ContextPeers: Scalable Peer-to-Peer Search for Context Information

Tao Gu
School of Computing
National University of Singapore
3 Science Drive 2, Singapore
gutao@comp.nus.edu.sg

Edmond Tan
School of Computing
National University of Singapore
3 Science Drive 2, Singapore
tanyikho@comp.nus.edu.sg

Hung Keng Pung
School of Computing
National University of Singapore
3 Science Drive 2, Singapore
punghk@comp.nus.edu.sg

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

ABSTRACT
Context information has emerged as an important resource to enable autonomy and flexibility of applications. The widespread use of context data necessitates efficient search services in wide-area networks. Centralized context search engines exist, but they suffer from single point of failure and single processing bottleneck problems. In this paper, we propose an efficient distributed context search system based on multiple overlay networks. Nodes that provide similar context data are semantically grouped into an unstructured Peer-to-Peer overlay. Context search requests are only routed to the appropriate overlays, reducing unnecessary query traffic on nodes that have irrelevant context data, and increasing the chances that context data will be found quickly. As the blind flooding mechanism generates a large volume of redundant messages, it will make the system unscalable. To reduce redundant query messages, we propose a Cost-Aware Selective Flooding technique. This technique makes use of 2-hop neighborhood and link cost information to ensure that only necessary messages are being flooded across the network. Our simulation studies demonstrate that our system is scalable and the intra-overlay and inter-overlay routing techniques are effective.

Categories and Subject Descriptors
C.2.2 [**Computer-Communication Networks**]: Network Protocols – *routing protocols;* C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design – *distributed networks, network communications;* C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications.*

General Terms
Algorithms, Routing, Distributing Computing, Performance

Keywords
Peer-to-Peer, Distributed Context Search, Multiple Overlays

1. INTRODUCTION
In recent years, the use of context information has attracted a lot of attention from researchers. Users and applications are often interested in discovering and utilizing various types of context information. 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]. For example, context information may include information on users (name, address, role, etc.), locations (coordinate, temperature, etc.), computational entities (device, network, application, etc.) and activities (scheduled activities, etc.). In previous systems, context data are often described as attribute-value pairs or modeled as Java programming objects. We have proposed an ontology-based model [3] in which context data are represented as Resource Description Framework (RDF) statements. RDF provides a universal platform for representing resources and asserting relations between resources in a machine-readable and machine-understandable way. This context model has the advantage of sharing and reasoning context information among all users, devices and services across multiple application domains. Each RDF statement is represented as a triple of the form <*subject*, *predicate*, *object*>.

Context data is usually stored in a network so that it can be retrieved by a user or an application. The efficient storage and search of context data is indeed a difficult problem in context-aware computing. In our previous work [4], we have built a distributed context repository using Jena [5] which supports both push and pull services. In this paper, we focus our attention on the problem of Peer-to-Peer (P2P) search for context data over wide-area networks. One approach is to use a centralized search engine such as RDFStore [6] to store context data and resolve lookup requests. Although this approach can provide fast responses to a context query, it has limitations such as single points of failure and single processing bottlenecks. A P2P approach seems to provide a potential solution to store distributed context data in the network and facilitate distributed query routing. Decentralized P2P systems have two different architectures: *structured* and *unstructured*. In *structured* models such as Chord [7], CAN [8], and Pastry [9], data placement is tightly controlled based on distributed hash functions. However when dealing with context data, it is more efficient that the data be stored close to a node where it is generated and where it is likely to be used instead. For example, when a person, say John, comes back home, a location sensor detects his presence. It is more reasonable to store his location data <*socam:John socam:locatedIn socam:Home*> in a node at his home rather than a particular node which may be far away based on the hash value of the RDF triple.

Unstructured P2P systems allow nodes to interconnect freely and do not impose any structure on the resources that can be stored. Gnutella [10] is an example of an *unstructured* P2P system that provides a reasonable underlying infrastructure for context lookup systems. Edutella [11] and its successor [12] provide an RDF-based meta-data infrastructure and a routing scheme based on Gnutella-like P2P networks. However, 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 only nodes which are likely to contain the relevant type of context data rather than to flood it to every node in the network. This can potentially lead to a lower network load and better search performance. In this paper we propose the use of multiple overlays to group various context producer nodes. Each overlay "ties" and manages a set of producer nodes with similar categories of context data. By contacting the nodes in one overlay or multiple overlays in parallel, context queries can be resolved quickly.

