Uğur Çetintemel 05/14/2003

Replica Location in Sand

by

George Kong

Brown University, 2003

Thesis

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

ACKNOWLEDGEMENT

I would like to thank Professor Ugur Cetintemel for his guidance and inspiration throughout the year. My project would not be possible without his inspiration both inside and outside of class. I would also like to thank Yanif Ahmad for working with me this year. All those late night debugging sessions really helped move this project along. .

SAND Replica Placement

Abstract

Replication is a common way for large-scale distributed systems to balance server loads and create tolerance for server crashes. A replica situated in a server closer to the client can also respond to a query in a more timely fashion. In an overlay network where a set of servers work together to provide more advanced in-network processing capabilities than the normal TCP/IP routing, replication placement algorithms can play a more significant role in promoting efficient utilization of system resources.

In the world of database queries, persistent queries are the ones that make repeated requests for the same set of data object, potentially with a predefined frequency. For these queries, it may be to the network and the client's advantage to optimize the query response process to better utilize systems resources. At the same time, it is also possible for the client to make certain Quality of Service (QoS) demands upon the network.

We introduce the Directed Path algorithm (DPA), a distributed dynamic replica placement algorithm for distributed systems. It is designed to take advantage of the routing and in-network processing capabilities of overlay networks to satisfy QoS constraints of clients who are making persistent queries. At the same time, this algorithm also attempts to minimize system resources used in meeting these demands.

TABLE OF CONTENTS

1. Introduction

Database queries in distributed database system are generally performed with minimal network-level optimization. Their spontaneous nature makes any efforts in the optimization of one query a difficult problem. However, since persistent queries are performed multiple times and have known behaviors, there is an incentive for both the network administrator and the client to optimize the process. SAND attempts to achieve this goal by constructing a process tree from the query operations and strategically placing some operations in intermediate nodes to reduce the amount of traffic in the network. It also takes advantage of multiple client demands for the same sets of data and attempts to satisfy several of them with one calculation.

The other component of SAND is its focus on satisfying the QoS of clients. Network quality assurance is becoming a valuable commodity on the market. We assume that each client has a utility function which correlates the utility of the client in relation to the price set by the network administrator. The system would assign its resources towards satisfying the clients that can benefit the most from it or pay the most for it.

The placement of replicas is directly related to one of the most important commodities in the system – it determines the response time to client queries. Replica placement is a tug of war between the cost to the system and the potential benefit to the clients. Having a lot of replicas located closer to the clients creates better response time and more load-sharing, but it generates more storage requirements and more consistency issues for the system. The other extreme would reduce system resource usage but would also cause tremendous delay for user queries.

Going directly to a more sophisticated utility function would be very difficult on a nascent system. At this point, we are defining the latency utility function of a client as a threshold function. A client would have a utility of 1 if the response time is smaller than the constraint and 0 if the response time is larger. DPA attempts to minimize the number of replicas required to meet the latency constraints of clients in the system by placing replicas in servers that would maximize replicas sharing between clients.

Within this project, I have worked mainly on making Tapestry compatible with SAND. This includes exposing some internal Tapestry functionalities to SAND. SAND needed the routing table for the Directed Path Algorithm. In order to enable testing of the system on an artificially generated network topology, I wrote a method to allow us to set the latency between nodes in a node's routing table. I also helped in devising the Directed Path Algorithm as well as the centralized greedy algorithm as the comparison basis for DPA. I also worked on integrating the different parts of the system to create a batch experiment for SAND.

The rest of the paper describes in detail how the algorithm attempts to achieve this feat. Section 2 describes Tapestry, the overlay system that SAND is built upon. Section 3 describes DPA as well as a global near-optimal algorithm that we use as a comparison for DPA results. This section also details some of the theoretical bounds realized by the two algorithms. Section 4 presents the implementation details of the algorithm. Section 5 shows some preliminary results from two experiments. Finally, Section 6 attempts to look at the current state of the project and looks ahead at future works that needs to be done and Section 7 describes related works.

2. Tapestry

In recent years, there has been a push by larger-scale content providers to use overlay networks to facilitate data distribution. These networks, composed of a group of dedicated servers, perform more advanced in-network processing to improve routing and scalability. We used Tapestry[1], an overlay network created in UC Berkeley, to enumerate a search for SAND, where we have direct access and control of the routing information. Tapestry performs location-independent routing that can route a message to the nearest location of a service or search for the nearest copy of a data object in the network.

