A Peer-to-Peer Architecture for Context Lookup

Tao Gu, Edmond Tan, Hung Keng Pung
School of Computing
National University of Singapore
{gutao, tanyikho, punghk}@comp.nus.edu.sg

Daqing Zhang
Institute for Infocomm Research
daqing@i2r.astar.edu.sg

Abstract

As computing technology moves towards pervasive computing, many applications are beginning to make use of context information to adapt to and respond appropriately to their environments. Such a trend necessitates efficient search for context information in wide-area networks. In this paper, we propose a semantic P2P context lookup system. Peers are grouped based on the semantics of their local data which are extracted according to a set of schemas and are self-organized as a semantic overlay network. Context search requests are only routed to the appropriate nodes that have relevant data, reducing unnecessary query traffic and increasing the chances that the context data will be found quickly. To reduce maintenance overheads incurred by high-dimensional semantic overlay networks, we propose a one-dimensional ring space to construct peers and facilitate efficient query routing. Our simulation studies demonstrate the effectiveness of our proposed routing techniques.

1. Introduction

Computing technology is moving towards pervasive computing. In the pervasive computing paradigm, devices, applications and humans are able to interact naturally and seamlessly with each other. In order for this to be possible, it is necessary to augment the capabilities of applications to include awareness of and sensitivity towards their environments, which are often dynamic in nature. This enables applications to respond and adapt appropriately to environmental changes and conditions. Applications that possess these qualities are known as context-aware applications. Context-aware applications typically acquire context information from context providers and subsequently manipulate this information to perform various context-sensitive tasks. Context information is characterized as an application's environment or situation [1], and as a combination of features of the execution environment, including computing, user and physical features [2]. Information on users (name, address, role, etc.), locations (coordinate, temperature, etc.), computational entities (device, network, application, etc.) and activities (scheduled activities, etc.) are some examples of context information. Context data can be represented in different ways. Previous systems have made use attribute-value pairs and Java programming objects to describe and model context data respectively. However, such approaches have limitations. In particular, they have limited semantic interoperability, which is the capability to exchange and share context data between different systems across different domains. To circumvent this problem, we have chosen to model context data using an ontology-based approach. This approach makes use of Resource Description Framework (RDF) statements to represent context data. Each RDF statement is a triple of the form <subject predicate object>.

Context information is usually stored in wide-area networks and consumed by users and applications in different domains. It is therefore necessary to have efficient storage and search mechanisms for context information. Broadly speaking, these services can be provided either via centralized or distributed approaches. Although centralized systems such as RDFStore [3] generally provide faster query responses, the use of a central server subjects the system to limitations such as single points of failure and single processing bottlenecks. Distributed systems on the other hand do not suffer from such limitations. A possible way of building a distributed system is to leverage on Peer-to-Peer (P2P) technology. Such an approach enables distributed context data storage in a network as well as distributed query routing. P2P systems can be structured or unstructured. Structured models such as Chord [4], CAN [5] and Pastry [6], rely on distributed hash functions to determine where data

should be stored. This means that each piece of data would be mapped to and subsequently stored at a particular location on the network, regardless of where the data is originally generated. Unstructured models such as Gnutella [7] do not restrict the placement of data. Edutella [8] and its successor [9] are examples of RDF-based meta-data infrastructures and routing schemes based on Gnutella-like P2P networks. Nodes in this system are organized as a hypercube. However in Edutella, a query has to be flooded to all nodes in the network which may include many nodes that do not contain the relevant context data. It is more efficient to forward a query to nodes which are likely to contain the relevant type of context data. This can potentially lead to a lower network load and better search performance.

In this paper we propose the use of multiple semantic overlays to group various context producer nodes. Each overlay "ties" and manages a set of producer nodes with similar categories of context information. By contacting the nodes in one overlay or multiple overlays in parallel, context queries can be resolved quickly. This approach works efficiently (demonstrated in our simulation) when the dimensionality is reasonably low. However, when the number of semantic overlays increases, the maintenance cost becomes non-trivial or expensive. To reduce the maintenance cost and facilitate efficient search in high-dimensional semantic spaces, we propose a one-dimensional ring space to group and arrange peers upon their joining. The ring structure enables the mapping of the clusters in a k-dimensional semantic space to a one-dimensional semantic space, hence reducing overlay maintenance overhead.