Studies in [13] and [14] have shown that Gnutella-like P2P networks contribute the largest portion of Internet traffic based on their measurement on some popular P2P systems, such as Gnutella itself. This is because the blind flooding mechanism used in such systems generates a large amount of unnecessary traffic. It also incurs additional processing overhead on each node, causing *unstructured* P2P systems to be far from scalable. Aiming at reducing unnecessary query messages, we propose a Cost-Aware Selective Flooding (CASF) technique to route a query within a Gnutella-like P2P overlay. Upon receiving a query, a node performs the CASF algorithm to select an optimized subset of forwarding paths based on the costs of its neighboring links within 2-hops. Our simulation studies show that the total number of query messages is significantly reduced by CASF and every node in an overlay receives the query.

The rest of the paper is organized as follows. We describe related work in Section 2. We then discuss multiple overlays in Section 3, and discuss the CASF technique 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 and Sesame [15] 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. [16] 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, the overlay maintenance cost is high in this system. In addition, storing each RDF triple multiple times in the network increases the storage cost. Schema-based P2P networks such as Edutella and Piazza [17] 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 [18], 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. The study in [19] uses probabilistic analysis to show that multiple overlays can improve the search performance considerably. Our system is based on *unstructured* P2P overlays and hence does not pose any constraint on data placement. We deploy the concept of multiple overlays in context-aware computing to cluster context producer nodes based on pre-defined schemas. Importantly, we focus on the query routing issues both within and across overlay networks and aim to increase the system scalability by reducing unnecessary query traffic. Many recent efforts have been made to avoid unnecessary traffic incurred by the blind flooding mechanism. Liu et al. [20] proposed a location-aware topology matching technique to solve the mismatching problem between the P2P overlay and the physical network. In their system, each node detects and cuts most of the inefficient and redundant links, and creates new links to its closer neighbors. While this technique is very efficient in reducing overall traffic and improving query performance, the cutting and creation of links for nodes in a global scale may incur a large amount of overhead. In addition, their approach requires that the clocks in all nodes be synchronized. On the other hand, our proposed CASF technique does not creates such overhead and takes away the need for global synchronization.

3. CONTEXTBUS ARCHITECTURE

We use multiple 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 a Gnutella-like overlay which we call a ContextBus. A query will first be preprocessed and mapped to 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. Within a ContextBus, the query is routed to other nodes based on CASF. The overall performance of context search can be improved by forwarding a query only to the nodes which contain the same type of context data as requested in the query.

The meta-data for classifying context data are defined in context ontologies. As context data 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 data into several categories such as *person*, *location*, *activity*, *device and network* and defined the domain ontology for each category in [4].

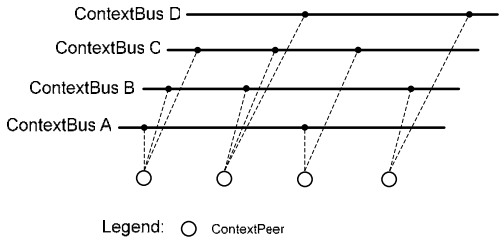

Figure 1. ContextBus architecture

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 a Gnutella-like overlay network. Upon receiving a query, a ContextPeer extracts the semantics from the query and obtains the information about the ContextBus, and then routes the query to appropriate ContextBuses. When the query reaches the designated ContextBus, it is 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 [21].

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). Thus, 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 where nodes participate in. Duplicate entries may exist across different ContextBuses as nodes may join multiple ContextBuses.

When a node, say 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 [20] 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 [22], 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 ContextBus topologies.

3.2 Inter-Overlay 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 for the *low-degree* node that will route the query to the appropriate ContextBuses. The query will then be flooded within a ContextBus using CASF.