Tapestry is an overlay network infrastructure that was originally designed to be the foundation of OceanStore[2], a global persistent data storage system that can build on top of untrusted servers. It is now a more independent project that is being used by several other groups as the foundation for their distributed projects.

The main advantage to using Tapestry is its ability to locate and route messages to the nearest copy of a data object in a deterministic manner. Each Tapestry node in the network has a Global Unique ID (GUID) that distinguishes it from all other nodes. Using a routing mechanism introduced by Plaxton, Rajamaran, and Richa[3], Tapestry routes the message through a series of intermediate nodes that makes independent decisions on what the next hop is on the path without knowledge of the rest of the path. Using dynamically updated routing tables containing neighbors at each node, Tapestry routes the message from the sender to the target node by incrementally comparing and matching the digits in the node's GUIDs. For example, routing from a node x with GUID 3952 to a node y with a GUID of 4724 would proceed as follows. x would find a node that ends in 4 in its neighbor table, forward it to that node, k_1. k_1 finds a neighbor, k_2, ending in 24, which then goes to its neighbor ending in 724, which would then route to y. If at any node k_i in the path, there is no exact match found for the next digit, k_i does an approximation to determine the next node in the path. Given a GUID of length m where each digit can be of any of n characters, a Tapestry network can have up to n^m different nodes. Routing from one node to another takes at most m hops on the overlay network, which is logarithmic to the number of nodes in the system. The space required to store the routing table for each node is $O(nm)$ which is also log to the size of the system. This log routing and storage bound allows the system to grow without incurring too large of a penalty on storage and latency.

When a node that stores a data object, its publisher publishes the object, it hashes the object to find its GUID, which can be matched to one of the nodes in the system. Let's call this the root node of this data object. If there is no exact match, the same approximation algorithm used for routing is used to determine the root amongst the existing nodes. The publisher notifies the root by sending a publish message to it. Each node on the path keeps a back pointer to the publisher for this object as it passes the message along.

When a client performs a locate operation on this object, it follows its own path to the root because the object GUID is matched with the root's. When the message reaches the root, it would automatically redirect it to the publisher of the data object. In most instances, the locate message doesn't have to reach the root to find what it's looking for. If the message encounters any of the nodes that contain a back pointer to the publisher of that data object, the locate is also automatically redirected to the publisher.

The latency stretch of an overlay network is the average ratio of the routing on the overlay network and the underlying IP network. Tapestry guarantees a theoretical stretch bound[4] of $O(|ID|log^2 n)$ where ID is the size of the GUID of the system and n is the number of nodes in the system.

3. Algorithm

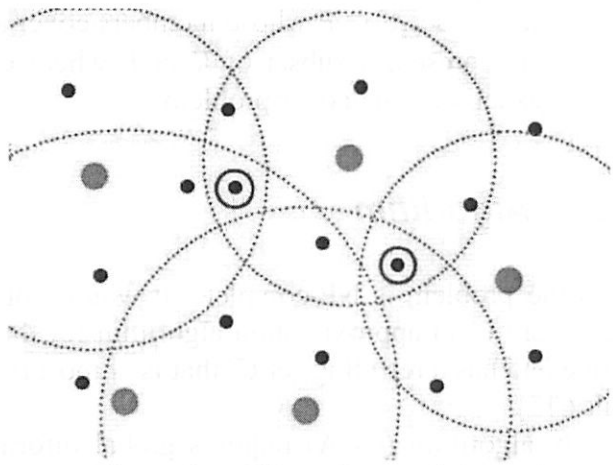


Figure 1: Client Constraint Graph

Before we talk about the Directed Path Algorithm, we need to explain in detail the problem we are trying to solve. When a client makes a query that requires the processing of several data objects in SAND, a request is sent out to retrieve these data objects. Let the latency of retrieving data object *i* be $lat(O_i)$. Then the latency of the entire operation:

