Jupiter: A Robust, Efficient and Secure Middleware for Geo-distributed Storage Systems

Quanqing Xu, Yonghong Wang, Khai Leong Yong, Khin Mi Mi Aung

Data Storage Institute, A*STAR, Singapore
{Xu_Quanqing, Wang_Yonghong, Yong_Khai_Leong, Mi_Mi_Aung}@dsi.a-star.edu.sg

Abstract

In order to meet the needs of increasing users and improve user-perceived latency, online services distribute and replicate data across geographically diverse data centers and direct user requests to the closest or least loaded server. Distributed Hash Table (DHT) is a structured overlay network that is widely utilized in geo-distributed storage systems, e.g., Dynamo. Some geo-distributed storage systems may need to locate an item with only keywords. In this paper, we present Jupiter, a DHT-based middleware system for building geo-distributed storage systems. Jupiter provides robust and efficient routing mechanisms under geo-distributed environments. In addition, it also presents a secure routing policy to prevent dangerous routing messages. The key innovation in Jupiter is the integration of three concepts: robustness, efficiency and security. We have prototyped Jupiter, deployed it on a network of Linux machines, and used it to develop several distributed applications. We confirm the practicality, effectiveness and efficiency of Jupiter by conducting an extensive performance benchmark measured by efficiency, robustness, consistency and bandwidth.

Keyword: Middleware, Geo-distributed Storage, Robustness, Efficiency, Security.

I. Introduction

In order to meet the needs of increasing users, scaling online services over the Internet is challenging to EB-scale storage [1]. To improve user-perceived latency that directly affects the quality of the user experience, online services distribute and replicate state across geographically diverse data centers and direct user requests to the closest or least loaded site. Ford et al. [2] proposed geo-replication as an effective technique to prevent data loss under large scale concurrent node failures. Geo-replication across geographically dispersed sites is a fail-safe way to ensure data durability under a power outage. However, not all storage providers have the capability to support geo-replication. In addition, even for data center operators that have geo-replication, e.g., Facebook's TAO [3], losing data at a single site still incurs a high fixed cost due to the need to locate or re-compute the data.

Distributed hash table (DHT), e.g., Chord [4], is a structured overlay network that is widely utilized in geo-distributed storage systems, e.g., Dynamo [5] and Cassandra [6]. Some geo-distributed applications may need to locate an item (e.g., a file or document) with only keywords [7], e.g., Content Distribution Network (CDN) [8]. Existing techniques in DHT-based networks concern several problems: robustness in routing table maintenance mechanisms, efficient routing, and security on routing. Work on robustness in the context of overlay network maintenance has mostly focused on how to handle churn [9], transient routing failures and high CPU load. Moreover, routing efficiency is another important problem in DHT networks. Many studies focus on routing efficiency in a DHT [10]. In addition, how to keep consistence of replicas in DHT or geo-replication storage is also a difficult issue [11]. For example, in cloud backup [12], backup data is stored in geo-distributed storage that maintains multiple replicas across several data centers.

Our goal is to make a new generation of general DHT-based robust and efficient middleware for geo-distributed storage systems. To do this, we have designed and implemented Jupiter, a DHT middleware system where we use the Boolean model (i.e. exact match) to retrieve information. We design a series of policies to maintain a robust routing table. In addition, from the standpoints of

routing efficiency, we present an iterative, concurrent UDP-based routing mechanism. The main contributions of this paper are fourfold:

- Jupiter is robust in routing. We design a series of policies to maintain a robust routing table in Jupiter, based on a previous study: node lifetimes in P2P systems follow a heavy-tailed distribution;
- Jupiter is efficient in routing. We design Jupiter DHT middleware system from the standpoints of routing mode, supporting concurrent routing or not, connection type between two machines, which make Jupiter efficient;
- Jupiter is secure in routing. All the received messages must be checked via several steps to prevent serious attacks;
- We confirm the effectiveness and efficiency of our methods by conducting an extensive performance measured by efficiency, robustness, consistency and bandwidth;

The remainder of this paper is organized as follows: Section II reviews the related work. We present the system architecture of Jupiter in Section III. Section IV describes how to maintain a routing table of Jupiter. Section V designs efficient routing protocols. Section VI discusses how to achieve high security on routing. Section VII reports the experimental results. Lastly, Section VIII leads to conclusion.