The rest of the paper is organized as follows. We describe related work in Section 2. We then discuss ContextBus in Section 3, and discuss the ring space in Section 4. We present the performance evaluation in Section 5, and conclude the work in Section 6.

2. Related Work

Centralized RDF repositories and lookup systems such as RDFStore, Jena [10] and Sesame [11] have been implemented to support the storing and querying of RDF documents. These systems are very fast and can scale up to many millions of triples. However, they have the same limitations as other centralized approaches, such as single processing bottlenecks and single points of failure. Cai et al. [12] proposed a distributed RDF repository that stores each triple at three places in a multi-attribute addressable network which extends Chord by applying a globally known hash function. Queries can then be efficiently routed to those nodes in the network where the triples in question are known to be stored if they exist. However, storing each RDF triple multiple times in the network increases the storage cost. Schema-based P2P networks such as Edutella and Piazza [13] that combine P2P computing and the Semantic Web are potential candidates for distributed context lookup systems. These systems build upon peers that use explicit schemas to describe their contents. However, current schema-based P2P networks still have some shortcomings, e.g. queries still have to be flooded to every node in the network, making it difficult for the system to scale. Unstructured P2P systems with multiple overlays have also been proposed. In [14], these multiple overlays are called Semantic Overlay Networks (SONs); queries are routed to the appropriate SONs, increasing the chances that matching objects will be found quickly and reducing the search load. We use the concept of multiple semantic overlays in context-aware computing to cluster context producer nodes based on pre-defined schemas; and we mainly focus on the query routing issues both within and across overlay networks. We aim to reduce the overlay maintenance cost incurred by high dimensional semantic overlays. Kleinberg [15] proposed a two-dimensional grid where every node maintains four links to each of its closest neighbors and one long distance link to a node chosen from a probability function. He showed that a query can be routed to any node in $O(\log^2 n)$ hops. Our work is inspired by Kleinberg's small world construction. We show how the basic idea can be applied to a semantic P2P context lookup system.

3. ContextBus Architecture

In this section we describe the use of multiple semantic overlays to organize nodes in our system. The idea behind this scheme is to classify a wide range of context producer nodes into certain groups based on the kind of context data they store. Upon creation, every node will be associated with meta-data and may participate in one or more groups. Nodes in the same group will form an overlay which we call a ContextBus. A query will first be preprocessed and mapped into one particular ContextBus or a subset of ContextBuses, and then routed to these ContextBuses based on the inter-ContextBus routing technique which we will describe in Section 3.2. The overall performance of context search can be improved by forwarding a query only to the nodes which contain the

same type of context information as requested in the query.

The meta-data for classifying context data are defined in context ontologies. As context information has a limited scope compared to other network resources, we should be able to classify a wide range of context data into a manageable number of categories using domain ontologies. We have classified context information into person, location, activity, device, network, etc, and defined the domain ontology for each category [16].

The ContextBus architecture is shown in Figure 1. A peer (which we call a ContextPeer) can act as a context producer, a context consumer, or both. Context producers provide various kinds of context data; whereas context consumers obtain context data by submitting their context queries and receiving query results. Upon creation, context producer peers will be clustered according to their data semantics and associated with one or more ContextBuses. The ContextPeers within the same ContextBus are interconnected and organized using an overlay network. Upon receiving a query, a ContextPeer extracts the semantics from the query, maps it to the relevant ContextBuses, and then routes the query to these ContextBuses. When the query reaches a designated ContextBus, it will be flooded to all peers within this overlay. ContextPeers that receive the query will do a local search and return results appropriately. Each ContextPeer maintains a local context data repository which supports RDF-based semantic query using RDQL [17].