$$lat(Q) \geq max(l(O_1), l(O_2), l(O_3) \ldots l(O_n))$$

where O_1 through O_n are all of the objects required for this query. Thus, in order to satisfy ε, the client's latency demand for this query, there needs to be a replica that can be reached in time $\varepsilon/2$ for every object required to answer the query. Graphically, we can draw an enclosure around a client on the network that separates the space that meets its constraint requirement and the space that do not. Drawing such constraint boundaries for each client in the network would give a graph that looks like Figure 1. Note that this graph represents the constraint circles of one particular data object that are needed by the clients. Thus, there are potentially other client constraint graphs that need to be satisfied for every client. The black (smaller) dots represent servers in the network. The blue (bigger) dots are the clients, which are the centers of their respective blue (larger) latency constraint circles. Ideally, we want to select the fewest number of replicas so that they will cover all of the clients. Graphically, a server can cover a client if it lies within the constraint circle of the client. For Figure 1, the two black dots enclosed by the red circles are the ideal replica placement candidates to store the objects. If a copy of each of the data is placed in both of the servers, then they would be able to cover all five clients.

3.1 Theoretical Framework

In order to determine how well DPA performs in real systems, it is necessary to compare it to some other method. We chose to compare DPA to the optimal result. This problem of trying to find the minimum number of servers that can satisfactorily cover all of the clients in a

network is an NP-complete problem. It can easily be translated into a well-known NP-complete problem, the set-cover problem.

An instance (X, F) of the set-covering problem is composed of a finite set X and a family of F of subsets of X, such that every element of X belongs to at least one subset in F. And the problem is to find a minimum-size subset $C \subseteq F$ whose members cover all of X. If we let the set of clients be X, and each server represent a subset of X in F where each subset contains the clients the node can cover, it is exactly the set cover problem.

3.2 Centralized Greedy Algorithm

Since the solution to the problem is NP-complete, it would not be efficient to find the optimal solution. However, there is an approximation algorithm for the set-cover problem that only requires polynomial time and has a resulting set C' that is at most $(ln|X|+1)$ times more than the size of the optimal result C[5].

The centralized greedy algorithm (CGA) requires global information of the system to calculate the set of servers that would cover all of the clients. Given all of the sets F_i of servers where each element in the set represents the clients the server i can cover, the algorithm proceeds as follows. First, it picks the set, F_k in F with the highest cardinality, removes it from the set, and marks it as a candidate for C'. If there is more than one set with the same highest cardinality, we arbitrarily break the ties. Then, for every element E_j in F_k, we remove E_j from the sets that are still in F. After removing these elements, we once again pick the set with the highest cardinality from the remainder, and repeat this pick and removal process until either all of the set are picked or all of the sets in the remainder of F are empty. C' consists of all of the nodes that are associated with the picked sets.

In such a system, we would need a central decision maker that would gather information from all of the servers and execute the commands. So the decision maker would need to flood the system, and wait for replies from all the servers every time there is a change in the client population.

In the CGA, clients would perform a constrained flooding, allowing the servers to build up information on what clients it can potentially serve. The constrained flooding would be described in more detail in the next section. The decision making node waits for a specified amount of time, and then executes the greedy set cover approximation to calculate the set of servers that can be used as clients.

3.3 Directed Path Algorithm

Now, we introduce the Directed Path Algorithm. This algorithm has the following major phrases of operation:

1) *Initial flooding/request:* It does a constrained flooding to inform all servers within range of the data objects that it's looking for. If a server within this latency range contains a replica of a requested data object, it replies to the client.

2) *Replica site search:* For each data object that does not have a replica in range, the client issues a locate message for that data object. This request is routed to the nearest server on

the overlay network containing a copy of the object. At this point, the server issues a directed search algorithm to try to find a suitable replica site to place the new copy of the replica.

3) *Replica relocation/creation:* Once a new site is found, the chosen site contacts other existing clients that it could cover and see if it is possible to move the replica site for these clients to the new location without losing coverage to any of them.

3.3.1 Initial Flooding/Request

In the first step, the new client starts a flood message to every one of its neighbors with the latency that it is aiming for and the GUID of the data objects it needs. After every hop, the message automatically decreases the latency value in accordance to the time it took for the hop, and forwards itself to all neighbors of the server it is on. When this latency goes below zero, we stop forwarding the message. A server that receives this message checks the data replicas it is currently storing and checks for any matches with the list of GUIDS the new client is looking for. If a match is found, a found message with the object GUID is sent back to the client.

