A Peer-to-Peer Overlay for Context Information Search

Tao Gu[1,2], Hung Keng Pung
[1] School of Computing, National University of Singapore
3 Science Drive 2, Singapore
{gutao, punghk}@comp.nus.edu.sg

Daqing Zhang
[2] Institute for Infocomm Research
21 Heng Mui Keng Terrace, Singapore
daqing@i2r.a-star.edu.sg

Abstract - **The widespread use of context information necessitates an efficient wide-area lookup service in pervasive computing. In this paper, we present Semantic Context Space (SCS), a semantic overlay network that facilitates efficient search for context information in distributed environments. Peers in SCS are grouped based on the semantics of their local data and self-organized into a one-dimensional ring space. Context search requests are only routed to the appropriate semantic clusters, reducing unnecessary search cost on peers that have irrelevant context data, and increasing the chances that the context data will be found quickly. By exploring parallelism in a semantic cluster, search request can be found quickly. Our simulation studies demonstrate the effectiveness of SCS.**

Keywords - Peer-to-Peer, Distributed Context Search, Semantic Clustering, Routing, Algorithms

1. INTRODUCTION

In recent years, the use of context information has attracted a lot of attention from researchers and industry participates. Users and applications are often interested in searching and utilizing widespread context information. Context information is characterized as an application's environments or situations [1]. With the vast amount of context information, how to provide an efficient context searching service is challenging in the context-aware research community. One approach is to use a centralized search engine to store context data and resolve search requests. Although this approach can provide fast responses to a context query, it has limitations such as scalability, processing bottlenecks and single points of failure. Peer-to-Peer (P2P) approaches, on the other hand, have been proposed to overcome these obstacles and are gaining popularity recently. P2P systems such as Chord [3], CAN [4], Pastry [5] and Tapestry [6] typically implement distributed hash tables and use hashed keys to direct a lookup request to the specific nodes by leveraging on a structured overlay network. However, data placement in these systems is tightly controlled based on the distributed hash function; and updating the relevant information on peer joining/leaving and content changes may introduces a high maintenance overhead. When dealing with context data, it is more efficient that context data is stored close to a node where it is generated and where it is likely to be used. For example, when a person comes back home, a location sensor detects his/her presence. It is more efficient to store his/her location data in a node at his/her home rather than a hashed node which may be far away from its source. As context information exhibits the dynamic characteristics [2], peers may join or leave the system frequently and context data stored in the peers may be changed rapidly. These characteristics imply that higher maintenance overheads may occur in the DHT-based overlay networks. Other P2P systems such as Gnutella [7] allow nodes to interconnect freely and do not impose any structure on the resources that are stored. These systems have low maintenance overhead. However, a query has to be flooded to all nodes in the network, which may include many nodes that do not contain the relevant data. The fundamental problem that makes search in these systems difficult is that data are randomly distributed in the overlay network with respect to their semantics. Given a search request, the system either has to search a large amount of nodes or run a risk of missing relevant data.

In this paper, we present Semantic Context Space (SCS), a one-dimensional semantic space where context data are stored and retrieved according to their semantics. Context data which are semantically similar are "tied" together in SCS so that they can be retrieved by a context query which has the same semantics. As a result, the system is able to forward a query to nodes which are likely to contain the relevant context data. This can potentially lead to a lower network load and a better search performance. The basic idea has been presented in our earlier work - ContextBus [8]. However, when the number of semantic clusters increases, the maintenance cost becomes very expensive. In this paper, we propose the SCS overlay network to overcome the drawbacks of the ContextBus architecture. The rest of the paper is organized as follows. We describe related work in Section 2. We then describe Semantic Context Space in Section 3. We present the simulation results in Section 4, and finally conclude the work in Section 5.

