OHT: Hierarchical Distributed Hash Tables

Kun Feng, Tianyang Che,
Tonglin Li, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology

{kfeng1,tche,tli13}@hawk.iit.edu,
iraicu@cs.iit.edu

ABSTRACT

This paper presents OHT, a hierarchical distributed hash table, which improves the performance of practical ZHT. When n-to-n connection is needed, every single node has to keep a large number of socket connections. This might be quite expensive when n is extremely large. By performing a hierarchical distributed hash table, OHT can solve this problem. Theoretically, we are able to support an OHT system who consists of millions of nodes. This paper is a final report to give a problem statement, describe related work, and illustrate our design idea and implementation details of OHT. We have evaluated OHT's performance under a Linux cluster with 512-cores. We scale OHT up to 32 physical nodes. The results shows that latency of OHT is around 2.7 ms and throughput can reach 12K ops/s for a 32-node scale. The experiments also show that OHT can work correctly even when failures arise. At the cost of losing some percentage of performance, OHT can be much more scalable.

KEYWORDS

ZHT, hierarchical architecture, DHT.

1. INTRODUCTION

Exascale systems are designed to provide a large number of precious opportunities for science which have never appeared before [1]. By using them, it is possible to use computing power as a helpful tool in theory and experiments related to understanding the basic components of human beings and the whole nature. In order to make an excellent exascale system, we must pay our attention to major architecture, software, algorithm and data challenges. In addition, adapting to newly emerging programming environments is another require [2].

Distributed hash table (DHT) systems are an important class of peer-to-peer routing infrastructures. They can support both long-distance storage and tracing information. They can also support the fast development of a wide range of exascale applications applied in many aspects, such as naming systems, file systems and application level multicast [3]. Distributed hash table systems typically are based on network.

ZHT is an implementation and improvement of distributed hash table. It is designed and tuned dedicatedly for high-end computing (HEC) [4, 17]. HEC includes highly reliable hardware, fast and stable networks, low latencies and patterns for scientific computing data access. It is a good product and platform, and has ambition as well as potential to be the next fundamental component of distributed systems for the upcoming a couple of decades.

However, it has some problems to be solved. First and the most important is the scalability issue. ZHT can support up to 8192 nodes working concurrently. Once beyond this threshold, ZHT is not able to guarantee it works nicely or delivers high performance. What's more, current ZHT version cannot support fault tolerance. Our implementation, OHT, is aimed to solve these two major problems. To give a high-level description, OHT can satisfy scalability by adding a new proxy layer. In order to become fault-tolerant, OHT mainly adopts replication mechanism.

2. BACKGROUND

ZHT is a zero-hop distributed hash table, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed system, such as parallel and distributed file systems, distributed job management systems, and parallel programming systems. The goals of ZHT are delivering high availability, good fault tolerance, high throughput, and low latencies at extreme scale, such as millions of nodes. At this moment ZHT have achieved most of these goals. But it still has some limitations. With n-to-n communication, all nodes have to maintain a large number of socket connections which is quite expensive when the system scale goes into extreme, like 1 million nodes. In some environment which users don't have root permission so are not able to change system parameters such as limit and maximum number of open file descriptors, concurrent connection will be limited to around 1024, and so would scale be limited. A practical approach to address this issue is hierarchical architecture, in which a layer of proxy nodes are added. Each proxy node serves a fixed set of compute nodes, and forwards the request to its server nodes. In this manner one-million-node scale can be easily achieved by using 1000 proxy nodes and each manages 1000server nodes.

3. FUNCTIONIALITY DESIGN

3.1 Overview

The main goal of OHT is to solve existing problems in ZHT but retains all its benefits at the same time. ZHT has high availability, good fault tolerance, high throughput, and low latencies. OHT keeps all these advantages of ZHT and overcomes its drawbacks at the same time. The main problem of current ZHT is the scalability. When the size of nodes reaches 8000, the system may crash down or not continue to have relatively good performance. The reason of this phenomenon is that the possibility that more than 1000 clients concurrently connect to the same server exists. To solve this problem, we revise the original ZHT in a hierarchical way, which is OHT.

OHT has the same application programming interface (API) as ZHT, because it is an improved version of ZHT instead of adding or removing any function. Essentially, OHT is built upon hash table, which means it supports all the basic operations of a typical hash table: 1. insert(key, value); 2. lookup(key); 3. remove(key); 4. append(key, value).