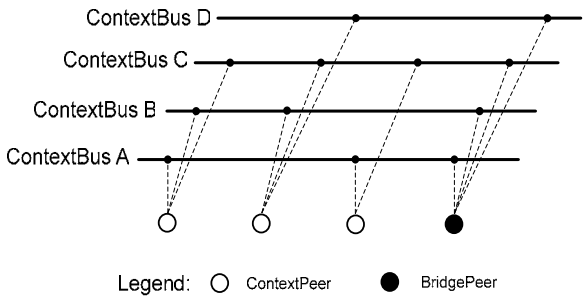

Figure 1. ContextBus architecture

For the rest of this section, we discuss the bootstrapping mechanism that takes place when a new ContextPeer joins the network, followed by the routing strategy across ContextBuses. Then we describe the maintenance of routing tables. In the sections that follow, we will refer to ContextPeers as nodes.

3.1 Bootstrapping

When a new node is created, it will first go through the bootstrapping process to join the network. A bootstrap server maintains information on available nodes in a certain region. We recognize the fact that nodes have different capability constraints, such as the maximum degree (i.e. the number of active connections per node). We classify nodes into two classes: high-degree and low-degree. We define M as the maximum degree of a node and C as the total number of ContextBuses in the system. A node is called a high-degree node if $M \geq C$ and a low-degree node if $M < C$. A node's entry in the bootstrap server is a pair of <*nodeID*, *nodeClass*> indicating the node's ID and its class information. Entries are grouped according to the ContextBuses which nodes participate in. Duplicate entries may exist across different ContextBuses as nodes may join multiple ContextBuses.

When a node x joins the network, it will first contact a bootstrap server and attempt to join certain ContextBuses. It does this by obtaining one or more node IDs for these ContextBuses from the bootstrap server and then connecting to each of these nodes. These node IDs will be stored in node x's routing table. For a high-degree node, the bootstrap process ensures that it is connected to at least one node in each ContextBus. For a low-degree node, it will not be able to connect all ContextBuses due to a limited number of available connections. In this case, we will first satisfy those ContextBuses which provide the same type of context data as provided by the node, and then assign the remaining connections to nodes in other ContextBuses. This ensures that a query for a particular type of context data reaches all nodes providing that type of context data. Here, we assume that the maximum degree of a low-degree node is greater than the number of ContextBuses providing the same types of context data as the node. For the assignment of the remaining connections, a low-degree node must connect to at least one high-degree node. This ensures that a low-degree node is able to route queries to any ContextBus either by itself or through a high-degree node.

Recently, researchers in [18] have realized the topology mismatching problem which limits the performance of various search and routing techniques. To ensure that the ContextBuses mirror the physical network as much as possible, we perceive that it is more efficient to perform topology optimization within each ContextBus upon node joining and leaving. This optimization requires the knowledge of link costs between every two nodes. In [19], a technique has

been proposed to determine these link costs, using the latency between each node to multiple servers. This technique can thus be employed to optimize the topologies of ContextBuses.

3.2 Inter-ContextBus Routing

Upon entry into the system, each node *x* creates a routing table containing a set of node IDs that are grouped according to ContextBus IDs. These nodes are the direct (or one-hop) neighbors of node *x*. As a high-degree node connects to at least one node in each ContextBus, it is able to forward any query to any ContextBus. If a query is generated at a low-capacity node, it will forward the query to a high-degree node if the query is destined for ContextBuses that it is not able to connect to directly. In this case the high-degree node will act as a bridge (which we call a BridgePeer as shown in Figure 1) for the low-degree node that will route the query to the appropriate ContextBuses.

3.3 Routing Table Maintenance

In the event of nodes joining and leaving the system, the routing tables of all affected nodes have to be updated to reflect the current state of the system as accurately as possible. The maintenance of routing tables makes use of the Ping and Pong messages.

In our system, nodes may or may not leave gracefully. In the case of graceful node leaving, the node deciding to leave the system will inform all its neighbors of its intention prior to leaving. Each of its neighbors can then delete the entries corresponding to this node from their tables and perform bootstrapping as necessary. However, if a node does not leave the system gracefully, this node's entries in the routing tables of neighboring nodes will become invalid. To remove such outdated entries, a node periodically sends a Ping to each direct neighbor in its routing table to check its availability. An active neighbor will respond to the Ping with a Pong. A neighbor that does not respond with a Pong is considered dead. The node will then purge that neighbor's entry from its routing table and may proceed to perform bootstrapping for the affected ContextBuses.