2. RELATED WORK

Cai et al. [9] 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, their solution is based on DHT and is thus unsuitable for context lookup as we have discussed in the previous section. Schema-based P2P networks such as Edutella [10] 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. Crespo et al. [11] proposed Semantic Overlay Networks (SONs) which queries are only routed to the appropriate SONs, increasing the chances that matching objects will be found quickly and reducing the search load. Kleinberg [12] proposed a two-dimensional grid where every node maintains for links to each of its closest neighbors and one long distance link to a node chosen from a probability function.

3. SEMANTIC CONTEXT SPACE

3.1 Overview

In SCS, a large number of nodes are arranged and self-organized into a semantic overlay network, in accordance with their semantics. Upon creation, peers will be grouped according to their data semantics and mapped into a semantic cluster in SCS. Each peer is responsible for managing its own context data corresponding to a semantic cluster and publishing their data indices to peers in other clusters. These indices serve as node pointers which provide location references regarding to where context data is physically stored. Upon receiving a context query, a peer first pre-processes the query and obtains the semantic cluster information associated with the query, and then route the query to an appropriate cluster in SCS. When the query reaches the designated cluster, it floods to all peers within this cluster. Peers that receive a query do a local search and report the results if available. Each peer maintains a local context data repository which supports RDF-based semantic query using RDQL [13].

There are several critical issues need to be addressed in the design of SCS. First, how to extract the semantics from both context data and queries efficiently and precisely, and how to cluster peers in accordance with their data semantics are critical in the first place. We propose to use ontologies as metadata to extract data semantics. The formal design of context ontologies minimizes the problems of synonyms and polysemy. Secondly, how to facilitate efficient navigation and search while minimizing maintenance cost in SCS? To enable navigation and search between clusters, an intuitive solution is to construct k-dimensional semantic clusters by connecting each peer to all dimensions of the corresponding clusters such as in the ContextBus architecture. However, for a high-dimensional semantic space, this approach makes maintenance costly. To address this problem, we propose a one-dimensional ring structure which enables the mapping of the clusters in k-dimensional space to one-dimensional space. Thirdly, how does the overlay resolve the issues related to the scalability, load balancing and fault tolerance? To address these issues, we propose a cluster encoding scheme which allows sub-clustering in a semantic cluster. The system can self-adapt to the number of peers by the operations of cluster splitting and merging. This scheme also enables us to search context data in a parallel fashion.

3.2 Ontology-based Semantic Clustering

In this section, we describe how to use ontology to extract the semantics of both data and queries. The semantics of context data are represented by schema, i.e. context ontology. Various context data are structured and classified according to these ontologies. This ontological structure is also exploited to extract the query semantics and formulate context query. We adopt a two-tier hierarchy in the ontology design. The upper ontology which defines common concepts is shared by all peers. Each peer can define its own concepts in its low-layer ontologies. Different peers may store different sets of low-layer ontologies based on their applications' needs. An example of ontological structure is shown in Figure 1.