3.2 System Architecture

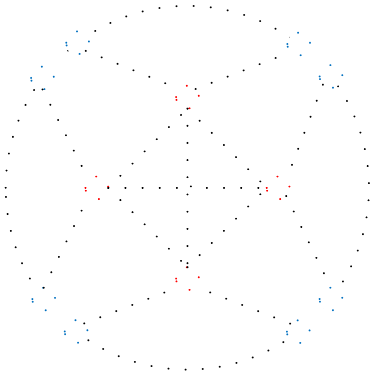


Figure 1. System Architecture.

Figure 1 shows the overall system architecture. Reds spot represents proxies and the blue ones stand for servers. Proxies have n-to-n inter-connection, which means all proxies are connected to each other. In this figure, if there is a solid line between two nodes, that means they have a connection. Each dash line means that two nodes at both its ends don't need to talk to each other, but they are in the server group under the management of the same proxy.

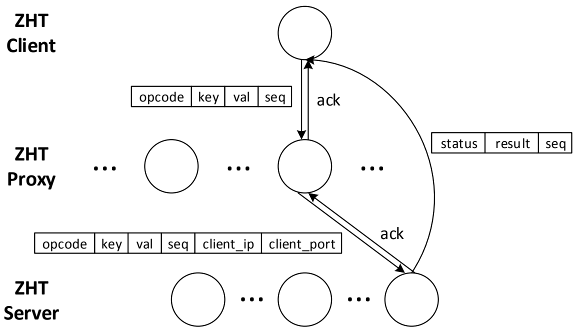


Figure 2. Three Layer System Architecture.

Figure 2 illustrates the details of the system. Clients and servers are the same as the corresponding ones in ZHT. A proxy is a ZHT instance which is responsible for bridging client and server. Each ZHT proxy has a server group consisting of a fixed number of servers. Usually, each proxy can only connect with the servers belonging to its server group. At runtime, a proxy gets a request from client, and forwards it to a server in its group. The way a client chooses a proxy and a proxy chooses a server is by using a hash function. This part will be introduced in Section 3.4.

3.3 Workflow

Inserting a proxy layer can bring in some changes in the original work flow. ZHT has two-step sequence: 1. client chooses a server using a hash function and then sends a request to it; 2. Server responses the request and sends the result back to the corresponding client.

OHT has three-step work sequence:

3.3.1 Client-end workflow

Client still chooses a server using a hash function, and sends the requests to it. We use *sendrecv()* method to pass the message and wait for the response. Each client has two threads: main thread and a dedicated listening thread. Main thread is responsible for picking a proxy, sending it a request and waiting for an acknowledgement. The listening thread waits for the results of the requests it sends out. Note that the result of each request is generated by a server, thus the listening thread only gets connections from servers.

Since a listening thread can get hundreds of results in OHT, each request must has a unique request id. Instead of keeping a request counter, we encapsulate a sequence number into a message to uniquely identify a request. The sequence number is calculated using following formula.

SequNum=hash(key+opcode+clientip+clientport)

When the listening thread receives a message from a server, it matches it with each one in the request queue to make sure the client gets the feedback. We also package the client's IP address and port into the message. In this way, the server is able to send the result back to the corresponding client.

3.3.2 *Proxy-end workflow*

The primary function of proxy is to connect client and server. When a proxy receives a message, it sends an acknowledgement to the client. Then it chooses a server in its server group using the hash function, and then forwards the message to the chosen server. In order to make sure the request has been forwarded to the server successfully, the proxy waits for an acknowledgement. Since proxies are not designed to handle any requests related to hash table, it doesn't have to store anything except its own information, such as the server group. Although the workflow of proxy is straightforward, it is an essential part of our implementation because the proxy's task is to alleviate server's burden.

3.3.3 *Server-end workflow*

Servers are the real workhorses. It gets the message forwarded by its proxy, and does the real operations related to the hash table, such as insert, remove and append. As long as the server gets a request, it sends an acknowledgement to the proxy. Then the server does the operation, and sends the result to a client – the creator of the request according to the IP address and port containing in the message. Servers are required to store the real hash table, and they are also informed the information of their proxies.

3.4 Hash Function Design

OHT uses hash functions in two scenarios: client to proxy and proxy to server. They use similar but different hash functions.

3.4.1 *Client-proxy hash function*

The hash function a client uses to choose a proxy is relatively easy. When each client starts up, it is informed the total number of proxies. Suppose the number of proxies is n_p, the hash function can be as easy as a simple modulo operation.

$$\textbf{\textit{proxy\_index = key\%}}\boldsymbol{n_p}$$

3.4.2 *Client-proxy hash function*

At first, we proposed to use the same hash function mentioned in Section 3.4.1, just replaced n_p with the size of a proxy's server group. Unfortunately, this method has a potential bug, which is hard to be discovered. It is meaningful to explain it clearly as we hope to help others avoid making the same mistake.