II. Related Work

1. Geo-distributed Storage Systems

Previous geo-distributed storage systems face the unpleasant trade-off between strong semantics and low latency. Spanner [13] provides strong semantics with order-preserving serializable transactions, but these are expensive: like its predecessor Megastore [14]. In Spanner, transactions are updated, which take many cross data center round trips to execute and commit. MDCC [15] is faster, but it still incurs cross data center latency to execute and commit transactions. At the other end of the

trade-off, Cassandra [6] and Dynamo [5] are key-value storage systems offering eventual consistency, while PNUTS [16] offers the slightly stronger per-record timeline consistency. Other systems provide stronger but relaxed semantics to achieve low-latency. Eiger [17] and COPS [18] offer causal+ consistency, in which write conflicts are resolved deterministically. Moreover Eiger and COPS require replication of all data across all data centers. Walter [19] provides parallel snapshot isolation and Gemini [20] provides Red/Blue consistency. Apart from weakened semantics, the latter two systems do not have a scalable design within a data center.

In Nomad [21], Tran et al. proposed storage overlay as an efficient migration mechanism to migrate e-mail data across multiple data centers. In Volley [22], Agarwal et al. utilized system logs of accesses to determine a data center for each data item based on access interdependencies, the identity and time stamp of data access, and the balance of storage capacity across multiple data centers. In Pileus [23], client applications are allowed to declare consistency and latency requirements in the form of SLAs, which include latency and staleness bounds but do not support the types of probabilistic guarantees. Internally, Pileus enforces the SLAs by choosing which replica to access in an SLA-aware manner, whereas Dynamo-style systems tend to always access the closest replicas. In Tuba [24], consistency SLAs are supported by automatically reconfiguring the locations of its replicas in response to the client's location and request rates. Other related works also include placing social media files across clouds [25] and a substantial body of literatures studying data placements in Content Distribution Networks.

2. DHT-based Storage Systems

Castro et al. [26] present a version of Pastry, MSPastry, which self-tunes its stabilization period to adapt to churn and achieve low bandwidth. MSPastry also estimates the current failure rate of nodes, using historical failure observations. Deb et al. [27] proposed a technique for improving average lookup times in DHT systems by caching auxiliary neighbours based on the access frequencies of nodes. It is particularly useful for applications such as name services in mobile environments or

location services, where there are low churn rate for nodes and relatively higher churn rate for items. Attacks [28] on the data management level may be used to create a high load imbalance, seriously degrading the correctness and scalability of DHT-based systems.

Bamboo [9] has a careful routing table maintenance strategy that is sensitive to bandwidth-limited environments. The authors advocate a fixed-period recovery algorithm, as opposed to the more traditional method of recovering from neighbour failures reactively, to cope with high churn. Bamboo also uses a lookup algorithm that attempts to minimize the effect of timeouts, through careful timeout tuning. Li et al. [29] proposed a protocol called Accordion that adjusts the size of the routing table based on a user-specified bandwidth budget and churn for maintenance. Mercury [30] employs a small world distribution for choosing neighbour links, but optimizes its tables to handle scalable range queries rather than single key lookups. A number of file-sharing P2P applications allow the user to specify a maximum bandwidth.

III. System Architecture

1. Preliminaries

A. The Keyword Search Problem in DHT Networks

To provide keyword search service in DHT networks, we assume that each item (e.g., a file or document) $x \in I$ is associated with a set K_x of keywords. For any item x, a keyword set K can describe x if $K \subseteq K_x$. For each set K of keywords, we define a set I_K of items, where $I_K = \{x | x \in I, K \subseteq K_x\}$. That is to say, I_K is the set of items that can be described by K. The size $|I_K|$ is called the keyword frequency of K.

To provide keyword search service, a distributed index (DHT) scheme is designed so that an item can be located via a query including a few keywords. There are two kinds of searches in the service: 1) Exact Search: Given keyword set K, the service should return the set $\{x|K_x = K\}$ of items that are associated with exactly the keyword set K, e.g., P2P file-sharing systems. 2) Top-k Search: Given

keyword set *K* and a given threshold *t*, the service should return a set of min(t, |K|) items that can be described by *K*.