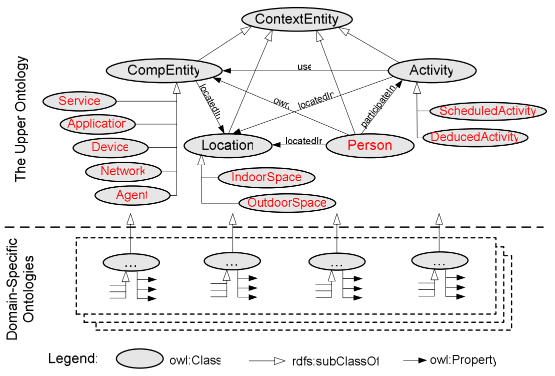

Figure 1. An example of ontological structure

The leaf nodes in the upper ontology are used as semantic clusters to cluster peers, and denoted as a set $E = \{Service, Application, Devices, \ldots\}$. Each of these pre-defined semantic clusters will be assigned with a unique ID upon their presence in SCS. The mapping computation is done locally at each peer. For the mapping of RDF data, a peer needs to define a set of low-layer ontologies and store them locally. Upon joining SCS, a peer first obtains the upper ontology and merges it with its local low-layer ontology. Then it creates instances (i.e. RDF data) and adds them into the merged ontology to form its local knowledge base. A peer can map its local data into one or more semantic clusters by extracting the predicates of its RDF triples. For example, we can map predicate '*locatedIn*' into semantic cluster '*IndoorSpace*' by checking its *rdfs:range* if the predicate is of type *ObjectProperty*. If the predicate is of type *DataTypeProperty*, for example, '*lightLevel*', we will check its *rdfs:domain* to get the class - '*Location*'. As '*Location*' is not a leaf node in the upper ontology, we need to find out its subclasses/superclasses until the leaf nodes are reached. Finally '*lightLevel*' is mapped into both '*IndoorSpace*' and '*OutdoorSpace*' semantic clusters.

3.3 One-dimensional Ring Structure

Peer Placement: 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 ID* (described in the next sub-section) which will be assigned dynamically. 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 to place the peer. In order for a query to reach all nodes that provide the same semantics, we adopt index publishing. Other than the semantic cluster a peer joins, the peer selects a node in each of the rest semantic clusters and publishes its index 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 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 where context data is physically stored. While we assume single cluster joint point here, multi-cluster joint points can be used as well.

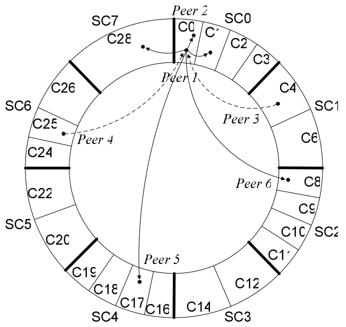

Figure 2. One-dimensional ring structure

Cluster Naming Scheme: In SCS, 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*; *cluster C20* and *C22* belongs to *semantic cluster SC5*. 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, a SCS can have a maximum of 2^k *clusters* and 2^m *semantic clusters*. An example of a SCS 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. Partitioning peers into a set of clusters within the same semantic cluster also facilitates load balancing and enables parallel search within the same semantic cluster.

Ring Construction: In SCS, 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 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 just like a Gnutella-like overlay network. 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, in Figure 2, *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 node who wishes to join the network performs this operation; resulting all the clusters are linked linearly in a ring fashion. With this ring structure, a k-dimensional semantic space can be linearized; and hence it significantly reduces the dimensionality of the semantic space. Maintaining two neighbors in the adjacent clusters for every node in SCS 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 either clockwise or anticlockwise until the destination cluster is reached. To accelerate search across clusters, node x maintains a set of nodes in other semantic clusters except the two adjacent clusters. These nodes provide *shortcuts* for node x to route a query to other semantic clusters quickly. For example, in Figure 2, node x creates and keeps track of two *shortcuts*: - one points to *Peer 5* and the other points to *Peer 6*. When a new semantic cluster is inserted into the ring space or in the process of cluster splitting/merging, a node needs to update the neighboring nodes in both its own cluster and its adjacent clusters. However, a node only needs to update its *shortcuts* upon the insertion or deletion of a semantic cluster as a *shortcut* points to an appropriate semantic cluster rather than a cluster.