Error! Reference source not found. shows a scenario where the failure occurs. Suppose 2 proxies exists in OHT proxy layer and each group has 2 OHT servers. When the client sends a request with a key equals to 0, it will send the message to proxy 0 by using the equation in Section 3.4.1. Then the proxy uses the same equation to choose server 0. As a result, as long as proxy 0 is chosen, server 1 in its group will never be chosen, which means half of the servers will be idle for a long time. This is a huge waste of computing sources.

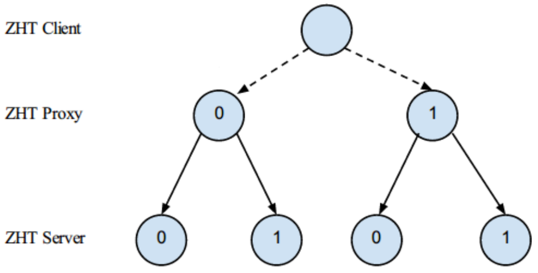


Figure 3. Hash Function Bug Example.

In order to fix this problem, we propose a new hash function to locate the destination server in proxy. Suppose the total number of proxies is $n_{p,}$ the size of server group is n_q, total number of server is *m*. We replace the hash function with the following one.

$$\textbf{\textit{server\_index = key\%m \%}}\boldsymbol{n_q}$$

4. FAULT TOLERANCE DESIGN

Fault tolerance is another aspect that needs much consideration while designing a distributed system. A distributed system can involve in up to millions of physical nodes. The more computers take part in, the more ones may arise failure. ZHT is able to handle failures nicely when it detects some nodes do not respond to requests and then tags them as failed. ZHT adopts replication to ensure data is able to exist when it occurs failure.

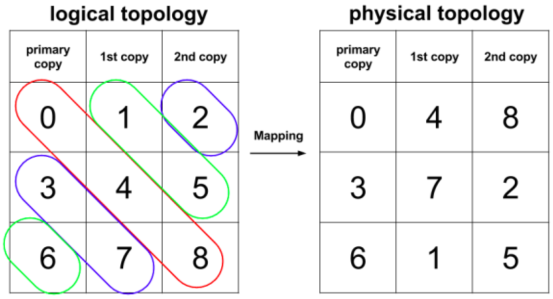
Similar to the original version of ZHT, we build OHT on top of replication. But more kinds of failures can arise in our system due to the import of proxy layer. Failures in clients are not in the range of our consideration since clients are a loose coupling part in OHT. By changing another port in the same physical node or sending requests from a new computer can easily handle this failure. Thus, all the failures can be divided into two large categories, proxy failure and server failure.

4.1 Replication Level

Replication level is a new terminology added by us. The replication level in each layer in OHT is the number of ZHT instances belonging to a single role in this layer. For example, if the replication level in proxy layer is 3, then besides the real proxy instance running on a physical node, there are two proxy instances on two different computers. Since proxy and server have different importance, making them own the same replication level is unreasonable. Thus, each layer has its independent replication level, which is configured by the user either in the command line or default configuration file. The first OHT instance in these replicas is called primary copy.

4.2 Mapping

We design to let all replicas running on different physical nodes. It is meaningful when the primary copy crashes down since a replica will take the charge of providing the same service, such as forwarding the message to a server or responding the request. In practice, thousands of proxies and servers are involved in the system, each proxy and server has different numbers of replicas running on its own and other physical nodes. Although our implementation can efficiently handle failures, we cannot avoid complicated mathematic relation between logical instance layout and physical instance layout. We give an example to show the mapping mechanism.



primary copy	1st copy	2nd copy
0	1	2
3	4	5
6	7	8

primary copy	1st copy	2nd copy
0	4	8
3	7	2
6	1	5

Figure 4. Instance layout example.

Figure 4 is an example showing the mapping mechanism between logical instance layout and physical instance layout. Suppose all these OHT instances are proxy instances, and proxy's replication level is 3. The indexes are from 0 to 8. Three physical nodes participates in the system.

Left table in this figure is the logical topology of these three proxies. Each row represents a proxy, naming proxy 1, proxy 2 and proxy 3 respectively. Right table shows the physical topology of OHT instances running on physical machines. Each row stands for a real computer, naming node 1, node 2 and node 3 respectively. To make all these instances are distributed equally and correctly, a mathematical relation is required. In the left table, instances in one ellipse are running on the same physical node. We put proxy 1's 1st replica, proxy 2's 2nd replica and proxy 3's 3rd replica on node 1. Deploy proxy 2's 1st replica, proxy 3's 2nd replica and proxy 1's 3rd replica on node 2. Arrange proxy 3's 1st replica, proxy 1's 2nd replica and proxy 2's 1rd replica on node 3. By utilizing this mapping, whenever a proxy's primary copy crashes down, we can choose an identical instance on another physical machine.

