consists of many individual nodes called ContextPeers, which act as context producers. Users and context-aware applications act as context consumers to obtain context data by submitting their queries to ContextPeers and receive the results. ContextPeers are self-organized into a semantic P2P network [6] for supporting P2P search. ContextPeers exploits semantic P2P overlay as the underlying network layer and extends it with RDF-based context storage, context queries and context subscription.

As shown in Fig. 1, each ContextPeer consists of five components: the semantic P2P network layer, the sensor wrappers, the local context storage, the RDQL [10] based query engine and the context subscription. The sensor wrappers capture various sensor data from physical or virtual sensors, and convert raw (i.e., direct sensor output) format into an RDF-based data model (i.e., in the form of RDF triples) and store the triples into the local context storage. The RDQL-based query engine parses and resolves context queries from users or applications. The context subscription registers subscription requests and notifies context consumers when context changes occur. The semantic network layer is responsible for network construction and maintenance, and query routing. Application developers utilize a set of APIs provided by our framework to access the functionalities of ContextPeers and build context-aware applications. The class diagram containing the major classes in a ContextPeer is shown in Fig. 2.

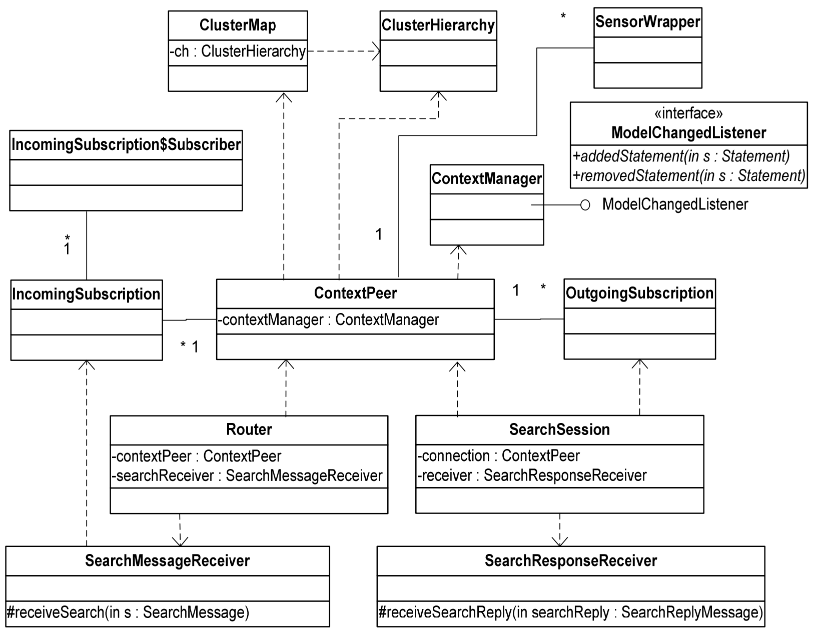
ClusterMap
-ch : ClusterHierarchy
ClusterHierarchy
SensorWrapper
*
«interface»
ModelChangedListener
+addedStatement(in s : Statement)
+removedStatement(in s : Statement)
IncomingSubscription$Subscriber
ContextManager
ModelChangedListener
*
1
1
IncomingSubscription
ContextPeer
-contextManager : ContextManager
1
*
OutgoingSubscription
* 1
Router
-contextPeer : ContextPeer
-searchReceiver : SearchMessageReceiver
SearchSession
-connection : ContextPeer
-receiver : SearchResponseReceiver
SearchMessageReceiver
#receiveSearch(in s : SearchMessage)
SearchResponseReceiver
#receiveSearchReply(in searchReply : SearchReplyMessage)

Fig. 2. Class diagram of a ContextPeer

2.2 Data Model

In our system, we use an RDF-based context model to represent context data. RDF provides a universal platform for representing resources and asserting relations between resources in a machine-readable and machine-understandable way. Each RDF statement is represented as a triple of the form <*subject*, *predicate*, *object*>. We adopt a hierarchical context ontology model defined in [6] which consists of a shared upper ontology and a set of domain-specific ontologies. The upper ontology defines common concepts, and it is shared among all ContextPeers. Each ContextPeer can define its own concepts in its low-layer ontologies which extend the leaf concepts in the upper ontology. Different ContextPeers may store different sets of low-layer ontologies based on their applications' needs. This design approach offers application developers the flexibility to define domain knowledge which is specific to their applications.