Cluster Splitting and Merging: If a cluster size exceeds M (the maximum cluster size), the splitting process is invoked to split the cluster into two. Each node maintains a parameter called *CurrentLoad* which measures the current load of the node in terms of the number of triples and data indices it stores. When a node, say x, joins the network, it sends a join request message to an existing node, says y. If y falls into the same semantic cluster x wishes to join, x joins the cluster by connecting to *y* provided the cluster size is below *M*. Otherwise, *y* performs a search to direct the request to a node, say *z*, in the semantic cluster *x* wishes to join, and then *x* connects to *z* if the cluster does not exceed *M*. If the cluster size exceeds *M*, the node (i.e. *y* or *z*, we call an initial node) will initial the splitting process. The initial node first obtains a sorted list of all the nodes according to their *CurrentLoads* by polling. Then it assigns these nodes to the two sub-clusters alternatively. After the 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 (where $n = k - m$). For example, assuming $m = 3$, $n = 2$, and there exists a cluster *C4* in the network. Initially, the *bit split pointer* points to the most significant bit of the *n*-bit string. When a cluster splitting occurs, the bit pointed by the *bit split pointer* is split into *0* and *1*, and then move the pointer forwards to the next bit in the *n*-bit string. Therefore, we obtain cluster ID - *C4* and *C6*, and both IDs correspond to the same semantic cluster *SC1*. Cluster *C4* or *C6* can be further split into *C4* and *C5* or *C6* and *C7*; and finally the *bit split pointer* is set to null which indicates no cluster splitting is allowed. The same mechanism follows for insertion of a new semantic cluster. After splitting, a node updates its *cluster ID*, the *bit split pointer*, the neighbors list in both its own cluster and its adjacent clusters. Cluster merging is an inverse process of cluster splitting, and hence we will not go into the details.

3.4 The Search Algorithm

In SCS, each node, say *x*, maintains a routing table with a set of node entries <*NodeID*, *ClusterID*> in *x*'s own cluster, two adjacent clusters and another two semantic clusters. It also keeps the state information about its own cluster, consisting of a *k*-bit *ClusterID* (where $k = m + n$) which indicates the cluster it resides in and *ClusterSize* which specifies the current size of its cluster. Each 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 to flood 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 the matched context data and indices. For example, as illustrated in Figure 3a, *Peer 1* in *C0* forwards the query to *C1*, the first node in *C1* forwards the query to a node in *C2*, etc; also the query is flooded in each of these clusters. 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*. For example, in Figure 3a, a query generated at *Peer 2* with *D* = *SC3* will go through *C16* and be flooded in *C14* and *C12*. If *D* neither falls into *x*'s own cluster nor its adjacent semantic cluster, *x* will reply on its *shortcuts* to route the query across clusters.

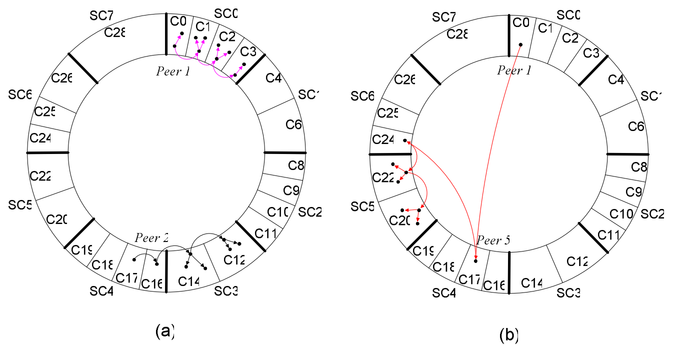

Figure 3. Query routing in SCS

In the design of *shortcuts*, our approach is based on the observation of the ring space can be equally divided into several partitions. Each node maintains two *shortcuts* that are used to partition the ring space; a query can be routed to a semantic cluster which is closer to the destination semantic cluster quickly with the helps of these *shortcuts*. Given the maximum cluster size *M*, the system can have a total of $M \cdot 2^{m+n-1}$ nodes when $M_{min} = 1$. Let *Cx* represent the cluster where node *x* resides in and SC_x denote the semantic cluster that *Cx* corresponds to. *SCx* can be obtained by truncating *Cx* to *m* bits from the most significant bit. The two semantic clusters SC_{half} and $SC_{quarter}$ that *x*'s *shortcuts* point to are denoted as $(SC_x + 2^i) \bmod 2^m$, where $i = m - 1, m - 2$. To initial a search, node *x* obtains *D* based on a query and checks which cluster range partitioned by *x*'s *shortcuts* D falls into. Then node *x* forwards the query to the closer semantic cluster through its *shortcut*. If *D* is closer to SC_x, *x* will forward the query across its adjacent cluster towards *D*. For example, as shown in Figure 3b, *Peer 1* generates a query and computes the destination semantic cluster as *SC5*. *Peer 1* first realizes that *SC5* falls into the interval [*SC4*, *SC0*] and *SC4* is close to *SC5*. Then *Peer 1* forwards the query to *Peer 5* at *C17*. As *SC5* falls into [*SC4*, *SC6*] and *C24* is closer to *SC5* as compared to *C17*. Hence, *Peer 5* forwards the query to *SC6* through its quarter *shotcuts*. Finally, the query reaches *SC3* and is then flooded in both *C22* and *C20*.