To make the mathematical relation more general, we will give a formula to compute which instances will be running on a given physical node.

Suppose we have n proxies, indexing from 1 – n. The number of physical nodes will be also n, since each primary copy occupies one computer. Similarly, they have indexes ranging from 1 to n. The replication level is r. We use *i* to stand for proxy's index, and *j* for physical node index. OHT instances running on node *j* can be calculated with following formula.

$$i = j,\ index\ i's\ first\ copy$$

$$i \neq j,\ i's\ (r - i\%r)\ th\ copy$$

By using this equation, we can compute all the OHT instances one physical node needs to run before it starts up. Through a number of experiments and serious verifications, we prove that this method works correctly to meet our requirements related to fault tolerance issue.

4.3 Strong consistence in proxies

OHT proxy and server has different importance. Proxy is much more important than server. If a server crashes down, it will not affect other servers in the same server group or the remote ones. Proxy is on the opposite side. If a proxy crashes down, the request cannot be forwarded to all the servers it manages, which means when the scale of OHT is large, thousands of nodes will not be reachable. If proxy and server both have weak consistence, OHT may be less reliable and has low availability. If they own strong consistence, more overhead will be imported due to too many and frequent connections among servers. As a result, we decide to implement strong consistence in proxy layer and relatively weaker one in server layer.

In OHT, a proxy's failure is detected by a client since all requests are generated and sent from clients. When one request times out, which means the message's receiver does not work, the client marks down the primary copy of the proxy, and chooses a copy randomly. Since the copy is running on the other physical node, the system can work properly with little sacrifice on performance resulting from choosing a copy randomly.

Each proxy stores information related to all the other proxies and their copies. These information includes each OHT instance's IP address, port and their statuses. One proxy also stores all the servers' information in its server group.

Information in proxies are frequently exchanged in proxy layer. The server failure is detected by a proxy due to server only waits for message forwarded from its proxy. Whenever a proxy finds out a server failure, it marks the server's primary down, and then sends broadcasts this failure information to all the other proxies immediately. This is an n-to-n connection, which means every proxy has the possibility to communicate with all the other ones. At first glance, this design may cause much overhead, but in fact it will not. The message needed to be broadcasted often has several KB size, and the total number of proxies is typically small compared to that of servers. Thus this method can store extremely important information in a number of proxies at the cost of little overhead.

5. EVALUATION

5.1 Setup

The experiments were conducted on a 65-node SUN Fire Linux cluster called HEC in SCS lab. Each computing node has two AMD Opteron(tm) processors, 8GB memory and a 250GB HDD. The operating system is Ubuntu 9.04 with Linux kernel version 2.6.28.10. All nodes are connected to a 1 Gbps Ethernet.

5.2 Methodology

In the experiment in [4], the ratio between the number of clients and that of servers is kept to be 1, which means the number of ZHT clients increases as the scale of ZHT doubles. So the scalability of ZHT can be evaluated simply by increasing the number of ZHT servers. However, compared to the original ZHT, we add a new layer in the system so that the number of proxies is another variable we can change during the test. In order to make the comparison between ZHT and OHT fairer, we keep the ratio between the number of clients and that of serves to be 1 as well firstly. And then we add the number of proxies to be the third variable which makes our test more comprehensive and reasonable.

In our experiments, all proxies and servers are running on different nodes to avoid potential local network communication. Such a configuration will make the result more stable since every connection between proxy and server will have almost the same latency. OHT clients, however, run on the same node as servers. That is exactly how the ZHT experiment was done then. In this setting, the inequality of local and remote connection is not a big concern because the requests issued from one client will be sent to all proxies and then forwarded to all servers.

In every run of our experiment, each client will issue 10000 requests to the proxies. Since each proxy manages a group of servers which store a partition of the whole key space, each proxy will handle a group of partitions in the key space and these partitions of all proxies are contiguous and have no intersection with each other. All 10000 requests will be approximately evenly sent to all proxies due to the randomly generated key in client. It can make the workload evenly distributed to make the system more stable.

For the implementation, we use the default ZHT parameters. TCP protocol is used and TCP connection cache is enabled to reduce the cost in creating and destroying sockets between clients and proxies. And we mimic the implementation in ZHT to enable TCP connection cache between proxies and servers as well.

We time the running time of the zht_ben benchmark using our own method instead of the original one since receiving an acknowledgement from proxy does not mean the request is finished. A request is not processed until the result is returned from server to client. In our timing method, the start time of one benchmark instance is recorded before the first request is issued and the end time is measured in the dedicated listen thread in client when the last request comes back without any error from the server. The average time of all benchmark instances is calculated to get a more accurate overall time for evaluation. The average processing time of one request is calculated by dividing the average overall time by the number of requests in one client, which is 10000 for our case.