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 Gnutella's *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 Gnutella *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. COST-AWARE SELECTIVE FLOODING

In a Gnutella-like overlay network, a query is simply forwarded to all neighbors within a certain network radius. In many cases, such a blind flooding mechanism results in a lot of redundant query messages. The reasons for this problem are twofold. First, a query may be forwarded to multiple paths that are merged to the same node. In this case, only one of the paths is necessary and messages generated along the other paths are redundant. Second, two neighboring nodes may forward the same query message to each other if they have not received the query from each other before.

In this section, we describe the CASF technique which aims to reduce redundant query messages while ensuring search completeness. The basic principle of this technique is to obtain a set of optimized forwarding paths for a node such that unnecessary query messages are avoided. To achieve this, a node executes the CASF algorithm using link cost information within its 2-hop neighborhood. We define a node's 2-hop neighborhood

as all its direct neighbors and their respective neighbors. CASF is subsequently performed by nodes in other relevant neighborhoods to determine a subset of forwarding paths in the whole overlay. CASF operates in three phases: neighborhood link cost measurement, basic routing algorithm, and routing decision mediation.

4.1 Neighborhood Link Cost Measurement

To measure and obtain the link costs of a node to all its direct neighbors, we create a *Link Cost Measurement* message. Each node in the system periodically floods this message to all its direct neighbors. This message contains the source node's IP address and the timestamp at the point this message was flooded. Upon receiving this message, a node *x* can compute the link cost from the source node to itself by computing the difference between the Source Timestamp and the time it receives this message. This message is discarded by each node upon receipt.

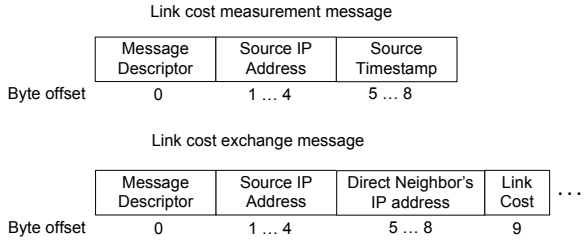

Figure 2. Link cost measurement and exchange messages

To obtain the link costs of a node to all its neighbors' neighbors, we create a *Link Cost Exchange* message. Each node periodically floods this message to all its direct neighbors. This message contains the source node's IP address, all its direct neighbors' IP addresses and the link costs of the source node to all its direct neighbors. When a node *x* receives this message, it will know all the direct neighbors of the source node and their respective link cost information. Node *x* will then store this neighborhood information, which consists of node IP addresses and link costs, in its routing table as shown in Table 1.

Table 1. Node *x*'s Routing Table

ContextBusID	Direct Neighbor, *y*	*y*'s Neighbors
A	<$nodeID_{y1}$, $cost_{y1}$>	<$nodeID_{y11}$, $cost_{y11}$>, …
	<$nodeID_{y2}$, $cost_{y2}$>	<$nodeID_{y21}$, $cost_{y21}$>, …
B	<$nodeID_{y3}$, $cost_{y3}$>	<$nodeID_{y31}$, $cost_{y31}$>, …
	…	…

4.2 Basic Routing Algorithm

The overlay topology within a node's neighborhood can vary in different ways. We recognize that there are three fundamental cases: *3-loop*, *4-loop* and *n-loop*. We define a loop as a group of nodes linked together in a ring fashion. Loops can consist of 3 or more nodes, and may be closed or open. In particular, we will call closed loops with 3 and 4 nodes *3-loop* and *4-loop* respectively. We shall refer to all other types of loops (containing 5 nodes or more, either closed or open) as *n-loop*. We now describe the basic routing algorithm of CASF, which selects optimized forwarding paths with respect to a loop.