The client waits for a time period that is a factor of its latency constraint, and then wakes up to check all the found messages that it receives. If the all data objects it needs are present, then the algorithm is finished. Otherwise, it sends out a slightly modified version of the locate message. Once this message locates the nearest node with a copy of the data, it triggers a directed replica site research operation.

3.3.2 Replica Site Search

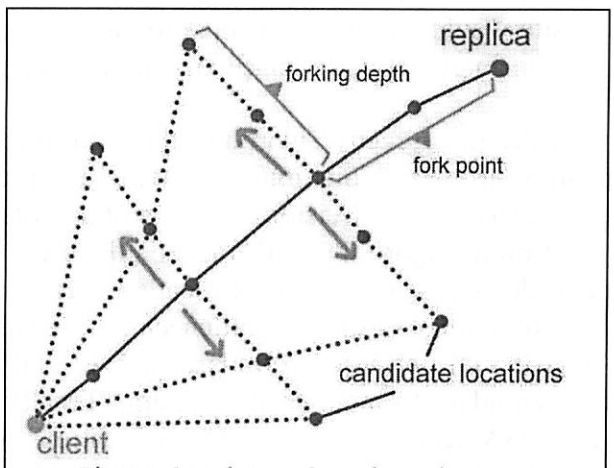


Figure 2: Directed Repica Site Search

This search takes advantage of the low stretch factor of Tapestry and uses this information to direct replica location searches in a direction that has a higher probability of finding a more suitable replication site. Since overlay routing has a low stretch, it is very likely that most of the path that routes a message from one node to another is going to be topologically located between the two nodes. We assert that there is a much higher chance of finding a good replica site candidate in the group of servers that lie near the path than from other servers. The algorithm specifies a starting point, server *i,* on the path where it starts forking off the path. Since forking off from the beginning will hit a lot of the servers that are outside the constraint

region of the client, it would be more efficient to do so after a certain number of digits on the GUID are match. On the path to the client, if a server GUID has x number of digits similar to the client's, it will look at level x (the level with x number of digit similarities) in its routing table, and fork out to those nodes, indicating that the client is searching for a possible replica site for relocation or creation. As we move down the graph, the neighbors that we choose strays further away from the path. When only a few digits are similar between the destination GUID and a node's GUID, there are much more There is also a parameter that specifies the number of hops that the algorithm should move off the path as well as a parameter denoting the number of fan-outs per hop. These two parameters determine the size of the candidate set that would reply to the client. When a client performs the bounded flooding during its initial phrase, all the servers within its latency range save the client GUID as a client that it can possibly cover. Over time, every server would have a set of clients that it can potentially cover. When a server receives a replica site search message, it sends its client coverage set to the new client requesting the replica relocation. Once the client receives replies from the candidates, it finds the candidate that covers the most number of clients and performs the relocation/creation at that point.

3.3.2 Replica Relocation/Creation

When a server is informed that it has been chosen as the site for a new replica, it first attempts to relocate a replica from another location to minimize the number of replicas used. It makes a check to see if moving a replica that covers one of the clients in its existing coverage set to the new location would drop coverage for a client. Algorithmically, this means that for every replica server that is currently covering a client in the new site's coverage set, we check if the new site's client coverage set is a super set of the existing replica server. If it is, then we know that relocating to the new site would maintain coverage for all existing clients as well as the new client. If this check fails, then a new replica is placed in the site, and the root of the data object is informed of this addition.

Compared to the centralized greedy algorithm, there are three factors that make DPA a more suitable algorithm in a large-scale distributed system. First, the greedy algorithm needs complete information in order to make a decision. A crash of any of the servers would cause the central decision maker fault on its algorithm. DPA acts on the servers' knowledge of their local network topology to make decisions about replica location. This makes the DPA more fault tolerant than the CGA. Second, the CGA has a bottleneck that is the central decision maker. If that node goes down, then the entire system grinds to a halt. DPA, on the other hand, is a distributed algorithm that can adapt to faults in the system. Finally, every time a client enters and leaves the system, the CGA needs to recompute new locations for every replica. Although not all replicas may need to move every time the client set population changes, it is definitely possible in the worst-case scenarios. The constant calculation and shifting of replica locations is costly to the system. DPA on the other hand, either shifts a replica or creates a new one if it is needed, incurring much less incremental costs as the system grows.