5.3 Performance Evaluation

5.3.1 *Effect of the number of proxies*

As mentioned above, the number of proxies is another important parameter in OHT. The more proxies, the requests can be forwarded to servers faster and the clients can get response from the servers in a shorter time. However, more proxies need more node resources and more sockets. To test the effect of the number of proxies, we firstly vary the number of proxies from 1 to up to 32 keeping the ratio between the scale of server and client. The results is shown as Figure 5.

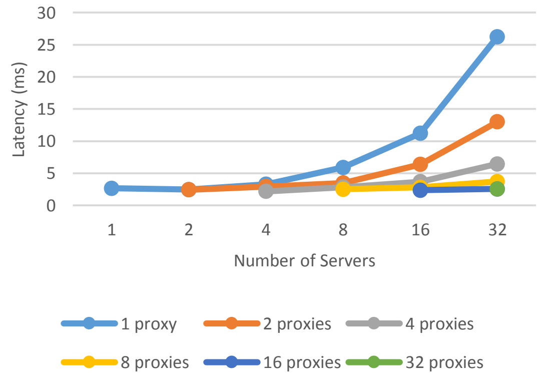


Figure 5. Latency of OHT under different number of proxies and the numbers of clients and servers keep same.

The trend is obvious that with the fixed number of servers, more proxies can bring shorter latency. Take the experiment of largest scale for instance, the shortest latency is obtained when the number of proxies is 32.

Removing half of the proxies almost double the latency. That is because the workload is distributed by less proxies, the queue length in the epoll loop in proxies and servers is almost doubled. But the latency is not reduced anymore when the number of proxies increase from 16 to 32. That indicates that 16 proxies can handle 32 clients very well. Introducing more proxies cannot bring any performance improvement and but only add more resource consumption. Similar trends can be seen for results of other scales.

It can also be observed from the figure that, for a fixed number of proxies, more clients will lead to larger latency. If there is only one proxy, the latency approximately follows an exponential curve when the number of servers increase from 4 to 32. The latency does not increase very fast when the number of servers is relatively small. It can be explained as the conclusion before. A smaller number of clients can be handled by one proxy very well. Thus for economical reason, keeping the ratio between the number of clients and proxies to be four is the most cost-effective setting for OHT.

Then we fix the size of clients to be 32 and vary the combination of the number of proxies and servers to test the capability of proxy to handle multiple servers. Theoretically, the more server one proxy manages, the higher the latency we get. That is because the proxy may have to send to and receive from multiple servers in the same time. However, as shown by the results in Figure 6, the number of servers under one proxy does not make too much difference. For the fixed number of proxies, the latency is almost identical for various scales of clients. We believe the result is reasonable since the number of requests arrive at one proxy at the same time is the most vital factor to affect latency. Communicating with multiple servers is not the bottleneck, especially when we use socket cache between this two layers.

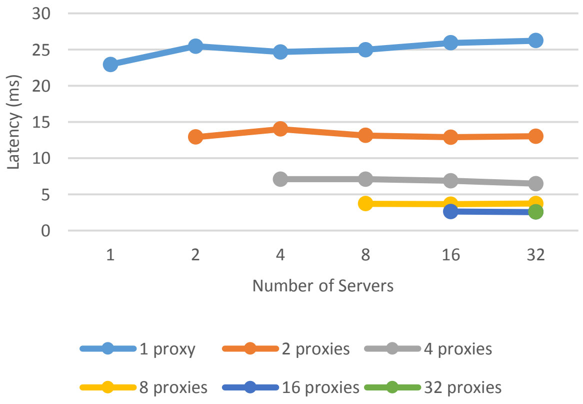


Figure 6. Latency of OHT under different number of proxies and servers with fixed 32 clients.

At last, we fix the size of servers and vary the number of proxies and clients. This is to evaluate how proxy can handle multiple clients in a system in a more detailed way. Such a configuration will be more like the scenario in real usage in which the number of clients is larger than the number of proxies. To eliminate the effect of the number of servers, we set the number of servers to be 32 to maximize the performance in server side. The results can be seen as the Figure 7.