The basic routing algorithm follows the least-cost principle, which means that it is desired to forward a query along a set of least-cost paths. A source node, say *x*, is able to detect a *3-loop* if the two direct neighbors of *x* are direct neighbors of each other. For example, in Figure 3, node *x* detects a *3-loop* as B and C (both direct neighbors of *x*) are also direct neighbors of each other. For a *3-loop* with source node *x*, *x* computes and selects a set of optimized paths (shown as solid arrows for various cases in Figure 3) based on the link costs *d1*, *d2* and *d3*. The paths are selected based on a minimum set of least-cost paths to ensure that a query reaches B and C quickly without redundant messages. For example, in Case 1 of Figure 3, a query will only be forwarded along the links *xC* and *CB* as *d1 > d2 + d3*. This is in contrast to the blind flooding mechanism where the same query message will be flooded along the paths *xC*, *CB*, *Bx* and *xB* (shown as dotted arrows), resulting in the messages on *Bx* and *xB* being redundant. As for other cases in Figure 3, CASF avoids two redundant messages as well.

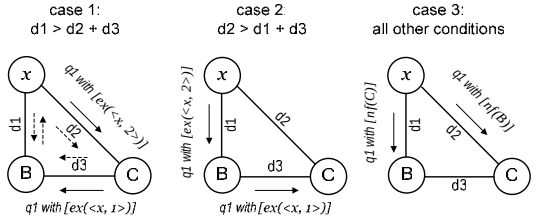

Figure 3. Optimized paths for a *3-loop* with source node *x*

To forward a query along the paths that are determined by CASF, we introduce two lists: *non-forwarding list* (*nf*) and *expected list* (*ex*). The *nf* list contains the node IDs of neighbors which a query should not be forwarded to (we call these neighbors *non-forwarding* nodes). The *ex* list contains the IDs of nodes (which may or may not be neighbors) that are expected to receive the query from another path (we call these neighbors expected nodes). Queries should not be forwarded to the nodes in the *ex* list as well. When receiving a query, each node executes the CASF algorithm and may add new entries into or delete existing entries from the *nf* and *ex* lists which are appended to the query message. The lists will then be subsequently retrieved by the receiver node and used for forwarding decisions. The lifetime of an entry in the *nf* list is always 1 hop, and hence this entry will be purged upon use. Each entry in the *ex* list has a corresponding TTL value which define the number of hops this entry can be used. An entry in the *ex* list will be purged either upon use or when its TTL equals to zero.

For example, as shown in Case 3 of Figure 3, source node *x* computes and knows that both *xB* and *xC* are the optimized links to forward a query (i.e. *q1*) along. Node *x* forwards the list *nf(B)* to node C to inform C not to forward the query to B, and the list *nf(C)* to node B to inform B not to forward the query to C. For the sake of clarity, we only show the entries in the *nf* and *ex* lists which are added with respect to the loop computation initiated at node *x*. The same reason follows for Figure 4 and Figure 5 as well. In Case 1 of Figure 3, node *x* discovers that *xC* and *CB* are the optimized links and decides to forward a query to C only. Source node *x* adds itself to the *ex* list and set its TTL value to 2. When node C receives the query, it executes the algorithm and decreases the TTL value of *x* in the *ex* list by 1. Then node C forwards the query with the list *ex(<x, 1>)* to node C. Node C will not forward the query back to *x* as *x* is the sender. When the query reaches node B, it decreases the TTL value of *x* to zero and removes node *x* from the *ex* list. Hence, node B will not forward the query back to *x* along the path *Bx*.

A *4-loop* with source node *x* is detected if two direct neighbors of *x* share a common direct neighbor. For example, in Figure 4, node *x* detects a *4-loop* as B and C (both direct neighbors of *x*) share a common direct neighbor D. The optimized paths for a *4-loop* with source node *x* and the appropriate *nf* and *ex* lists associated with the query (i.e. *q1*) for different cases are shown with solid arrows in Figure 4. In the case of equality for Cases 1 and 2, since both choices have the same effect, we arbitrarily fix the routing path to that in Case 1. The same reasoning follows for Cases 3 and 4. In all cases, two redundant messages are removed as compared to the blind flooding mechanism. For example, in Case 1, the two redundant messages along the paths *BD* and *DB* (shown as dotted arrows) are removed.