4. Implementation

The experiment is performed on custom-built machines with Athlon 1800 XP processors running at 1.3 GHz. Each machine has 512 MB of RAM and is running a custom version of

Debian Linux on top of Linx Kernel 2418. They are connected via a 100 Mb/sec Ethernet LAN. Before getting into the experimental details, we will describe the implementation of the SAND system.

As mentioned before, SAND is built on top of Tapestry, which is written in Java, version 1.3.1. The code is compiled using IBM's Jikes compiler over IBM's Java libraries. We are using an unofficial release of Tapestry that has fixed several bugs and improved functionalities over Tapestry 1.0. Tapestry uses SEDA[6], an eventdriven concurrent system architecture that separates different components in the system into different stages. Tapestry has a Network Stage that handles socket level networking, a routing stage that implements the Plaxton routing algorithm, and an application stage for application using Tapestry. Information between stages is passed via messages. This makes Tapestry easily expandable, but it also leaves a considerable footprint in the memory. The SAND portion of the implementation is implemented in OCaml following the event driven message passing model. We use C and JNI to negotiate the interface between Java and OCaml.

Most of the functionalities of SAND were added in a separate layer above Tapestry. SAND is designed to be a scalable distributed database that could be run on any DHT overlay network. In this case, Tapestry just happens to be the DHT implementation we chose. We can just as well use Chord, Pastry, or CAN as the underlying DHT structure. Although adding functionalities in one of the Tapestry layers may expedite processing of requests, we separated the functionalities of Tapestry and CAN.

4.1 Tapestry Modifications

To run SAND on top of Tapestry and to conduct our experiments, several modifications were made in Tapestry.

First of all, more message primitives were needed to perform all of the functions that SAND requires in Tapestry. The handlers for most of these message primitives reside in the routing layer. They are placed in that level to reduce the response time to some of the more frequently used message types. The delay from placing them at the SAND level would be higher since handlers are applied to messages under a chain of responsibility design pattern.

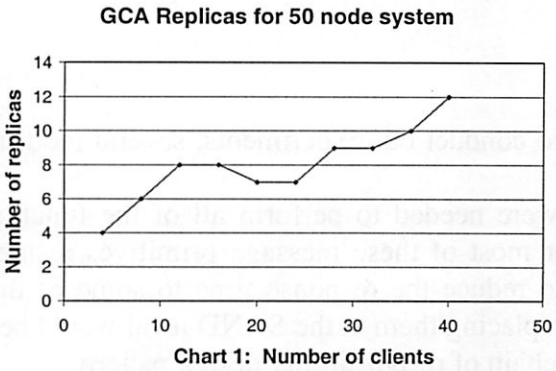
New primitives were implemented to allow the topologically directed replica site search operation to run. A bounded flooding message was added into Tapestry for the first DPA phrase. This is accomplished by maintaining a latency value within the message which is decremented according to the routing table latencies as it hopped form one node to another. To eliminate sending repeated flooding messages, a fixed-length queue maintains the most recently seen message GUIDs. When a new message is received, its message GUID is checked against the GUIDs in this queue. Processing is only performed only if the GUID is not in the queue. The queue is a FIFO queue, so the oldest message GUIDs will be the first ones remove from the queue. In a concurrent system where multiple events are occurring, it is advantageous to keep the flooding message GUIDs around a little longer so that later messages that flood back would not be processed again. Therefore, we reinsert repeated GUIDs at the head of the queue even though we don't process the message.

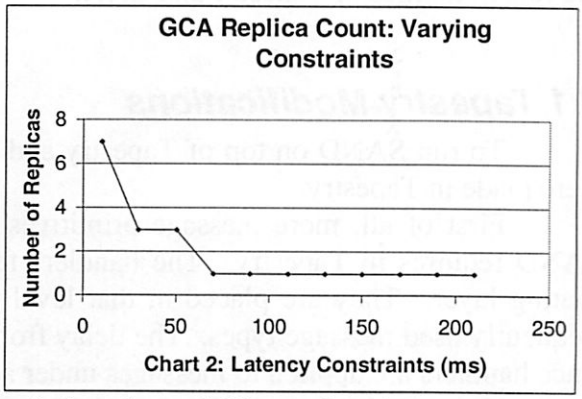
Since the experiment is performed on only a few physical machines, multiple Tapestry nodes have to be started on the same machine. The latencies between these nodes are 0, while the latencies between nodes residing on different machines connected by LAN are fairly identical. This makes experimental results rather meaningless. Therefore, we implemented a