2.3 Sensor Wrapper

A ContextPeer acquires context information from the various sensors attached to. We create an appropriate wrapper for each type of sensors. This approach can avoid explicit binding of the application to a particular underlying context sources technology. Wrappers capture sensor data, and convert them into RDF-based context data. A ContextPeer selects a set of wrappers based on its contextual interests, subscribes to the wrappers and gets updated.

We use the *SensorWrapper* class to construct a wrapper. A *SensorWrapper* object instance is associated with a set of *ContextTriple* objects specifying the provided context and an *UpdateHandler* object implementing actions for context update. For example, an RFID-based location sensor wrapper is able to convert sensor data <*`RoomID RFID-John`*>, where *`RoomID`* represents the ID of a bedroom and *`RFID-John`* represents the RFID sensor attached to a person – John, to the RDF statement <*`John locatedIn Bedroom`*>, representing that John is currently located in the bedroom.

2.4 Local Context Storage

Each ContextPeer maintains a local repository for storing context data. The repository stores context data ontologies, static context data and sensed context data. Static context data refers to context data that does not change frequently, such as the spatial information of buildings (e.g., John's bedroom is located in John's house). Sensed context data refers to data obtained from sensors (e.g., John is located in his bedroom). Such data is typically dynamic and changes frequently.

The *ContextManager* class built on Jena’s *Model*s [8] is responsible for managing the repository. It provides methods to add or remove context data and answer context query, and also provides a set of operations to combine ontologies or context data. Since sensed context data changes frequently, the *ContextManager* class attaches a *ModelChangedListener* to the sensed context data *Model* to monitor these changes. This is especially important for responding to incoming subscribed queries.

2.5 Data Mapping

Upon creation, a ContextPeer needs to decide which cluster to join in the semantic P2P overlay network. This is done by using an ontology-based semantic mapping technique to map a ContextPeer’s local data to semantic cluster(s) as defined in the context ontology, and count the number of triples corresponding to each semantic cluster. This technique traces the hierarchy of OWL [7] classes and maps their predicates to the associated classes. We create two structures – *ClusterHierarchy* and *ClusterMap*. We first map each triple to an OWL class using *ClusterMap*, and then map the triple to an appropriate semantic cluster using *ClusterHierarchy*. Let SCn_{sub} , SCn_{pred} , SCn_{obj} where $n = 1, 2, \ldots$ denote the semantic clusters extracted from the subject, predicate and object of a triple respectively (Note: unknown subjects/objects or variables are mapped to all). If the predicate of a triple is of type *ObjectProperty*, we obtain the semantic clusters using $(SC1_{pred} \cup SC2_{pred} \cup \ldots SCn_{pred}) \cap (SC1_{obj} \cup SC2_{obj} \cup \ldots SCn_{obj})$. If the predicate of a triple is of type *DatatypeProperty*, we obtain the semantic clusters using $(SC1_{sub} \cup SC2_{sub} \cup \ldots SCn_{sub}) \cap (SC1_{pred} \cup SC2_{pred} \cup \ldots SCn_{pred})$. The semantic cluster with the highest triple counts (called the major semantic cluster) is selected for the ContextPeer to join. To achieve this, each ContextPeer creates and maintains a *HashMap* and iterates through all the triples in its model. Upon successful execution, the method returns a vector containing all the semantic cluster IDs corresponding to all its local context data. The first element in this vector indicates the ID of the major semantic cluster.