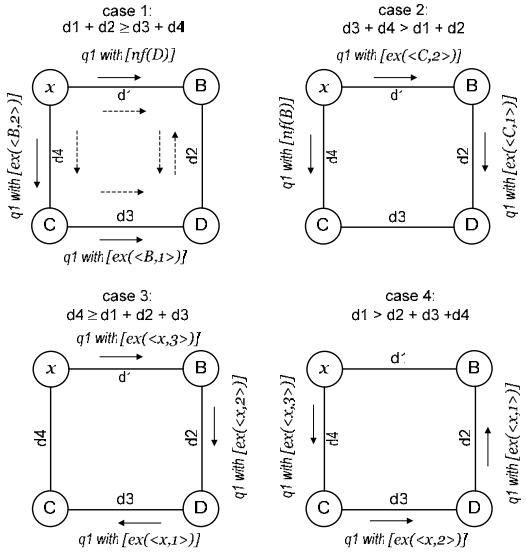

Figure 4. Optimized paths for a *4-loop* with source node *x*

The neighbors of node *x* for which no *3-loop* and *4-loop* are detected are considered to be part of an *n-loop*. Figure 5 shows an example of *n-loop* where *n* = 5. Node *x* detects an *n-loop* as B and C are not direct neighbors of each other and do not share a direct neighbor, and then forwards the query with the appropriate *ex* lists along both paths (*xB* and *xC*). As shown in Figure 5, two redundant messages along the paths *ED* and *DE* removed. The same method follows in the case of *n* = 6. For an *n-loop* where *n* ≥ 7 containing no sub-loops within it, the algorithm will not able to remove redundant messages as we limit the scope of neighborhood information to 2-hops. However, this case seldom occurs in a P2P overlay network. As a result, the TTL value of the entries in case of an *n-loop* is set to 4. We will describe the justification of this choice in Section 4.4.

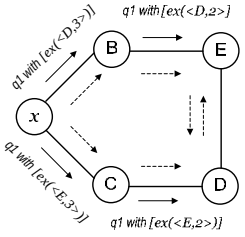

Figure 5. Optimized paths for a *n-loop* where n=5

4.3 Routing Decision Mediation

In the Gnutella overlay topology, a node's neighborhood may contain a *3-loop*, a *4-loop*, an *n-loop* or any combination of these loops. A node, say *x*, will perform loop detection with respect to each of its neighbors. For each loop detected, the basic routing algorithm as we have described in the previous section is performed to obtain a sub-decision for that particular link. These sub-decisions will then be mediated to a final forwarding decision, which would determine whether the query should be forwarded to a particular direct neighbor of *x*.

The routing decision mediation is based on the following two principles:

- Node *x* forwards a query to a direct neighbor if and only if the computation for every loop detected with respect to that neighbor yields a positive forwarding sub-decision to that neighbor.
- If the computation for at least one loop concludes with a negative forwarding sub-decision to that neighbor, node *x* will not forward the query to that neighbor.

To understand the intent behind the decision mediation, it is worthwhile noting that a negative forwarding sub-decision signifies that there is a more favorable alternative path to the destination node than the current path being considered for a particular loop. On the other hand, a positive forwarding sub-decision indicates that the path being considered is the best possible path for a particular loop. Thus, a single negative sub-decision is sufficient to nullify the effect of all other sub-decisions (even if all of them are positive) as it is able to offer a better path than what the other loop computations think is best. As the decision mediation is performed throughout the lifetime of a query flood, the query will thus be flooded along a path that is almost optimized.