The more *shortcuts* we create, the finer the granularity we get. If a node maintains its *shortcuts* that point to every other semantic cluster, it is identical to the ContextBus architecture. However, more *shortcuts* imply higher cost of creating, updating and maintaining these *shortcuts*. In SCS, we set the number of *shortcuts* to two. To partition the ring space in a finer granularity when the number of semantic cluster m increases, we can place the longest *shortcut* into different points. The other *shortcut* always points to the middle semantic cluster between SC_x and the semantic cluster that the longest *shortcut* points to. For example, if we place the longest shortcut to one-quarter of the ring, the ring space is divided by eight, and so on.

4. EVALUATION

In this section, we use simulations to evaluate SCS. We first describe our simulation model and the metrics. Then we report the results from a range of experiments.

4.1 Simulation Model and Metrics

To simulate the performance of SCS in a more realistic environment, we create two types of network topologies in our model: 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. All peer nodes are a subset of nodes in physical topology. We generate these topologies using the AS model since it has the properties of both the small world and power law.

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. If a semantic cluster exceeds the maximum size M, it will be split into two and this operation may continue until the number of sub-clusters reaches 2^n. After the network reaches a certain size, a mixture of node joining and leaving are invoked to simulate the dynamic characteristic of the overlay network. Context data are randomly replicated on nodes at a fraction α. A query is selected randomly among different semantic dimensions. In our simulation study, we use a Gnutella overlay network to organize nodes within a cluster. The average outgoing degree in a cluster is set to 4 and shortcuts are set to the half and quarter of the ring space. To measure the effectiveness of SCS, we use the following performance metrics:

Fraction of nodes contacted per query is the average fraction of nodes contacted for a query. It captures the efficiency of a lookup system.

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

Search cost is the average number of query messages incurred during a lookup operation in the network.

Maintenance cost is 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/merging and index publishing. We measured these costs in terms of number of messages.

In the following sections, we will report the effects of parallel search and clustering. The results for fraction of nodes contacted per query, search path length and search cost are omitted due to the space limit.

4.2 The Effect of Parallel Search

In SCS, we explore the parallel search mechanism within a semantic cluster. We evaluated the parallel search effect by comparing SCS and ContextBus. We set up a network with $m = 4$ and varied the network size from 2^{10} to 2^{13}. We set $n = 2$ and 3 respectively for SCS, as a result a semantic cluster will be split into two when the size exceeds $N/2^5$and $N/2^6$. Hence a search can be performed in parallel among these sub-clusters. Figure 4 shows that the parallelism in SCS effectively reduces the search path length as compared to ContextBus.

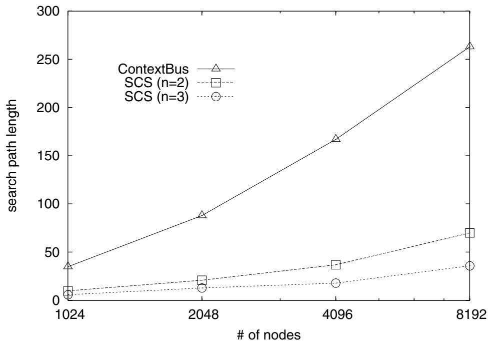

Figure 4. The effect of parallel search in SCS

4.3 Clustering Effects

We evaluate the effect of clustering in SCS by varying the cluster size M from 2^0 to 2^{10}. We first evaluate the effect of cluster size on search path length by setting a network of size $N = 2^{10}$. We turn off the parallel search within a semantic cluster by setting $n = 0$, and ensure no data duplication in SCS. Hence all clusters are semantic clusters. Figure 5 plots the search path length in SCS when M increases from 2^0 to 2^{10}. The search path length across clusters increases while the search path length within clusters decreases with larger cluster sizes (note that there are 2^{10} clusters in the network when $M=1$ and only one cluster when M = 2^{10}). This is because that with a fixed network size, the total number of clusters in SCS