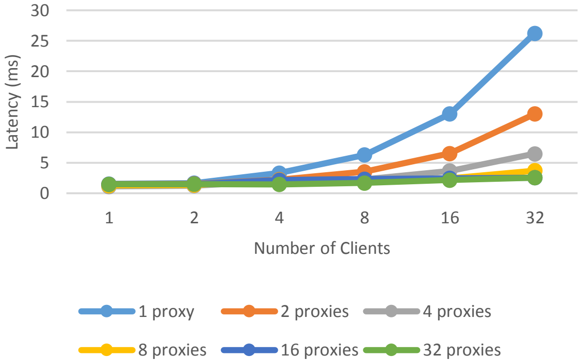


Figure 7. Latency of OHT under different number of proxies and clients with fixed 32 servers.

As we expected, with a fixed number of proxies, more clients will introduce higher latency. More clients result in longer events queue in the epoll loop in proxy so that requests will wait longer time to be processed. In addition, it should also be mentioned that one proxy can about 4 clients very well simultaneously. It means epoll can handle four concurrent events without losing too much performance. When the ratio between the number of clients and proxies go higher than four, the latency will increase proportional to the number of clients. In order to keep the latency under an acceptable threshold, the number of proxies need to be increased as the scale of clients in the real usage.

Above all, it always holds true that the largest number of proxies can always bring best performance. For the remaining part of this report, the result is obtained with the most proxies.

5.3.2 Latency

We evaluated the latency of OHT on HEC using the setting mentioned above. We use up to 32 servers and 32 clients and the number of proxies vary from one to the number of servers/clients.

For the setting of the number of proxies, we choose the best setting from the experiment above in which clients, proxies and servers have the same scale. The result is shown in Figure 8.

We can see from the figure that the latency of OHT is about three times larger than that of the original ZHT.

That is due to the hierarchical architecture we use in OHT. In the original ZHT, only two messages only needs to be transmitted: sending from request to server and receiving result from server. But in OHT, due to the introduction of the proxy layer, six messages needs to be transferred as discussed in section 2. Three times more network messages and forwarding cost in proxies generates a reasonable 3.38 larger latency on average. The latency curve of ZHT under different scales is very stable while OHT shows more turbulence because more connections and routes are involved in the process of one request. But OHT still shows acceptable stability compared with other more complex systems.

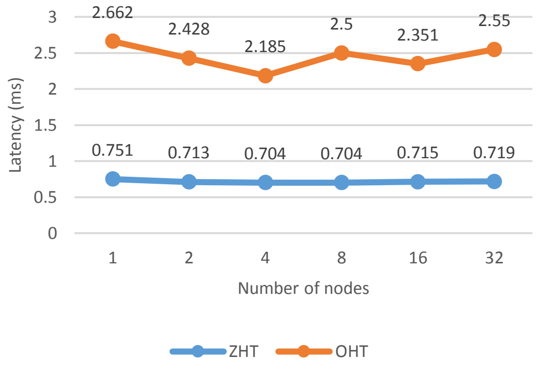


Figure 8. Latency comparison between OHT and ZHT.

5.3.3 Throughput

Throughput indicates how many requests the system can handle in a period of time. The comparison of throughput between ZHT and OHT can be seen from Figure 9.

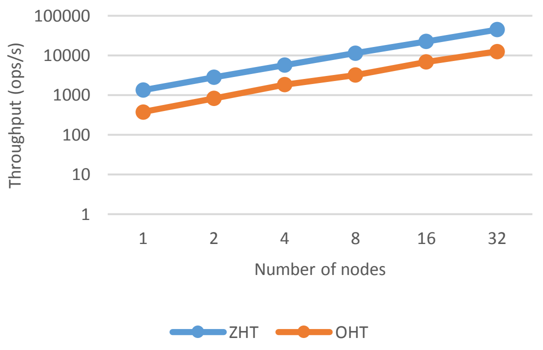


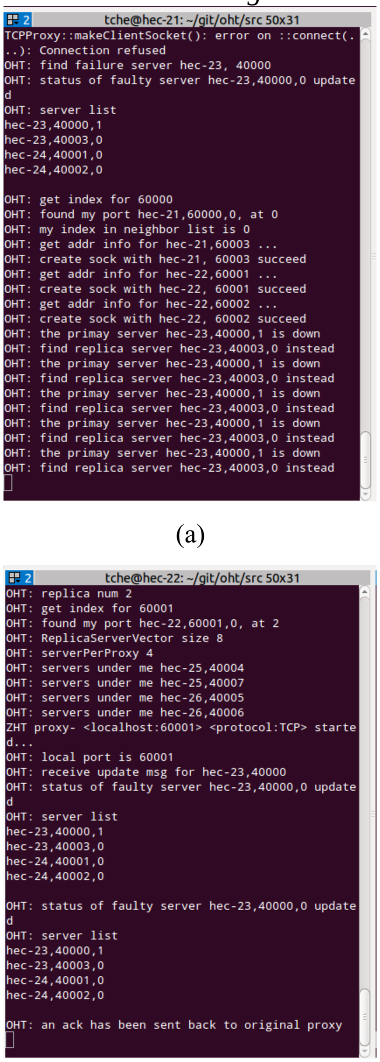
Figure 9. Throughput comparison between OHT and ZHT.