B. Jupiter Design Principle

A general DHT provides the following semantic operations:

- **Put(*Key, Value*)** Publishes a pair (*Key*, *Value*) into DHT networks;
- **Get(*Key*)** Returns the *Value* of the *Key*;
- **Lookup(*Key*)** Provides general access to the node maintaining the *Key*;
- **Routing(*Key*)** Provides general access to the node responsible for the *Key*, and to each node along the routing path;

where "Put/Get" interfaces are provided to upper layers. In fact, "Put/Get" interfaces should be provided by another component since routing and storage are both important infrastructures. The "Routing" interface may be replaced by "Lookup". Therefore, Jupiter provides a "Lookup" interface to upper layers and can support many applications at the same time. We present the following scenario as read in Figure 1, where Content Delivery Network (CDN), Social Networking Service (SNS), and Online Backup run on Jupiter.

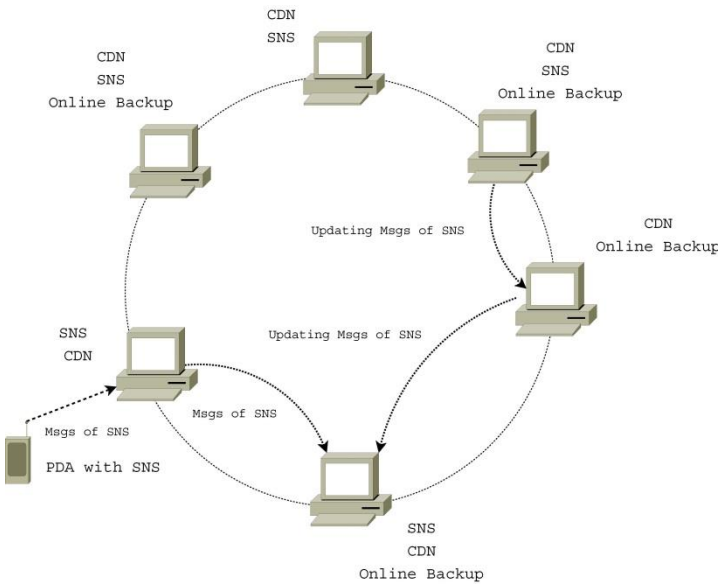


Figure 1: A Jupiter-based Scenario

2. Architecture Description

We present Jupiter's system architecture as shown in Figure 2. Typically, upper applications such as social network service, content delivery network and online backup, are placed on top of Jupiter, which in turn is built on a physical network. Here we have inserted a keyword search layer in between the application layer and the DHT overlay to facilitate item retrieval. An asynchronous distribution approach [31] can be employed in Jupiter.

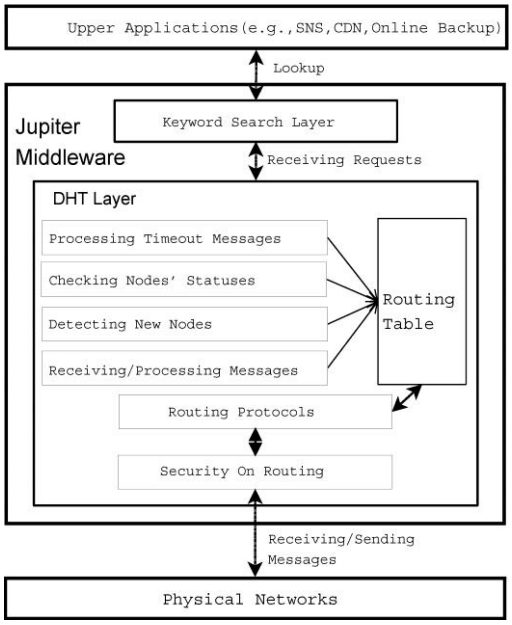


Figure 2: Jupiter's System Architecture

The keyword search layer is built on a DHT network to provide guaranteed search, in which it can be located with reasonable cost if an item is residing somewhere in the network. The DHT overlay of Jupiter in Figure 2 consists of the following components: 1) maintaining a routing table; 2) routing protocols; 3) security on routing. There are four components to maintain a robust routing table:

- *Receiving/processing new messages.* This component receives and processes new messages. In addition, it verifies the integrity of messages and gives the corresponding operations;
- *Detecting new nodes.* This component is regularly run. When the number of nodes in a routing table is not enough, this component tries to detect new nodes. It will be discussed in detail in IV-B;

- *Checking nodes' statuses in a routing table.* This component is also regularly run to clean the off-line nodes, and the nodes that have not been contacted for a long time. We will discuss in detail in IV-B;
- *Processing time-out messages.* For a requesting message, a response message is necessary for answering the requesting one. This component maintains a message queue for waiting responses. For a responding timeout message, this component records a timeout for the corresponding field of the destination node in a local routing table. When many timeouts happen, the corresponding node is considered to be off-line and removed from the local routing table via a specific component.

IV. Design a Robust Routing Table

Routing table is a core component in a DHT. Each node maintains its routing table and utilizes it to connect other nodes. In DHT networks, there is a rule: each node records as many valid nodes as possible and finds target nodes with the cost of O($logN$). Therefore, a series of ingenious data structures are designed to reach the best performance in theory with the smallest memory cost.

1. Routing Table Structure

From the view of a routing table's size, a big routing table can provide more routing choices, which can bring a shorter number of hops and provide many different delay choices for each hop. However, a big routing table incurs too much cost: the corresponding communication cost is spent for keeping statuses of nodes in the routing table new enough. For finding target nodes with the cost of O($logN$), all the entries in a routing table of Jupiter are organized to a tree, which includes two kinds of entries: intermediary entry and leaf entry. Leaf entry is called bucket. A routing table at least includes a bucket, which includes a series of nodes' information: IP address, port and some relevant information for maintaining the routing table.

2. Maintaining Routing Table

Churn is inevitable in today's data centers [32] and must be considered in designing geo-distributed storage systems. For maintaining routing performance (efficiency and fault-tolerance), a node needs to pay the price of bandwidth to maintain its routing table. On the one hand, a node needs to assure that nodes' information in its routing table is new enough. In other words, nodes in its routing table can be confirmed to be accessed and connected for decreasing invalid nodes as many as possible. Invalid nodes in a routing table result in larger delays because of waiting time-outs of network packets in routing. On the other hand, a node needs to retrieve new nodes' information in geo-distributed storage systems within the limit of bandwidth and CPU resource.

If node lifetimes follow a memoryless exponential distribution, the probability p of a neighbour being alive is determined only by the time interval since the neighbour was last known to be alive. However, in real systems, the distribution of node lifetimes is often heavy-tailed: nodes that have been alive for a long time are more likely to stay alive for an even longer time. In subsequent design and maintenance of routing table, we suppose that node lifetimes follow a heavy-tailed distribution.

2.1 Adding New Nodes

A node needs to consolidate its routing table with trying various methods when its routing table does not have enough nodes. At the same time, the node also needs to be added into other nodes' routing tables. There are two means to find new nodes in Jupiter:

- **Passive monitoring.** When a node p receives a message from an unknown node, this unknown node is validated and added into the local routing table.
- **Active detecting.** This component checks each bucket regularly. If there are less than θ_p (θ_p =60 in the current setting) percent of the nodes in a bucket, this component randomly detects a node to consolidate this bucket. We present a policy to generate this node's ID. For a given bucket *Bkt*, a certain amount of front bits are the same. Therefore, a node's ID may be generated: a certain amount of front bits lies in the bucket *Bkt* pluses back bits that are

generated randomly. And then a Lookup operation is exploited to find this node. After this Lookup, some other nodes are added into the local routing table with high probability.