4.4 The Main CASF Algorithm

The main CASF algorithm combines both the basic routing algorithm and routing decision mediation. We now present an overview of the main CASF algorithm which is shown in Figure 6. Let *N* represent the set of all direct neighbors of a node, say *x*. Upon receiving a query message *q*, node *x* starts to execute the CASF algorithm. First, it extracts the *nf* and *ex* lists appended to *q*. We denote these lists as the sender's *nf* and *ex* lists respectively. These lists will be used for making decisions. The *ex* list will also be forwarded to all neighbors later. Note that these lists are empty if *q* is initiated at node *x*. Node *x* also decreases the TTL values of all entries in the sender's *ex* list by one. Node *x* than performs loop computation with respect to each of its direct neighbors - *n*, provided *n* is not in the sender's *nf* or *ex* lists. If *n* is found to be in the sender's *nf* or *ex* lists, its entry will be purged. The loop computation involves detecting a particular loop, updating the *nf* and *ex* lists for *n* and setting the variable *forwarding_decision* which represents the forwarding decision along the link from *x* to *n*. If *x*'s sender is a member of a particular loop and the link from the sender to *x* is part of that loop, no loop computation is necessary. This is because the result of the loop computation done previously by the sender has determined the forwarding decision for that loop. Note that *forwarding_decision* is a variable shared by all the loop computations with respect to *n*. It will be set to *true* if all the loops desire to forward *q* along the link from *x* to *n*; or *false* when at least one loop desires not to do so. This is where routing decision mediation takes place. Finally, the loop computation for each direct neighbor *n* of *x* concludes with a

forwarding decision that determines whether *q* should be forwarded to *n*. All direct neighbors that node *x* decides to forward to will be placed in a list known as the *forwarding_list* with their respective *nf* and *ex* lists.

After loop computation has been done for all direct neighbors of node *x*, node *x* will begin post-processing which consists of the combination of *nf* and *ex* lists and the cleaning up of the *ex* list. For those direct neighbors of *x* in the *forwarding_list*, node *x* will combine their *ex* and *nf* lists with those of the sender. Subsequently all the entries with zero TTL values in the *ex* list will be purged. Those *n-loop* entries of node *n* which are not in the *forwarding_list* will also be removed since they will not be used. When all the entries in the *forwarding_list* have been finalized, the query message will be forwarded to all direct neighbors of node *x* in the *forwarding_list* together with their corresponding *nf* and *ex* lists.

```
node x obtains sender's nf and ex from the query message;
decrease TTL of all entries in sender's ex by 1;

/*node x computes for each direct neighbor n */
for each n ∈ N do
  if (n ∈ sender's nf) then purge n from sender's nf;
  else if (n ∈ sender's ex) then purge n from sender's ex;
  else
    forwarding_decision ← true; //the forwarding decision
                                  from x to n
    for each direct neighbor m of n do
      perform loop detection;
      update nf and ex lists for n based on link-cost;
      update forwarding_decision;
      if (forwarding_decision == false) then
        break;
    endfor
    if (forwarding_decision == true) then
      add n, nf, ex to forwarding_list; //keep nf and ex
                                          for each n
    endif
  endif
endfor
end
```

Figure 6. The main CASF algorithm

CASF is a distributed algorithm and does not require a global view of the entire network. Instead, each node has a limited view of the network within the scope of its neighborhood. This gives rise to the possibility of redundancy when *n-loop*s with 7 nodes or more (with no sub-loops within) exist, as a result of limited neighborhood information. In order to reduce this possibility, the scope of neighborhood information needs to be increased. However, this would incur greater computation complexity and storage overhead on each peer. Therefore, there is a tradeoff between the scope of neighborhood information and the degree of redundancy reduction. An overlay topology with a large number of *n-loop*s which have no smaller sub-loops contained within them can only exist when many nodes have only 2 neighbors each (i.e. node degree = 2). However, the study in [13] showed that P2P overlay topologies follow the small world property which has large clustering coefficients (i.e., average of fraction of edges connecting neighbors of a node) and short average path lengths between two nodes. In fact, the node degree of a real Gnutella overlay lies approximately between 4 and 20 based on their observations. Hence, a large number of *n-loop*s without smaller sub-loops contained within them is unlikely to happen. Our simulation studies show that maintaining neighborhood information within a scope of 2 hops provides a good tradeoff between overhead cost and performance, owing to the fact that a large number of *n-loop*s in a Gnutella-like topology is rare.

5. EVALUATION

In this section, we use simulations to evaluate the effectiveness of both ContextBus and CASF, and compare their 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

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. A *high-degree* node helps a *low-degree* node forward its query to a particular ContextBus where the *low-degree* node does not participate in. In our simulation, we assign degrees to nodes based on the power-law distribution as the study in [23] has shown that Gnutella-like networks follow the power-law property. We have 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. The link cost between two nodes in the overlay is calculated based on the shortest physical path between these two nodes.