As can be observed in the figure, OHT can reach 12K ops/second on 32 nodes scale. But compared with ZHT, OHT shows a much lower throughput. On average, OHT has 29.5% of the throughput of the original ZHT. We attribute the performance loss to the additional four network message transmissions in OHT. Compared with Cassandra in [4], the performance is about the same.

5.4 Fault Tolerance Evaluation

As mentioned before, OHT supports tolerance of two kinds of failures, server failure and proxy failure. In order to verify the fault tolerance of OHT, we test two failures separately. In each test, other parts of the system remains fault-free.

5.4.1 Server Failure Handling



```
TCPProxy::makeClientSocket(): error on ::connect(.
..): Connection refused
OHT: find failure server hec-23, 40000
OHT: status of faulty server hec-23,40000,0 update
d
OHT: server list
hec-23,40000,1
hec-23,40003,0
hec-24,40001,0
hec-24,40002,0

OHT: get index for 60000
OHT: found my port hec-21,60000,0, at 0
OHT: my index in neighbor list is 0
OHT: get addr info for hec-21,60003 ...
OHT: create sock with hec-21, 60003 succeed
OHT: get addr info for hec-22,60001 ...
OHT: create sock with hec-22, 60001 succeed
OHT: get addr info for hec-22,60002 ...
OHT: create sock with hec-22, 60002 succeed
OHT: the primay server hec-23,40000,1 is down
OHT: find replica server hec-23,40003,0 instead
OHT: the primay server hec-23,40000,1 is down
OHT: find replica server hec-23,40003,0 instead
OHT: the primay server hec-23,40000,1 is down
OHT: find replica server hec-23,40003,0 instead
OHT: the primay server hec-23,40000,1 is down
OHT: find replica server hec-23,40003,0 instead
OHT: the primay server hec-23,40000,1 is down
OHT: find replica server hec-23,40003,0 instead
```

(a)

```
OHT: replica num 2
OHT: get index for 60001
OHT: found my port hec-22,60001,0, at 2
OHT: ReplicaServerVector size 8
OHT: serverPerProxy 4
OHT: servers under me hec-25,40004
OHT: servers under me hec-25,40007
OHT: servers under me hec-26,40005
OHT: servers under me hec-26,40006
ZHT proxy- <localhost:60001> <protocol:TCP> starte
d...
OHT: local port is 60001
OHT: receive update msg for hec-23,40000
OHT: status of faulty server hec-23,40000,0 update
d
OHT: server list
hec-23,40000,1
hec-23,40003,0
hec-24,40001,0
hec-24,40002,0

OHT: status of faulty server hec-23,40000,0 update
d
OHT: server list
hec-23,40000,1
hec-23,40003,0
hec-24,40001,0
hec-24,40002,0

OHT: an ack has been sent back to original proxy
```

(b)

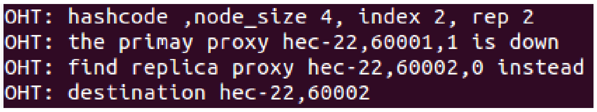
Figure 10. Server failing handling process. (a) Initiator proxy; (b) Receiving proxy.

We manually kill a server during the benchmark to test the fault tolerance of OHT. In OHT, the server failure is detected by the proxy when it tries to connect with the server. As shown in Figure 10(a), a connection error is recognized as the failure of the server. To simplify the code, we did not check whether the error is caused by the failure or other reasons like network congestion. The proxy firstly marks the server to be down in local server list and then sends the update event to other proxies to maintain strong consistency in server list information. The broadcast among proxies is done by one-by-one communication as shown in the figure. When a new request comes, the failed server will not be selected as destination. One of the replicas will take the place of the original copy.

As a replica or proxy managing other servers, it just works as normal. When a request comes with the aforementioned operation code indicating a failure event, the proxy will update its local server list to mark the corresponding to be down as shown in Figure 10(b). Then when some proxy fails, this information can be used to recover the latest server list under the management of that failed proxy.

5.4.2 Proxy Failure Handling

Proxy failure is detected by the client. The handling is much easier than server failure.



```
OHT: hashcode ,node_size 4, index 2, rep 2
OHT: the primay proxy hec-22,60001,1 is down
OHT: find replica proxy hec-22,60002,0 instead
OHT: destination hec-22,60002
```

Figure 11. Proxy failure handling process.