Algorithm 6.1: Adding a New Node

Input: a bucket *Bkt*

1 Calculating the active node ratio θ_p in *Bkt*;

2 if $\theta_p < \theta_{\text{th}}$ then

3 NodeID = concatenating a certain amount of front bits in *Bkt* and randomly generated back bits;

4 Nodes = Lookup(NodeID);

5 Adding Nodes into the local routing table;

2.2 Deleting Old Nodes

A node must be new enough in a routing table. That is to say, the node must be connected with high probability in that moment. However, a dead node is probably not connected and is also called an invalid one. Invalid nodes in a routing table can degrade routing performance. Two nodes in the network across data centers can communicate with each other by sending messages. During this procedure, a node can check another node's status by timer and time-out mechanisms. However, casual network error and congestion can result in misunderstanding the counterpart's status. Therefore, casual message time-out cannot cause that the node is deleted from a routing table. If a node has several continuous message time-outs, it is removed from a routing table. This component keeps nodes in a routing table new enough and decreases the number of invalid nodes by detecting and recording online nodes' information.

Thus, a node's information in a routing table includes the entry time, the last contact time and the number of continuous time-outs. This component is regularly run (2 minutes each time in the current setting) to maintain each bucket. For each node p in a bucket, this component calculates the difference d_p of the last check and the current time. If $d_p < T_{th}$ (T_{th}=10 minutes in the current setting), the node p is new enough, which is not checked. Otherwise, if p has N_{th} (4 times in Jupiter) timeouts,

the node *p* is invalid and may be removed from the local routing table. If not, the initiator sends a *PING* message to *p*. If *p* returns a *PONG* message to the initiator, the number of *p*'s timeout is initialized to be zero and its last check time is modified. For saving bandwidth, only a node in each bucket is detected. If there are several nodes to be checked for each bucket, the last node that is added into the local routing table is checked because of the heavy-tailed distribution of node lifetimes.

Algorithm 6.2: Deleting an Old Node

Input: p

1. Calculating the difference d_p of the last check and the current time;
2. **if** $d_p < T_{th}$ then
3. p is not checked;
4. **else**
5. **if** *p* has N_{th} timeouts **then**
6. *p* is invalid and may be removed from the local routing table;
7. **else**
8. the initiator sends a *PING* message to *p*;
9. **if** *p* returns a *PONG* message to the initiator **then**
10. the number of *p*'s timeout is initialized to be zero;
11. *p*'s last check time is modified

2.3 Cost Analysis

We analyze bandwidth consumption in brief, which is mainly relevant with the several factors: the size of a routing table, the fresh degree of nodes in a routing table, the active degree of nodes in geo-distributed storage systems, and the active degree of nodes in upper applications. In general, bandwidth consumption consists of the following four components:

- **Requesting messages driven by upper applications**. This cost is inevitable. It can be as low as possible through organizing the routing table reasonably.

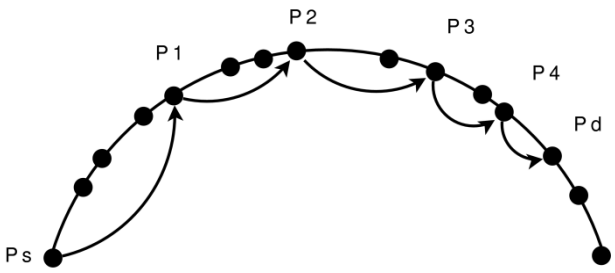
- **Replying other nodes' requests**. This cost is also inevitable. If the system can function properly, received messages by each node are relatively normal. If not, quite a lot of messages can be received by some nodes in that malicious users utilize system vulnerabilities to destroy the whole system.
- **Bandwidth in the component of deleting old nodes.** This cost depends on the size of a routing table, the updating frequency of nodes in the routing table.
- **Bandwidth in the component of adding new nodes.** When a node starts to enter Jupiter DHT network, some bandwidth is necessary to find new nodes. When Jupiter DHT network is stable, nodes do not waste too much bandwidth in this component.

V. Designing Efficient Routing Protocols

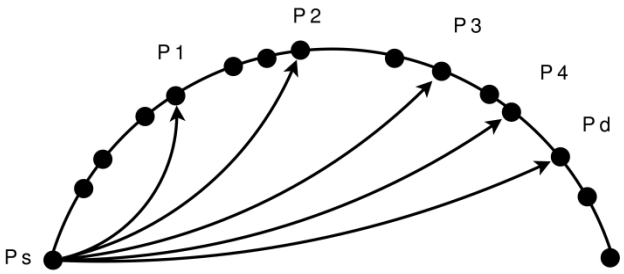
As designing routing protocols, we need to make some choices: routing mode, supporting concurrent routing or not, connection type between two nodes. These choices interact with one another, and any choice is not absolutely right. Thus, we need to present an optimal plan based on these choices.