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. Context data are classified into a set of categories and each category is associated with an ID (*ContextBusID*). 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.

To measure the effectiveness of ContextBus and CASF, we use the following performance metrics:

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

Number of messages per query: the number of query messages generated when executing a search request in the network. We aim to minimize the number of messages forwarded by each node.

Search completeness: the ratio of the number of nodes contacted per query to the total number of nodes in a particular ContextBus which has the same *ContextBusID* as in the query. The value of this metric lies in the range 0 to 1.

5.2 Effectiveness of ContextBus

In reality, a context search in the form of an RDF query always maps to one particular ContextBus. For example, a single context query with an RDF triple pattern <socam:*John socam:locatedIn ?x*> will be pre-processed and mapped to the *Location* ContextBus. Complex queries can be done by issuing multiple single search requests to respective ContextBuses. The efficiency of a search request is achieved by contacting only a fraction of nodes rather than all the nodes in the system. This fraction 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.

For all our simulations, we use BRITE [24] to generate physical topologies with 20,000 nodes. BRITE is a topology tool that provides the option of the AS model to generate topologies. The overlay topologies (ContextBuses) are generated by varying the number of nodes from 500 to 8,000. The minimum node degree (*min_deg*) is set to 4 and the maximum node degree (*max_deg*) is set to 64.

In the first experiment, we examine the effectiveness of ContextBus. 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 2. From the table, notice that for a complete search request, Gnutella has to contact every node in the overlay. In contrast, 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 per query. With less nodes contacted by ContextBus, the network traffic load incurred by a query will also be reduced. Notice that if $\overline{C}$ = C_{max}, then we have to contact every node in the system. When designing a ContextBus system, if a ContextBus becomes popular (i.e. a large number of nodes join this ContextBus), this ContextBus should be split into several ContextBuses to increase C_{max}. This leads to a reduction in network load as the ratio $\overline{C}$ / C_{max} is decreased.

Table 2. 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%

We now compare the average number of messages generated per query incurred by ContextBus against that of the Gnutella protocol as shown in Figure 7. We set the number of neighbors per node to 4 and 16 respectively and set $\overline{C}$ to 4 and 12 respectively. The results show that ContextBus reduces the number of messages per query by 66% when $\overline{C}$ = 12 and by 85% when $\overline{C}$ = 4. From the experiment, we also notice that decreasing $\overline{C}$ will reduce the number of messages per query as expected.

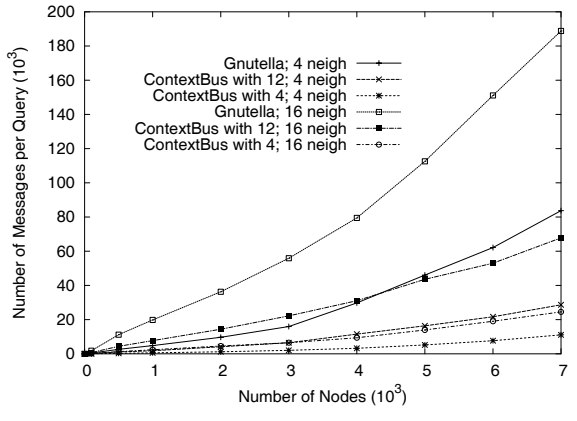

Figure 7. The effectiveness of ContextBus

5.3 Effectiveness of CASF

In this experiment, we evaluate CASF by issuing a complete search request to a particular overlay. We compare the number of query messages incurred by the Gnutella protocol and CASF. We only present the results based on the overlay with 4000 nodes. Figure 8 shows that CASF reduces the average number of query messages significantly by about 60% as compared to the Gnutella protocol. The search completeness when using CASF equals to 1.