2.6 Query Routing

We follow the principle of a small world network model [9] and extend it with clustering operations to build the semantic network. After obtaining the semantics of its local data, nodes are organized in such a way that those have semantically similar data are grouped together in a semantic cluster. As a node's data may correspond to multiple semantic clusters, a node joins its major semantic cluster and publishes the indices of its data (i.e., reference pointer) to its minor semantic clusters.

Routing Table Construction: Each node builds its routing table by creating a set of local contacts in its own cluster, a short-range contact in each of its neighboring clusters, and a small number of randomly chosen long-range contacts. Each newly joining node builds its routing table in the same way resulting in all the clusters being linked linearly in a ring fashion. As illustrated in Fig. 3, *Node 1* builds two local contacts (*Node 2* and 3) in *SC1*, two short-range contacts (*Node 4* and 5) in *SC0* and *SC2* respectively, a long-range contacts (*Node 6*) in *SC5*, and publishes its indices to a random node (*Node 7)* in *SC3*.

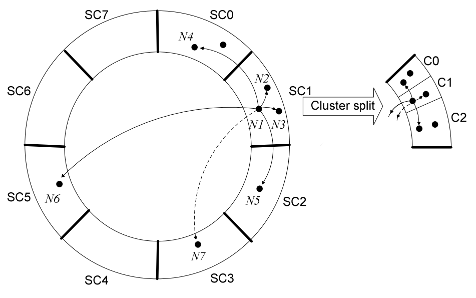

Fig. 3. Constrcution of the Semantic Network

Grouping context providers with similar data according to their major semantic clusters has the effect of minimizing the cost of node joining, leaving, and data changes. A node will stay in its major semantic cluster as long as the majority of data does not change. However, a large number of nodes in a semantic cluster may result in a scalability issue. We design the follow clustering operations to enable the network to scale to a large number of nodes.

Clustering Operations: When the number of nodes in a semantic cluster exceeds a certain size, cluster splitting occurs. Let *M* represent the maximum cluster size. If the size of a cluster exceeds *M*, the cluster is split into two. Each node maintains a *CurrentLoad* which measures its current load in terms of the number of triples and data indices it stores. When node *x* joins the network, it sends a join request message to an existing node, say *y*. If *y* falls into the same semantic cluster that *x* wishes to join, *x* joins the cluster by connecting to *y* if its cluster size is below *M*; otherwise *y* will direct the request to a node, say *z*, in the semantic cluster that *x* wishes to join, and *x* will connect to *z* if its cluster size does not exceed *M*. If the cluster size exceeds *M*, node *y* or *z* (called an initial node) will initiate the splitting process. The initial node first obtains a list of all the nodes in this cluster which is sorted according to their *CurrentLoads*. Then it assigns these nodes in the list to the two sub-clusters alternatively. After splitting, we obtain two clusters with relatively equal load. The initial node is also responsible for generating a new cluster *ID* for each of the two sub-clusters. To obtain a new cluster ID, each node maintains a *bit split pointer* which indicates the next bit to be split in the *n*-bit binary string. Initially, the *bit split pointer* points to the most significant bit of the *n*-bit string. When cluster splitting occurs, the bit pointed by the *bit split pointer* is split into 0 and 1 and move the pointer forwards to the next bit in the *n*-bit string. The same mechanism follows for the insertion of a new semantic cluster. A semantic cluster can be split into a maximum number of 2^n clusters. After splitting, a node updates its *cluster ID*, the *bit split pointer* as well as its local contacts and short-range contacts.

When the number of nodes in a cluster falls below a threshold, cluster merging occurs. When node *x* leaves the network, it first checks whether its cluster size has fallen below a threshold M_{min}. If the current size is above M_{min}, *x* simply leaves the network by transferring its indices to a randomly selected node in its cluster. Otherwise, this cluster needs to be merged into one of its neighboring clusters within the same semantic cluster. The leaving node triggers cluster merging which is an inversed process of cluster splitting.