1. Recursive or Iterative Routing?

There are two basic routings: recursive routing and iterative routing. In recursive routing as shown in Figure 3(a), the initiator sends a routing request to its neighbour, and when the request reaches its destination node, the destination node sends its routing information directly back to the initiator. In iterative routing as shown in Figure 3(b), the initiator acts as a coordinator in the whole routing procedure and sends each routing request. Other nodes receive the routing request and return the nearest node lists to the initiator in their routing tables. The initiator adds the node lists into its local routing results. According to its local routing results, the initiator makes the next decision: routing has been finished and the destination node has been found, or the further routing is necessary and which nodes should be selected to send the next routing request. We present the comparisons of recursive and iterative routing in Table I.



ó I·ú wŜŎdzℕăλØŜ w2dzűλyƎ



óōú LűŜℕI·úλØŜ w2dzűλyƎ

CλƎΦ oΦ w2dzűλyƎ a 2ŘŜ

In recursive routing, each node in the lookup path directly forwards a lookup to the next node, and when the lookup reaches the key's predecessor, the predecessor sends its successor list directly back to the initiator, while in iterative routing the initiator sends a lookup message to each successive node in the lookup path, and waits for the response before proceeding. Iterative routing is easier for the initiator to manage, and the initiator can decide which node is the next one of a current iterative lookup, which is important to support concurrent lookup and security. The initiator can control the granularity of concurrent lookup, i.e., send routing requests to many nodes, which can decrease the influence of invalid nodes in a routing table and reduce routing latency. Therefore, from the standpoint of time-consuming, iterative routing has lower latency than recursive one. However, it is hard for recursive routing to support concurrent lookup. The initiator may send routing requests to many successive nodes, these nodes may further send routing requests to their successive nodes, and so on, which makes lookup messages be increased exponentially. However, this routing converges a small identifier space in the end and incurs that the rear nodes in this routing path receive many repetitive routing requests.

Table 1 Comparisons of recursive and iterative routing

Comparison / Routing	Concurrency	Controllability	Latency	UDP	Security
recursive routing	bad	bad	slow	unfriendly	a bit bad
iterative routing	good	good	quick	friendly	a bit good

Moreover, to some extent, iterative routing can prevent malicious nodes to pollute routing tables. The initiator in iterative routing makes a decision for each hop according to local information. The initiator can make a right judgement from the routing information returned by many neighbours unless its routing table is polluted as bootstrapping. However, the initiator in recursive routing loses routing control after sending a routing request so that any malicious node in this routing procedure can implement routing cheating.

2. Concurrent Routing

We call concurrent routing: a node concurrently sends routing requests to many nodes. Concurrent routing can reduce routing latency that are caused by invalid nodes in that it can send routing requests to many nodes. Each routing has only a path in Chord. Chord cannot support concurrent routing. In each routing selection, Chord utilizes the greediest policy: choosing the nearest node in logic space to route. There are two problems about this policy.

The optimization in logic space cannot guarantee the optimization in routing. Routing time is the most important metric to evaluate routing performance, i.e., how long is it for routing to the destination node. Gummadi et al. [10] discussed the efficiency of routing and came to a conclusion that a routing can be completed within almost equal hops even if any node (not an optimum one) in the same area is chosen in the routing. Therefore, for saving routing time, a node in Jupiter selects several nodes in the same area to be as the next routing nodes according to not logical distance but network distance. This node concurrently sends routing requests to those nodes, which can decrease the probability of choosing nodes that are close in logic space but far in network distance.

The cost waiting time-out messages makes routing slower if the node is invalid, which makes concurrent routing necessary.

Concurrent routing cannot be utilized by Bamboo [9] because Bamboo uses recursive routing. Accordion [29] is another typical DHT, which employs recursive routing. Each Accordion node maintains a “parallelism window” variable that determines the number of copies it forwards of each

received or initiated lookup. When a node initializes a routing request, it marks one of the parallel copies with a “primary” flag which gives that copy high priority. This policy prevents the routing request to be cancelled. However, the mechanisms make the Accordion DHT complicated. Thus, iterative routing is a better choice than recursive routing to support concurrent routing.