4. One-dimensional Ring Space

The ContextBus approach works efficiently when the dimensionality is reasonably low. However, when the number of semantic clusters increases, the maintenance cost becomes non-trivial or expensive. In addition, as the ratio of low-degree nodes to high-degree nodes increases, processing bottlenecks may exist at the high-degree nodes, subsequently decreasing the search efficiency.

In this section, we present a new approach aiming to reduce maintenance cost and facilitate efficient navigation and search in a high-dimensional semantic context space. We build an overlay network using a one-dimensional ring structure which enables the mapping from *k*-dimensional semantic space into a one-dimensional semantic space. We now discuss how to construct the ring space.

4.1 Peer Placement

In our design, one crucial issue is how to design a naming space to facilitate efficient routing and support cluster splitting and merging. We distinguish the concepts of *cluster* and *semantic cluster*. A *cluster* refers to a partition which consists of a set of nodes bundled together such as *cluster C0* and *C1* in Figure 2. A *semantic cluster*, on the other hand, refers to a set of *clusters* corresponding to the same semantics. For example, *cluster C0*, *C1*, *C2*, and *C3* belongs to *semantic cluster SC0*. We propose our cluster encoding scheme as follows. A *Cluster ID* which is represented by an *k*-bit binary string (where $k = m + n$) is an unique ID that identifies a *cluster*. The first *m*-bit binary string (we call it *Semantic Cluster ID*) is used to identify a *semantic cluster* which corresponds to one particular domain context ontology. Hence, the system can have a maximum of 2^k *clusters* and 2^m *semantic clusters*. An example which assumes $k = 5$ and $m = 3$ is illustrated in Figure 2. The rational behind this encoding scheme is that, for a given query, we need to obtain the appropriate *Semantic Cluster ID* (rather than *Cluster ID*) to match the same semantics of the query and route the query among *semantic clusters*. *Semantic clusters* can be viewed as an additional semantic layer on top of actual clusters. A query has to reach all *clusters* within a *semantic cluster* for search completeness.

Upon joining the network, a peer needs to obtain the semantics from its local context data and place itself into an appropriate *semantic cluster*. The computation is done locally at each peer and requires global information (i.e. a set of domain context ontologies) to function. Each of the domain ontologies corresponds to a unique *Semantic Cluster ID*. A new *Semantic Cluster ID* is sequentially generated and placed inside an ID pool when a new type of context data is introduced to the system. Every new node that joins the system will be dynamically assigned an ID from this pool. As a peer may obtain multiple semantics extracted from its context data, we choose

the *semantic cluster* corresponding to the largest set (i.e. majority) of context data and place the peer into this *semantic cluster*. In order for a query to reach all nodes that provide the same semantics, we adopt index publishing. A peer selects a node from each *semantic cluster* excluding the one it joins and publishes its index (i.e. reference pointer) to these nodes. For example, *Peer 1* publishes its index to *semantic cluster SC1* by putting its index to *Peer 3* in *cluster C4* which is selected in random within *SC1*. As a result, a particular *semantic cluster* can be viewed as a set of interconnected nodes separated by *clusters* and a collection of references stored in these nodes which point to the other nodes in other *clusters* where context data is physically stored.

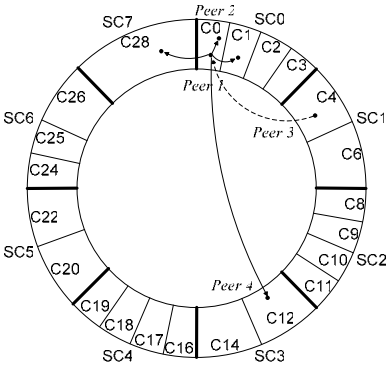

Figure 2. One-dimensional ring space

The above scheme has several positive effects. For example, if a peer has homogeneous data in its local repository, most of its data will fall into one corresponding *semantic cluster*, therefore reducing the cost to publish data indices. This is likely to be the case in context-aware environments, i.e. a context producer node usually provides homogeneous context data in a real scenario. Such nodes can be, for example, an in-house location node that provides location information for all users in a smart home or a node stationed in an organization that provides personal profiles for all employees. Furthermore, in many cases, a query issued by a peer shares the same semantics as those of its nearby peers. For example, many context-ware applications are designed in such a way that a node is likely to query for context data available in its nearby nodes. By placing a node based on the majority of its context data into one particular *cluster*, context search can be performed efficiently. While we assume single cluster joint points here, multi-cluster joint points can also be used.