Query Routing: The query routing process involves two steps: inter-cluster routing and intra-cluster routing. Upon receiving a query, node *x* first obtains the destination *Semantic Cluster ID* (denoted as *D*). This is done following the same mapping process as described in Section 2.5. Then node *x* will check whether *D* falls into its own semantic cluster by comparing *D* against the most significant *m*-bits of its *ClusterID*. If that is the case, *x* will flood the query to all the local contacts and also forward the query to its short-range contacts in its adjacent clusters corresponding to *D*. The forwarding processes are recursively carried out until all the clusters corresponding to *D* have been covered and all nodes in each of the clusters are reached.

If *D* falls into neither node *x*'s own cluster nor its adjacent semantic cluster, *x* will rely on its long-range contacts to route the query across clusters. To initiate a search, *x* obtains *D* based on a query and checks which cluster range (partitioned by *x*'s long-range contacts) *D* falls into. Then node *x* forwards the query to the closer semantic cluster through its long-range contact. If *D* is closer to SC_x, node *x* will forward the query across its adjacent cluster towards *D*.

2.7 Subscription

Other than a context query which pulls context data from the network, context consumers can issue a subscribed query to subscribe context data and be notified when data changes occur.

Upon receiving a subscription request, a ContextPeer attempts to match it against the context data in its base model. If the request's predicate is of type *DatatypeProperty*, the ContextPeer determines if its base model contains statements with the same *subject-predicate* pair as the request. Similarly, if the predicate is an *ObjectProperty*, the ContextPeer determines if its base model contains statements with the same *predicate-object* pair as the request.

Whenever a change occurs with respect to the subscription request, the *ModelChangedListener* informs the *ContextManager* of the RDF statement that has been added or removed. Subsequently, the *ContextManager* scans through all *IncomingSubscriptions* and identifies those that are affected by the change. This is done in the following manner: Let the added or removed RDF statement be $<\mathtt{subject}_c, \mathtt{predicate}_c, \mathtt{object}_c>$. Let the RDF triple pattern of a particular *IncomingSubscription*'s criteria be $<\mathtt{subject}_{sr}, \mathtt{predicate}_{sr}, \mathtt{object}_{sr}>$. Define the Boolean variable $\mathit{isAffected}_c$ as:

$$\mathtt{isAffectedc} = (\mathtt{subjectc} == \mathtt{subject}_{sr}) \wedge (\mathtt{predicate}_c == \mathtt{predicate}_{sr}) \wedge (\mathtt{object}_c == \mathtt{object}_{sr})$$

where ^ denotes the logical AND operation. A variable can take the value of any arbitrary constant and is thus equal to any constant value. An *IncomingSubscription* is affected by a change *c* if $\mathit{isAffected}_c$ is true. For each affected *IncomingSubscription*, the *ContextManager* sends *QueryHit* messages to all its subscribers to supply them with the updated context data.

3 Application Development

We have fully implemented our framework in Java SDK 1.4.1, and also developed two context-aware applications: Tele-monitoring alert and Tele-medical record.

The Tele-monitoring alert application monitors a patient's health parameters and informs the relevant parties when abnormal signs are observed. The application scenario is illustrated in Fig. 4. Alan just had his heart bypass operation and is now discharged from his hospital and recuperating at home. The doctor gives him a wearable health monitoring device to track his pulse rate, temperature or the blood pressure of the wearer. The Health Monitor runs on ContextPeer with the connections to physical sensors. The HealthCare Assistant application runs in Alan's portable device (i.e., PDA). It tracks the health status of Alan by subscribing to the sensor information provided by Health Monitor. When abnormal situation occurs, for example, HealthCare Assistant detects a drop in blood pressure and a decrease in pulse rate which it deduces could be a case of imminent heart failure, it immediately alert the Hospital Services Assistant in Alan's private hospital or a nearby emergency center. A health status report can also be transferred to the Hospital Services Assistant for the doctor to have more information to diagnose Alan's condition. The HealthCare Assistant will send a SMS to alert a caretaker of the situation.