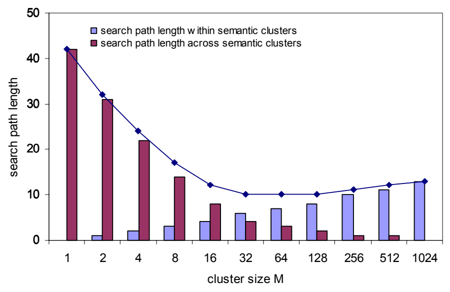

Figure 5. Search path length vs. cluster size *M*

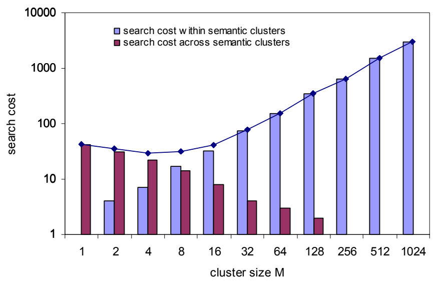

Figure 6. Search cost vs. cluster size *M*

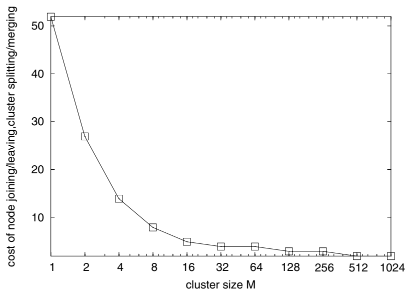

Figure 7. The cost of node joining/leaving and cluster splitting/merging vs. cluster size *M*

decreases with larger cluster sizes.

With the same setting as in the previous experiment, we evaluated the search cost and its breakdown within clusters and across clusters with various cluster sizes. From Figure 6, we observe that the search cost in SCS increases rapidly from a point where *M* =16. This is due to the effect of blind flooding within a cluster.

We plot the cost of node joining/leaving and cluster splitting/merging over different cluster sizes in Figure 7. As there are lesser clusters in SCS with larger cluster sizes, a new node requires a smaller number of hops to join the network. Therefore the cost of joining/leaving decreases with respect to *M*. With a larger cluster size, cluster splitting and merging occur less frequently, resulting in a lower cluster splitting/merging cost.

5. CONCLUSION

In this paper, we have proposed SCS - a semantic overlay network for searching context information in distributed environments. Our simulations show that SCS works effectively. We believe that SCS can make a significant practical impact on building large-scale, schema-based P2P systems. Encouraged by our simulation results, we are currently building a working prototype to demonstrate how SCS can work effectively in real life.

6. 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] T. Gu, H. K. Pung, and D. Q. 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.

[3] 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.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In Proceedings of ACM SIGCOMM, 2001.

[5] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science, 2218:161–172, November 2001.

[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications, 22(1):41–53, January 2004.

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

[8] T. Gu, E. Tan, H. K. Pung, and D. Zhang. A Peer-to-Peer Architecture for Context Lookup. In Proceedings of the International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous), San Diego, California, July 2005.

[9] Min Cai, Martin Frank. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In Proceedings of the 13th International World Wide Web Conference, New York, May 2004.

[10] 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.

[11] A. Crespo and H. Garcia-Molina. Semantic overlay networks for P2P systems. Technical report, Stanford University.

[12] J. Kleinberg. The small-world phenomenon: an algorithm perspective. In Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000.

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