4.2 Ring Construction

In this system, *clusters* are placed in the ring based on their *cluster IDs*. Each node maintains a set of node entries in its routing table for the purpose of both intra-cluster routing and inter-cluster routing. A node, say *x*, decides which *semantic cluster* to participate in based on its context data and randomly picks a *cluster* within this *semantic cluster* to join. It joins the *cluster* by connecting to and keeping track of a number of nodes in the *cluster*. The nodes within this *cluster* are interconnected. These node entries (called *x*'s neighbors in its own *cluster*) will be maintained in *x*'s routing table as intra-cluster routing information. Node *x* also maintains two node entries in each of its adjacent *clusters*. We call these two nodes *x*'s neighbors in its adjacent *clusters*. For example, *Peer 1* keeps track of a node in its own *cluster C0* and another two nodes in its adjacent *clusters* - *C1* and *C28* respectively. Each new node that wishes to join the network will perform this operation. This results in all the *clusters* being linked linearly in a ring fashion. With this ring structure, a *k*-dimensional semantic space can be reduced to one-dimensional semantic space. Maintaining two neighbors in the adjacent *clusters* for every node also ensures that a query generated at any node will be able to reach any other *cluster* by navigating the ring space. However, queries have to be passed around the ring linearly until the destination *cluster* is reached. This approach may not be efficient when the number of *semantic clusters* is large. To accelerate search across *clusters*, node *x* also maintains a set of nodes in other *semantic clusters* other than the two adjacent *clusters*. These nodes provide random *shortcuts* (similar to long contacts in Kleinberg's small world) for node *x* to route a query to other *semantic clusters* quickly. For example, in Figure 2, node *x* creates and keeps track of a *shortcut* to *Peer 4*.

4.3 Cluster Splitting and Merging

The operations of cluster splitting and merging enable our system to scale to a large number of peers. Let *M* represent the maximum cluster size. If a *cluster* size exceeds *M*, the splitting process is invoked to split the *cluster* into two. When a node *x* joins the network, it sends a join request message to an existing node, says *y*. If y falls into the same *semantic cluster* that *x* wishes to join, *x* will then join *y*'s *cluster* by connecting to *y* if the *cluster* size is below *M*; otherwise *y* performs a search to direct the request to a node *z* in the *semantic cluster* that *x* wishes to join, and

subsequently *x* connects to *z* if *z*'s *cluster* size does not exceed *M*. If the *cluster* size exceeds *M*, the node (i.e. *y* or *z*, which we call an initial node) will initiate the cluster splitting process. Cluster splitting partitions a *cluster* into two *clusters* of equal size. A *semantic cluster* can be split into a maximum number of 2^n *clusters*. After splitting, a node updates its *cluster ID* and also the neighbors list in both its own *cluster* and its adjacent *cluster*.

When a node *x* leaves the network, it first checks whether the current *cluster* size has dropped down to 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 with one of its neighboring *clusters* within the same *semantic cluster*. The leaving node triggers cluster merging which is an inversed process of cluster splitting. If the last node in a *semantic cluster* leaves, it initiates two messages to all the nodes in its two adjacent *clusters* informing them to update their neighbor lists. Subsequently, the *semantic cluster* will be removed from the system.

4.4 Query Routing

In this section, we describe the search operation. As described above, each node *x* maintains a routing table with a set of node entries <*NodeID*, *ClusterID*> in *x*'s own *cluster*, two adjacent *clusters* and *shortcuts*. It also keeps the state information about its own cluster, consisting of a *k-bit ClusterID* which indicates the cluster it resides in and *ClusterSize* which is the number of nodes in the cluster. In addition, a node also maintains a number of indices. The query routing process involves two steps: inter-cluster routing and intra-cluster routing. When node *x* receives a query, a *Semantic Cluster ID* is generated based on the semantics of the query. This ID, denoted as *D*, is the destination semantic cluster the query is searching for. Node *x* will first 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 nodes in its own *cluster* and also forward the query to the nodes in its adjacent *clusters* corresponding to *D*. The first node in each of these adjacent *clusters* is always responsible for flooding the query in its *cluster* and forwarding the query to its adjacent *cluster*. The forwarding processes are recursively carried out until all the *clusters* corresponding to *D* are covered. Every node, upon receiving a query, will check its local data repository and return any matching context data and indices. If *D* falls into *x*'s adjacent *semantic cluster*, the query will be forwarded to *D* and flooded to all the *clusters* corresponding to *D*. If *D* neither falls into *x*'s own *cluster* nor its adjacent *semantic cluster*, *x* will rely on its *shortcuts* to route the query across *clusters*.