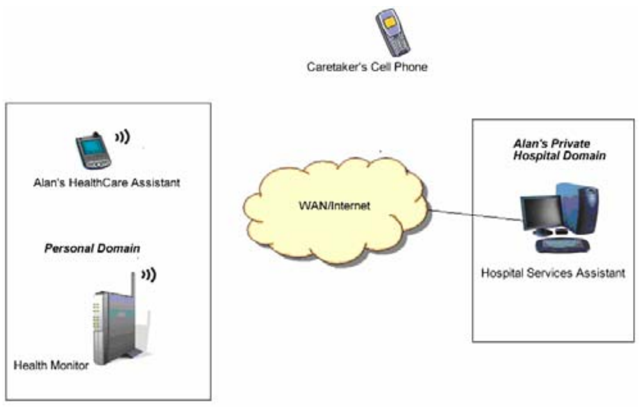
Caretaker's Cell Phone
Alan's HealthCare Assistant
Personal Domain
Health Monitor
WAN/Internet
Alan's Private Hospital Domain
Hospital Services Assistant

Fig. 4. Scenario illustration of the Tele-monitoring alert application

The Tele-medical record application allows for easy transfer of medical records from one hospital to another hospital facilitating the doctor's treatment of a patient. In the above scenario, upon reaching a nearby hospital or an emergency center, Alan's HealthCare Assistant automatically sends his personal information (i.e., name, age, gender, medicine allergies, etc) to a local Hospital Services Assistant. The Hospital Services Assistant checks and realizes Alan is in his first time visiting and hence the hospital does not have Alan's medical record. Thus the Hospital Services Assistant sends a request to Alan's HealthCare Assistant to request for his medical record. The HealthCare Assistant next sends a query to Alan's private hospital's Hospital Services Assistant to retrieve his latest medical record. After receiving the record, the HealthCare Assistant filter out any privacy information, e.g., his drug addiction ten years back, and forward the medical record to the local hospital's Hospital Services Assistant.

4 Prototype Measurements

We have deployed a prototype system to demonstrate the working principle of ContextPeers and assess practical issues. In this section, we report the measurement results obtained from our prototype testbed.

We set up the prototype testbed in a wide-area network. Most of the ContextPeers run on Pentium 800MHz desktop PCs with 256MB memory. We create a set of context ontologies and context data for each ContextPeer. Each ContextPeer stores the upper context ontology and one or more domain-specific context ontologies. Before the evaluation starts, we need to place context ontologies and context data at each ContextPeer. The evaluation starts by connecting each ContextPeer to the network. The network is constructed when ContextPeers randomly join the network. A ContextPeer obtains the IP of an existing ContextPeer from the bootstrap server. We test the bootstrap process by connecting all the ContextPeers to the network in different joining sequences; hence, the structure of the network obtained may differ from one to another.

4.1 Bootstrapping

When a ContextPeer starts, it first goes through the semantic clustering mapping process to identify which semantic cluster to join. The mapping process is done by iterating each of the RDF data triples and identifying its corresponding semantic cluster. Then the ContextPeer chooses the major semantic cluster to join. On average, the program initialization process takes about 4.26 seconds, and the mapping process for each RDF data triple takes about 0.251 ms. The initialization process involves reading and merging the ontology files stored locally and generating internal data structures for mapping. It is done only once when a ContextPeer starts and is only repeated if there is a change in these ontologies. Upon joining the network, each node creates and maintains a set of peers in its routing table. The joining process involves initiating the Join message, connecting to those nodes in the JoinReply message received and registering its reference if needed. The results for different steps in the bootstrap process are summarized in Table 1.