3. Connections between Nodes

There are no direct physical links between a node and its neighbours in a DHT. Nodes send UDP messages to confirm if their neighbours are available. Nodes in Chord DHT employ TCP to link one another, while TCP is not suitable to a large scale dynamic network. There are the following reasons: 1) nodes are dynamic and their neighbours are also dynamic. For keeping TCP links between a node and its neighbours, new TCP links have to be constructed and old TCP links have to be destroyed. Too many TCP links result in too much communication cost. 2) TCP is used as a transferring protocol to bring start delay, congestion control and too much time for evaluating overtime. Therefore, TCP links between a node and its neighbours are kept to a minimum, and then routing is finished by these neighbours, e.g., pond [33]. Thus, TCP is impossible to be used in iterative routing and nodes may only use recursive routing. 3) TCP-based congestion control constricts choices of error-handling during designing protocols.

Meanwhile, resources in geo-distributed storage systems are difficultly controlled in the routing layer. DHash [34] used a transfer ring protocol named STP (Striped Transport Protocol) based on UDP. In addition, some TCP-like features are added into STP. A re-implemented TCP in the user layer is utilized in Tapestry [35]. In conclusion, as a transmission mode, TCP has some disadvantages, while UDP-based transmission protocol with constraints is a better choice in geo-distributed storage systems.

VI. Achieving High Security on Routing

1. Importance of Security in Geo-distributed Systems

A distributed system deployed in the Internet must be aware of security. A bad design or security hole can lead geo-distributed systems to go into dangers and even threaten other applications in the Internet in that geo-distributed systems are decentralized so that the systems are difficult to be controlled. We give some basic considerations about security in our system. Our main principle is that the consistency principle must be guaranteed during processing received messages. On the one hand, it can prevent system exceptions, which are brought by incorrect manipulations or network errors. On the other hand, it can also prevent the malicious users that analyze messages to destroy the whole geo-distributed system.

2. Encapsulating Transferred Messages

We use XML to stand for communications among nodes in our system. There are good reasons for using text streams as carriers of network protocols [22]: 1) text streams are very easy for human beings to read, write and edit without specialized tools, which is helpful to catch and debug possible bugs; 2) as carriers of protocols, text streams are useful to upgrade and extend with the maximum degree of freedom. Furthermore, we use XML text streams as carriers of Jupiter protocol, which adequately utilizes the serialization of XML. Each message of Jupiter is an in-memory object, which is serialized and sent.

3. Preventing Dangerous Messages

Zhou et al. [9] found that vulnerability in eMule network, combined with the character of the DNS service, a more serious DDos attack can be launched. This vulnerability is that a node receives PING messages from unknown nodes, and then returns PONG messages to those nodes. In brief, for a request/respond message system: if a node that does not send a request message receives a respond message, this response message is suspicious; if a received message has some invalid bits, this

message must not be processed. Therefore, all the received messages in our Jupiter are checked via Algorithm 3.

Algorithm 3: Preventing a Dangerous Message

Input: Message

1 The received message need to be deserialized;

2 Each bit of the received message is checked;

3 **if** there is an invalid bit in this message **then**

4 **return**;

5 *p* checks the sender and receiver of the message to confirm the receiver;

6 **if** *p is not the receiver* **then**

7 **return**;

8 **if** (*This message is a request one*) && (*p has sent a request message to the sender*) **then**

9 *p* responses this message;

10 **return**;

All the received messages need to be deserialized so that any error message is discarded (line 1). Each bit of a received message is checked if it is valid such that any message with invalid bits is also discarded (lines 2-4). A node checks the sender and receiver of the received message to confirm if it is the receiver, if yes, it response this message, in addition, it needs to check if it has sent a request message to the sender (lines 5-10). Finally, all the messages through the checking procedures can be processed.