In the design of these *shortcuts*, we have several design options. We need to decide which *semantic cluster* each *shortcut* points to and how many *shortcuts* each node maintains. An intuitive strategy is to select a set of *semantic clusters* randomly and assign each *shortcut* to a node in each of the *semantic clusters*. Each node can have *s shortcuts* ($s \geq 1$) with the tradeoff that the cost of creating and maintaining these *shortcuts* is proportional to *s*. Upon receiving a query, if the distance between *D* and the *semantic cluster* that its *shortcuts* point to falls below a threshold (a preset minimum distance in terms of number of hops) the query will be forwarded to the closest *semantic cluster*. Subsequently the query will hop towards the destination *semantic cluster*. If not, node *x* selects a *shortcut* randomly, and forwards the query to this node. The same process is invoked until the distance to *D* is below the threshold. This approach is similar to Kleinberg's Small World network model where each node maintains four links to each of its closest neighbors and one long distance link to a node chosen with a probability function.

5. Evaluation

In this section, we use simulations to evaluate the effectiveness of our system, and compare its performance to the Gnutella protocol. We first describe our simulation model and the metrics. Then we report some preliminary results from a range of experiments.

5.1 Simulation Model and Metrics

In our simulation model, we have two types of network topologies: physical topology and P2P overlay topology. The physical topology represents the real-world Internet topology. The P2P overlay topology is built on top of the physical topology. Previous studies have shown that both Internet physical topologies [20] and P2P overlay topologies [21] follow the small world and power law properties. We generate these topologies using the AS model since it has both small world and power law properties.

Context data are classified into a set of categories. There are different sets of keywords for different categories. Each of these keywords maps to a set of context data in each category. Context queries are modeled as searches for specific keywords. All context data associated with a specific keyword are potential hits for a query with that keyword. Context data are

randomly replicated on nodes at a fraction α. Thus, querying for a keyword with fraction α implies that a query hit can be found at a fraction α of all the nodes in the system. Each node x is also assigned a query generation rate, which is the number of queries that node x generates per unit time. In our experiments, each node generates queries at a constant rate. If a node receives queries at a rate that exceeds its capacity to process them, the excess queries are queued in its buffer until the node is ready to read the queries from the buffer. In our simulation study, we use a Gnutella overlay network to organize nodes within a *cluster* or a ContextBus.

For the evaluation of ContextBus, each node in our system is assigned a class ID (1: high-degree or 0: low-degree) based on the number of degrees they have and the total number of ContextBuses in the system. In our system, a high-degree node may not necessarily be a high-capacity node. For the evaluation of the ring space, the simulation is started by having a pre-existing node in the network and then performing a series of join operations invoked by new coming nodes. A node joins a semantic cluster based on its local context data and publishes its data indices. Various context data are mapped into different *semantic clusters* and each *cluster* is associated with a unique ID ranging from $0 \sim 2^m$. If a *semantic cluster* exceeds the maximum size M, it will be split into two *clusters* and this operation may continue until the number of *clusters* reaches 2^n. After the network reaches a certain size, a mixture of node joining and leaving operations are invoked to simulate the dynamic nature of the overlay network. We use the following performance metrics:

Number of nodes contacted per query: this captures the efficiency of a search system.

Search path length: the average number of hops traversed by a query to the destination.

Search cost: the average number of query messages incurred during a search operation in the network.

Maintenance cost: the average number of messages incurred when a node joins or leaves the network. It consists of the costs of node joining and leaving, cluster splitting and merging as well as index publishing. We measured these costs in terms of number of messages.