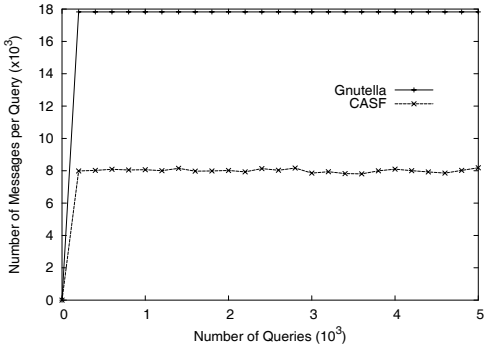

Figure 8. The effectiveness of the CASF algorithm

This experiment also verifies that CASF guarantees to reach every node in a particular overlay for a given search request. The query response time by using CASF is almost the same as the time taken by using the standard Gnutella protocol. Notice that we only use one overlay in this experiment. By using ContextBus, the number of messages per query will be further reduced as fewer nodes will be contacted.

The overhead of the CASF technique falls into two aspects - processing overhead and traffic overhead. The processing overhead is trivial as compared to the Gnutella protocol and we omit the result due to space constraints. We only present the result of traffic overhead. CASF uses two additional messages described in Section 4.1 and adds *nf* and *ex* lists to the original Gnutella query message. In this experiment, we measure the network traffic incurred by Gnutella and CASF respectively in terms of number of bytes generated per second. We set the number of neighbors to 4. Figure 9 shows that CASF consumes less bandwidth compared to Gnutella. The two messages we created only generated about 3.5% of the total traffic. The bandwidth consumed by query messages is also reduced as expected. Our results show that the additional overhead introduced by CASF only constitutes a small percentage of the total network traffic.

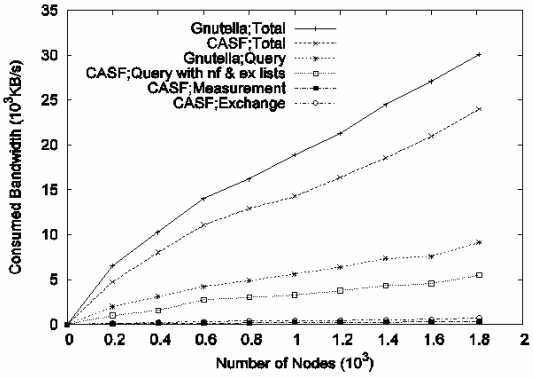

Figure 9. Bandwidth consumption

6. CONCLUSION

We have proposed the use of multiple overlays (ContextBuses) to cluster 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. We have also proposed the CASF algorithm to minimize unnecessary query traffic. Our simulations show that these schemes significantly reduce unnecessary query messages while ensuring search completeness. It should be noted that CASF can be applied to any flooding-based P2P systems to increase scalability.

The approach of multiple overlays works well when the dimensionality of the system is reasonably low. However, when the number of ContextBuses increases, peers may need to participate in a large number of ContextBuses and hence the overlay maintenance cost becomes more expensive. In addition, as the ratio of *low-degree* nodes to *high-degree* nodes increases, a processing bottleneck may exist at the *high-degree* nodes, hence decreasing search efficiency. We are working on how to introduce certain structures on the overlay network to minimize the overhead incurred by high-dimensional overlays and to overcome the inefficiency of the flooding-based routing. We are also building a prototype system to deploy our proposed techniques in real 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] T. Gu, X. Wang, H. K. Pung, and D. Zhang. An Ontology-based Context Model in Intelligent Environments. In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference, pp. 270-275, San Diego, California, January 2004.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] S. Sen and J. Wang. Analyzing Peer-to-Peer Traffic across Large Networks. In Proceedings of ACM SIGCOMM Internet Measurement Workshop, 2002.

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

[16] 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.

[17] 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.

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

[19] E. Cohen, A. Fiat, and H. Kaplan. A Case for Associative Peer to Peer Overlays. ACM SIGCOMM Computer Communication Review, Vol.33, Issue 1, pp. 95–100, January 2003.

[20] 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.

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

[22] 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.

[23] S. Saroiu, K. P.Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An Analysis of Internet Content Delivery Systems. In Proceedings of the 5th Symposium on operating Systems Design and Implementation, 2002.

[24] BRITE, http://www.cs.bu.edu/brite