As shown in Figure 11, the client just marks the proxy to be down when the similar situation happens in the client as the error in proxy for server failure. And it finds randomly one of the replicas to send out the request. In this test, (hec-22, 60001) is down and the replica (hec-22, 60002) will be selected instead.

6. RELATED WORKS

There have been a lot of existing distributed hash table (DHT) algorithms and implementations. Some of the existing DHT are Kademlia [5], CAN [6], Chord [7], Memcached [8], Dynamo [9], Cassandra [10] and C-MPI [11]. DHT plays an important role in building support for scalable metadata service across extreme scale system. For example, FusionFS [11] uses ZHT as the metadata storage to build a distributed file system for extreme large scale system. As result in [4], ZHT outperforms Cassandra and has almost the same performance as Memcached which is an in-memory implementation.

For DHT with a hierarchical design, several works have been proposed. Canon [12] is a paradigm for designing hierarchically structured DHTs. Chord, CAN and Kademlia all can be transformed into a hierarchical architecture following Canon. Cyclone [13] is another similar which follows a uniform leaf-based approach that considerably reduces the overall number of links per node. HyCube [14] is a DHT based on a hierarchical hypercube geometry. It adopts the Steinhaus transform for variable metric. A multi-level distributed hash table is proposed in [15] in name resolution service (NRS) for information-centric networking (ICN) system to minimize inter-domain traffic and to reduce latency.

7. FUTURE WORK

The most significant contribution of OHT is the scalability. A real test on a real large scale system is the best way to verify the potential. In the future, we plan to do such experiments to confirm the design and implementation of OHT. In this report, we have an assumption that the underlying servers have replicas which support eventual consistency. Merging code with another group working on eventual consistency is a must to build a complete and practical system.

8. CONCLUSIONS

In this report, we designed and implemented a hierarchical ZHT named OHT. By adding a proxy layer into ZHT, OHT can support one million nodes via deploying 1000 proxies with 1000 servers under one proxy. The problem in ZHT is solved by isolating enormous amount of servers from clients and using proxies to forward requests. Experimental results show that OHT performs about 3.38 times slower than ZHT due to the newly introduced layer and, consequently, network communications. Considering OHT can support much larger scale, the latency increase is acceptable.

9. ACKNOWLEDGMENTS

Our great thanks to Tonglin Li for his enormous help to the design and implementation of our ideas. And we also want to thank Dr. Raicu for giving us an excellent course and in-depth knowledge about data-intensive computing.

10. REFERENCES

[1] A. Geist and R. Lucas, “Major Computer Science Challenges At Exascale,” *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 427–436, Sep. 2009.

[2] S. Borkar, “The Exascale challenge,” in *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, 2010, pp. 2–3.

[3] H. Zhang, A. Goel, and R. Govindan, “Incrementally improving lookup latency in distributed hash table systems,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, p. 114, Jun. 2003.

[4] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, “ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table,” in *2013 IEEE*

27th International Symposium on Parallel and Distributed Processing, 2013, pp. 775–787.

[5] D. M. P. Maymounkov, “Kademlia: A Peer-to-peer Information System Based on the XOR Metric,” in *Proc. IPTPS*, 2002.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, “A scalable content-addressable network,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, Oct. 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord,” in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM ’01*, 2001, vol. 31, no. 4, pp. 149–160.

[8] B. Fitzpatrick, “Distributed caching with memcached,” *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205, Oct. 2007.

[10] “Cassandra,” 2012. [Online]. Available: http://cassandra.apache.org/.

[11] Ioan Raicu, Dongfang Zhao, Chen Shou, Zhao Zhang, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, “FusionFS: a distributed file system for large scale data-intensive computing,” in *2nd Greater Chicago Area System Research Workshop (GCASR)*, 2013.

[12] R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., “HyCube: A DHT Routing System Based on a Hierarchical Hypercube Geometry,” in in *Parallel Processing and Applied Mathematics Lecture Notes in Computer Science*, vol. 6068, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[13] M. D’Ambrosio, C. Dannewitz, H. Karl, and V. Vercellone, “MDHT,” in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking - ICN ’11*, 2011, p. 7.

[14] H. Garcia-Molina, “Canon in G major: designing DHTs with hierarchical structure,” in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004, pp. 263–272.

[15] M. S. Artigas, P. Garcia Lopez, J. P. Ahullo, and A. F. Gomez Skarmeta, “Cyclone: A Novel Design Schema for Hierarchical DHTs,” in *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P’05)*, 2005, pp. 49–56

[16] T. Li, R. Verma, X. Duan, H. Jin, I. Raicu. "Exploring Distributed Hash Tables in High-End Computing", ACM Performance Evaluation Review (PER), 2011.