Search completeness: the ratio of the number of nodes contacted per query to the total number of nodes in a particular *cluster* or ContextBus. Its value lies in the range 0 to 1.

5.2 Search Efficiency

The efficiency of executing a search request is captured in the fraction of nodes contacted and search path length during the search. For a given query, the ring space only needs to contact a fraction $N/2^m$ of nodes where N is the total number of nodes in the system as well as those nodes pointed to by a set of indices. The fraction of nodes contacted per query in the ring space decreases as m increases. In the case of ContextBus, the fraction of nodes contacted is equal to $\overline{C}/\ C_{max}$, where $\overline{C}$ is the average number of ContextBuses each node participates in and C_{max} is the maximum number of ContextBuses in the system. In the experiments, we set C_{max} to 32 and vary the average number of ContextBuses each node participates in from 4 to 32. The average fraction of nodes contacted per query is shown in Table 1. As expected, ContextBus only contacts a fraction of the nodes depending on $\overline{C}$. The smaller the value of $\overline{C}$, the fewer the number of nodes that will be contacted for a query. Notice that for a search request, Gnutella has to contact every node in the network. With less nodes contacted by ContextBus and the ring space, the network traffic load incurred by a query will also be reduced. The search completeness for both the ring space and ContextBus are equal to 1.

Table 1. Average fraction of nodes contacted per query

	Gnutella	ContextBus		
Avg ContextBuses per Node ($\overline{C}$)	N.A.	4	8	16
Avg Nodes Contacted per Query	100%	11%	25.7%	48.6%

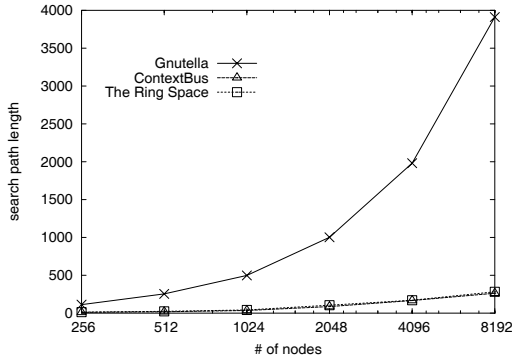

Figure 3. Search path length

Figure 3 shows the search path length comparing the ring space, ContextBus and Gnutella when the network size is varied from 2^8 to 2^{13}. We set M to 1

and n to 0, so that there will be no flooding within a semantic cluster. As shown in Figure 3, the search path lengths for both the ring space and ContextBus increase slowly with the network size as compared to Gnutella. The search path length for the ring space is almost identical to the one for ContextBus, showing that they have the same search effectiveness.

5.2 Overheads

In this experiment, we evaluated search cost by comparing search costs among the ring space, ContextBus and Gnutella. We set the number of *semantic clusters* to 16 and 32 respectively, and varied the network size from 2^8 to 2^{13}. As shown in Figure 4, the search cost of Gnutella increases rapidly when the network size grows. In contrast, the ring space and ContextBus significantly reduce the search cost with the settings of 16 and 32 *semantic clusters*.

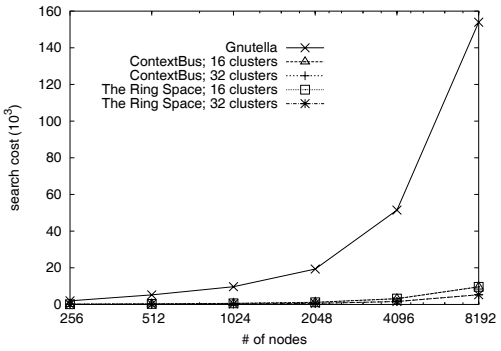

Figure 4. Search cost

Subsequently, we evaluate the average maintenance cost by comparing the ring space and ContextBus. The maintenance cost of ContextBus only includes the cost of node joining and leaving. As shown in Figure 5, the maintenance cost for ContextBus increases rapidly when the number of *semantic clusters* (dimensions) grows. This is because the required number of outgoing degrees for a node in ContextBus increases in proportion to the dimension. In the case of the ring space (M = 32 and n = 2), the average maintenance cost of a node consists of the costs of node joining and leaving, cluster splitting and merging as well as index publishing. The maintenance cost in the ring space also increases with respect to the dimension, but much more gradually. This confirms our design goal of reducing maintenance overheads incurred by high-dimensional semantic overlay networks.