way to infiltrate the routing layer of Tapestry so we can manipulate the latency between nodes from the SAND layer and impose our own underlying physical network topology. We plan to run experiments in a network topology generated randomly as well as one conforming to power-law, which more closely resembles the structure of the Internet. When conducting an experiment for k servers, a set of k random numbers are generated and each is assigned as the ID to one of the SAND nodes started. After the nodes start up, we reassign the latencies of a node's neighbors in its routing table to the artificially generated values and lock them down to make the network static. Although this defeats the purpose of Tapestry's dynamic routing updates, it is currently the best way to perform meaningful experiments on a set of LAN-connected machines.

To test the algorithms, we first start a set of servers on the machines. Then, we publish a single object, and then begin testing the two algorithms. To coordinate the different actions in the experiment, we use several perl scripts to set up the timing between the different stages of the experiment. Tapestry nodes spawning are fairly time-intensive and client operations takes time to propagate through the network. Taking into account the events of spurious node failures, the timing of the different processes are time triggered. For example, instead of waiting for all of the tapestry nodes to be activated before we start the clients, we wait a specified amount of time so that random node failures would not hinder that batch of the experiment.

5. Results





Most of the implementation and ground work has been laid out for the experiment. Several preliminary experiments were performed. Two sets of experiment were performed. In the first one, we fixed the number of servers to 50, fixed the constraints at 20 milliseconds, and varied the number of clients from 4 to 40 in intervals of 4. As expected, the number of replicas increased as the number of clients increased. Note that the first data point with 4 clients required 4 replicas because these clients are sufficiently apart that none of them had constrained regions that intersected. Thus, each required its own copy of the data object. However, as the number of clients increased, more clients are sharing the data replicas. The ratio between the number of clients and the number of replicas went from 1 to 10/3. At 40 clients, more than 3 clients shared a replica on average.

In the second experiment, we fixed the number of servers at 50 and the number of clients at 10, and tested the effects of varying latency constraint values on the number of replicas created. Constraint was varied from 10 milliseconds to 210 milliseconds in intervals of 20 milliseconds. The result was the expected decrease in the number of replicas required as constraint went up. When the constraint went up to 70 ms, only 1 replica is needed to satisfy the demands all of the clients.

6. Conclusion/Future Works

We presented the Directed Path Algorithm, a replica placement algorithm for SAND. In a dynamically changing topology where clients have different latency constraints, DPA attempts to minimize the number of replicas created to serve the clients. Initially built on top of Tapestry, DPA is a geographical aware algorithm designed to work with any DHT overlay network to localize replica placement searches in a limited region. We believe that this approach have a high probability of finding a near-optimal solution while lowering the network cost for new clients that join the system.

In this project, I worked on the interface between Tapestry and SAND. I was responsible for creating messages and message handlers to allow SAND to gain access to some of Tapestry's internal data structure, to gain access to hidden Tapestry functions, and to create new functionalities in Tapestry. These include exposing the Tapestry node's routing table, writing the bounded flooding protocol, and allowing SAND to set latency values in the Tapestry node's routing table. I also helped design the Directed Path Algorithm as well as the centralized greedy algorithm, and worked on designing and implementing the experiments to test our algorithm.

There are several other sets of experiments we can run to verify the effectiveness of DPA. One of the most important results we are looking for is the number of replicas created to satisfy the clients. Another system resource we are using is network bandwidth. A good indication of network usage is the number of messages transmitted. Before sending each network message, we write a log entry with a predefined format into a log file. At the end of the experiment, we count the number of such log entries to get a count of the message complexity of the operations. Another figure we should look at is the response time of the system in satisfying the join request of a client. We can run some of the experiments described below to obtain meaningful data for the algorithm.

First, fixing the number of servers, client latency constraints, and the number of clients, we can randomly choose client and server placements to compare the number of replicas created by DPA and GCA. This will give us some data points to perform statistical analysis on the range of results that are likely to occur and variance on the results.

Varying the number of clients in the system is another experiment we could perform. It should have a positive correlation with the number of replicas used. Varying the number of servers would be a very interesting experiment to perform. Intuitively, having more servers would increase the chance of a server being in the intersection of clients' constrained regions, and decrease the number of replicas we will need to sustain the same number of clients.