Table 1: The results for the bootstrapping process

Processes	**Average Time Taken**
Program Initialization	4.26 s
Semantic Clustering Mapping	0.251 ms/RDF triple
Joining Process	2.56 s

4.2 Dynamic characteristic

We evaluate the dynamic characteristic of the network in our prototype by forcing ContextPeers to join and leave different semantic clusters randomly. Cluster splitting/merging may occur when the cluster size is greater/lower than the default size. For testing the dynamic characteristic of the network, we introduce a parameter: ***T****ime-to-****S****tability (TS)*. We define the steady state of ContextPeer as the state in which a ContextPeer maintains live connections to the peers in its routing table. The steady state of a ContextPeer may collapse if one of the following events occurs:

- Its *short-range contacts* or *long-range contacts* leave the network or some of these peers change their major semantic clusters (due to their local data change).
- Its reference peer(s) leave the network or their major semantic clusters change.

Queries routing may be affected when ContextPeers are not in the steady state. The *TS* parameter is measured from the time when the steady state of a ContextPeer collapses until it reaches the steady state again. We measure the *TS* of the affected ContextPeers for different test cases and the results are summarized in Table 2 (note that no backup links are used in these cases).

Table 2: Results on *TS*

Test Cases (without backup links)	**Average *TS***
Case 1: The short range contacts or long range contacts leaves the network or changes its major cluster or cluster splitting/merging occurs	271 ms per connection
Case 2: Reference hosting nodes leave/change	87 ms per reference

In a highly dynamic network, peers leave and join frequently; this may result in relapse rate very high. A high relapse rate may affect query routing in the network. To prevent this, we use a backup link for each type of connections. Once the steady state collapses, a ContextPeer can switch to the backup link immediately for the affected connection. With this backup scheme, we can minimize the disruption to query routing in the highly dynamic network where peers frequently leave and join.

4.3 Response time analysis

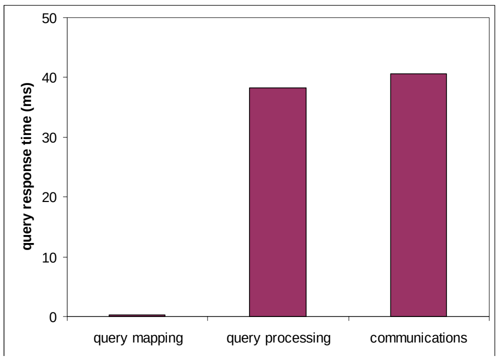

Fig. 5. Response time for context queries

In this experiment, we analyze the important factors that affect the query response. We randomly select context queries, and measure the average response time. The query response time can be broken down into three portions: query mapping, query processing and communication. Query mapping is the time taken by a ContextPeer to map a query to the appropriate semantic cluster(s). Query processing is the time taken by a ContextPeer to process a query. Query processing involves performing a local lookup against the base model. Communication represents the time taken for queries and their responses to travel over the network. It is the sum of the time taken to send a query from a consumer to a ContextPeer and the time taken to send the query's response from the ContextPeer back to the consumer. The results are shown in Fig. 5. As we can see from the above results, the processing time for query mapping can be ignored; the costs of query processing and communication are the major factors.

4.4 Query processing capability

This experiment evaluates the capability of a ContextPeer to process simultaneous queries. In the experiment, a context consumer continuously sends a varying number of queries to the network by randomly picking them from a large query pool. Fig. 6 plots average query processing time against number of simultaneous queries. The graph displays a linear relationship; and shows that the capabilities of a ContextPeer scale well to number of context queries.