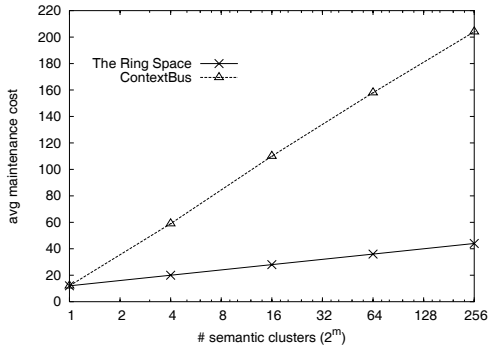

Figure 5. Average maintenance cost

6. Conclusions and Future Work

We have proposed ContextBus and the ring structure to group peers based on pre-defined ontologies. It is important to know that this concept can be well applied to any P2P searching systems where schemas are explicitly defined such as P2P searching for RDF-based web information. Our preliminary simulation results show that our system works effectively. We are currently optimizing the performance of our system. In this paper we assume the use of Gnutella-like overlay networks to organize peers within a cluster; a DHT-based overlay network can be used to provide a more efficient routing scheme as compared to flooding within a cluster. We are also studying how our proposed techniques can be effectively applied to other P2P systems such as CHORD and CAN. We also plan to build a prototype system to deploy our proposed techniques in real life applications.

7. References

[1] R. Hull, P. Neaves, and J. Bedford-Roberts. Towards Situated Computing. In Proceedings of the 1st International Symposium on Wearable Computers, Cambridge, October 1997.

[2] B. Schilit, N. Adams, and R. Want. Context-aware Computing Applications. In Proceedings of Workshop on Mobile Computing Systems and Applications, Santa Cruz, December 1994.

[3] RDFStore. http://rdfstore.sourceforge.net.

[4] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proceedings of ACM SIGCOMM, 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In Proceedings of ACM SIGCOMM, 2001.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale peer-to-peer systems. In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms, Lecture Notes in Computer Science, 2218:161–172, November 2001.

[7] Gnutella, http://gnutella.wego.com

[8] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. S. A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P Networking Infrastructure based on RDF. In Proceedings of the 11th World Wide Web Conference, 2002.

[9] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Lser. Super-peer-based Routing and Clustering Strategies for RDF-based Peer-to-Peer Networks. In Proceedings of the 12th World Wide Web Conference, May 2003.

[10] Jena 2 - A Semantic Web Framework, http://www.hpl.hp.com/semweb/jena2.htm

[11] J. Broekstra and A. Kampman and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Proc. of the 1st International Semantic Web Conference, Sardinia, Italia, June, 2002.

[12] M. Cai and M. Frank. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In Proceedings of the13th International World Wide Web Conference, New York, May 2004.

[13] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In Proceedings of the 12th International World Wide Web Conference, Budapest, Hungary, May 2003.

[14] A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Technical report. Stanford University.

[15] J. Kleinberg. The Small-World Phenomenon: an Algorithm Perspective. In Proc. of the 32nd ACM Symposium on Theory of Computing, 2000.

[16] T. Gu, H. K. Pung, and D. Zhang. A Service-Oriented Middleware for Building Context-Aware Services. Journal of Network and Computer Applications, Vol. 28, Issue 1, pp. 1-18, January 2005.

[17] RDQL, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/

[18] Y. Liu, X. Liu, L. Xiao, L. M. Ni, and X. Zhang. Location-Aware Topology Matching in P2P Systems. In Proceedings of IEEE INFOCOM 2004, Hong Kong, China, March 2004

[19] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays using Global Soft-State. In Proceedings of International Conference on Distributed Computing Systems, 2003.

[20] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, Network Topology Generators: Degree-Based vs. Structural, in Proceedings of ACM SIGCOMM'02, 2002.

[21] S. Saroiu, P. Gummadi, and S. Gribble, A Measurement Study of Peer-to-Peer File Sharing Systems, in Proceedings of Multimedia Computing and Networking, 2002.