Experiments with 50 to 100 nodes are fairly small networks and would not test the scalability of the system to see how well it responds to heavy loads and highly concurrent messages. Therefore, running experiments with more nodes can tell us more information about whether the operations are being localized and how well the algorithm can respond to multiple concurrent joins. We plan to test our system in Emulab and Planetlab where we can start up a larger system.

Although not presented here, we need to implement the removal of clients from the network. If this removal is voluntary, then the client would inform all the nodes within its constraint region about its intention to leave, then exit the network. The servers would remove the client from its client coverage set. If the replica that was serving this client has no other clients to serve, then we can remove the replica from that server.

Currently, this system does not place any consideration on fault tolerance and load balancing. If there is only one replica server that covers a client, the server's crash would cause the client to malfunction. Therefore, it would be beneficial to have multiple servers with the replica within any client's constraint region.

7. Related Works

Our work on SAND is an extension on the progress made by overlay networks in pushing computation into the network. Active Network researches [7, 8] propose that these computations should be done on the packet level, while most other software overlay networks proposes a solution that moves the processing to an overlay layer which controls the network routing at the application layer. Networks such as CAN, Tapestry, Pastry, and Chord [1, 9, 10] can provide large Content Deliver Networks such as Akamai [11] and Digital Island [12] with more effective ways to store their data.

SAND utilizes the capability of the overlay network to improve the quality of continuous query in database systems. Previous work in this area [13, 14, 15] have neglected scalability of the system and QoS concerns of clients. While trying to optimize system performance, our system places is also constrained by the QoS demands proposed by the clients. Therefore, SAND's replication scheme have a two-fold purpose: satisfying client QoS demands as well as minimizing system resources being used in achieving the first goal.

Many previous replica placement schemes [16] view client latency assumes some knowledge of the network topology and places replicas according to this knowledge. We propose a system that directs the replica placement search in a topological area that is most likely to minimize unnecessary node creation while satisfying QoS requirements of all clients. Our ability to control the fan-out and forking depth of the algorithm also enables it to decide whether to sacrifice higher network usage to find a better candidate or to settle for a less stringent search with lower network usage.

Reference

1. Ben Y. Zhao, John Kubiatowicz and Anthony Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing.*

2. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. *OceanStore: An Architecture for Global-Scale Persistent Storage.* In Proceedings of the 9th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

3. C.G. Plaxton, R. Rajaraman and A.W. Richa, *Accessing Nearby Copies of Replicated Objects in a Distributed Environment.* In Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97), pp 311-320, Newport, RI, USA, 1997

4. Kirsten Hildrum, John Kubiatowicz, Satish Rao and Ben Y. Zhao. *Distributed Object Location in a Dynamic Network.* Accepted for publication to Theory of Computing Systems. Expanded journal version of SPAA 2002 paper.

5. Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms.*

6. Matt Welsh, David Culler, and Eric Brewer. *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,* In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October, 2001

7. D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. *The Switch Ware Active Network Architecture.* IEEE Network, 12(3):29-36. May/June 1998.

8. Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. *PLAN: A Packet Language for Active Networks.* In *Proceedings of the Third ACM SIG-PLAN International Conference on Functional Programming Languages,* p. 86-93, 1998.

9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. *A Scalable Content-Addressable Network.* In Proceedings of ACM SIGCOMM, 2001.

10. Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.*

11. Akamai Corporation. www.akamai.com.

12. Digital Island Corporation. www.digitalisland.net.

13. R. Avnur and J. Hellerstine. Eddies. *Continously Adaptive Query Processing.* In *Proceedings for the 2000 ACM SIGMOD Intl. Conference on Management of Data,* 2000.

14. J. chen, D. J. DeWitt, F. Tian, and Y. Wang. *NiagaraCQ: A Scalable Continous Query System fo Internet Databases.* In *Proceedings of the 2000 ACM SIGMOD Intl. Conference on Management of Data.* 2000.

15. J. M. Hellerstine, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. *Adaptive Query Processing: Technology in Evolution.* IEEE Data Engineering Bulletin, 23(2):7-18, 2000.

16. P. Radoslavov, R. Govindan, D. Estrin. *Topology-Informed Internet Replica Placement.*