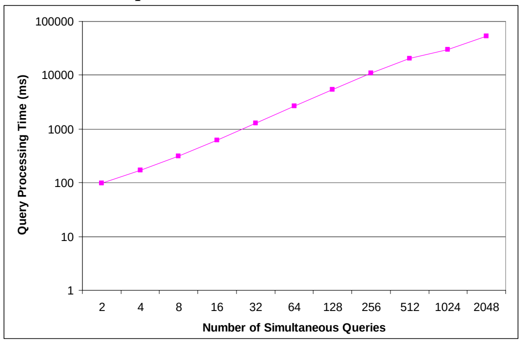

Fig. 6. Query processing capability

5 Related Work

The Context Toolkit [1] provides a software framework and a number of reusable components to support rapid prototyping of sensor-based context-aware applications. However, its context delivery assumes the priori knowledge about the presence of a widget or a context broker. Chen, et al. [2] proposed a platform, named Solar, to support data fusion services and context dissemination to context-aware applications. Solar provides a policy driven data dissemination service based on a multicast tree. However, building a multicast tree for context dissemination may incur large overhead in the presence of node changes. Hong, et al. [3] proposed the Confab infrastructure, which includes a flexible and distributed data store to make it easy to model, store and disseminate context data; and a context specification language for declaratively stating and processing context needs. While our context lookup framework shares the similar idea of distributed context storage of Confab in which the context data is kept close to where it was generated and where it is likely to be used, our emphasis is on how to provide a scalable semantic lookup service using an overlay network in multiple smart spaces. Gaia [11] is an infrastructure supporting the construction of applications for smart spaces. It consists of a set of core services and a framework for building distributed context-aware applications. Different from the context service in Gaia, we focus on providing a semantic lookup service which context information can be shared in a semantic manner. Knoll, et al. [12] proposed a P2P architecture for context-based system based on Pastry [4]. They modified the Pastry algorithm to optimize the data distribution towards geographic locality. In our framework, data distribution is based on where the context data was generated, and nodes are self-organized into the network according to their semantics.

6 Conclusion

This paper presents the design of a semantic P2P framework for context lookup in multiple smart spaces. The framework offers a de-centralized way for acquiring context data from sensors, storing data and resolving context queries.

References

1. Dey, A.K., Salber, D. Abowd, G.D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Anchor article of a spe1cial issue on Context-Aware Computing, Human-Computer Interaction (HCI) Journal, Vol. 16(2-4), pp. 97-166, 2001.
2. Guanling Chen. Solar: Building a Context Fusion Network for Pervasive Computing. Ph.D. Dissertation. Department of Computer Science, Dartmouth College. August 2004.
3. Jason I. Hong and James A. Landay. An Infrastructure Approach to Context-Aware Computing. In Human-Computer Interaction, Vol. 16, 2001.
4. Rowstron, A., Druschel, P. Pastry: Scalable, Distributed Object Location for Routing for Large-scale Peer-to-peer Systems. In Proceedings of IFIP/ACM Middleware 2001, Germany, 2001.
5. Nagel, K., et al. The Family Intercom: Developing a Context-Aware Audio Communication System. In Ubicomp 2001. Atlanta, GA. pp. 176-183. 2001.
6. T. Gu, H. K. Pung, and D. Zhang. Information Retrieval in Schema-based P2P Systems using One-dimensional Semantic Space. Elsevier Journal of Computer Networks, Special Issue on Innovations in Web Infrastructure. 2007.
7. M.Smith, C. Welty, and D. McGuinness. Web Ontology Lanugauge (OWL) Giude. August 2003.
8. Jena 2 - A Semantic Web Framework, http://www.hpl.hp.com/semweb/jena2.htm
9. J. Kleinberg. The Small-World Phenomenon: an Algorithm Perspective. In Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000.
10. RDQL, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.
11. Anand Ranganathan and Roy H. Campbell. A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. In Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, June 2003.
12. Mirko Knoll, Torben Weis: A P2P-Framework for Context-based Information, 1st International Workshop on Requirements and Solutions for Pervasive Software Infrastructures at Pervasive 2006, Dublin, Ireland, May 2006.