VII. Experiments

1. Data Sets

Since there is no public data set for geo-distributed storage system, we choose the data sets coming from shared resources of two weeks in a real P2P system: Maze, which is one of the largest P2P systems over CERNET (China Education and Research Network), with an average of 20K simultaneously online users, and it has been renamed as AmazingStore. There are 7,565 active users

and 30,001,293 files totally. We pre-process the shared resources, e.g., removing shared resources in a system disk (e.g., C:\), WINDOWS installation directory (e.g., C:\WINDOWS) and programs installation directory (e.g., C:\Program Files), which are shared by free riders.

2. Experimental Setup

We have prototyped Jupiter consisting of around 6K lines of C++ code, deployed it on a small network of 20 Linux machines withRedhatELAS4 across sites, where10 machines are configured with Intel(R) Xeon(TM) 2.80GHz CPU×4 and 4GB memory, 5 machines are configured with Intel(R) Xeon(R) E5310 @ 1.60GHz CPU×8 and 8GB memory, and 5 machines are configured with Intel(R) Xeon(TM) 2.80GHz CPU×4 and 8GB memory. In our experiments, there are many simulated nodes in each computer. We equip each node with real shared files from a user in Maze. Each node is an independent process to monitor its port and records its trace. All the experimental results are based on analyzing nodes' traces. The running time of each experiment is forty minutes.

3. Experimental Results

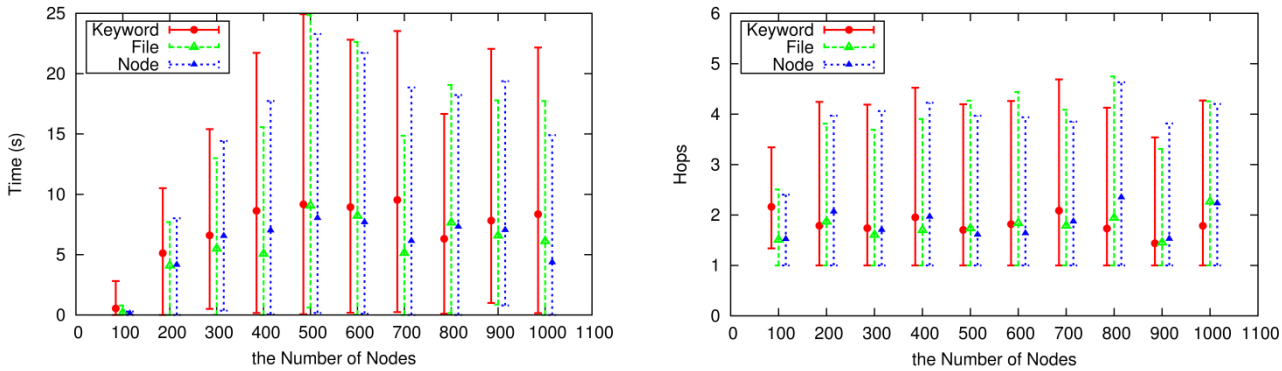
This section presents experimental results to evaluate the efficiency, robustness and consistency of Jupiter.

A. Efficiency

Efficiency is an important metric in the routing layer, which affects upper applications. We evaluate the efficiency of lookup when Jupiter is almost stable. There are two metrics: 1) average lookup time which measures the real running efficiency of lookup and 2) average lookup hops which measures if routing results can be returned in O(*logN*) steps. We measure routing efficiency when any node performs a search for a node, a file or a keyword. Here the minimum and maximum values are the mean of values of the lower and upper ten percent of metrics respectively.

Figure 4(a) shows the results of average, minimum and maximum lookup time. As seen from Figure 4(a), the time of average lookup increases slightly with the increment of network size. The minimum time of lookup is almost the same in different network sizes. For the maximum time of lookup, there

are two situations: 1) the routing table of a node does not have enough nodes when the node is just online; 2) a node is offline, which incurs there is a routing timeout. Figure 4(b) shows the results of the average, minimum and maximum number of lookup hops. As can be seen from Figure 4(b), the average number of lookup hops also increases slightly with the increment of network size. The hops of average lookup are almost two in that the network size is not huge. The hops of minimum lookup are almost the same: one. That is to say that a node can find the destination within one hop under the best conditions. For the maximum number of lookup hops, there are similar reasons to the maximum time of lookup.



(a) Lookup Time (b) Lookup Hops

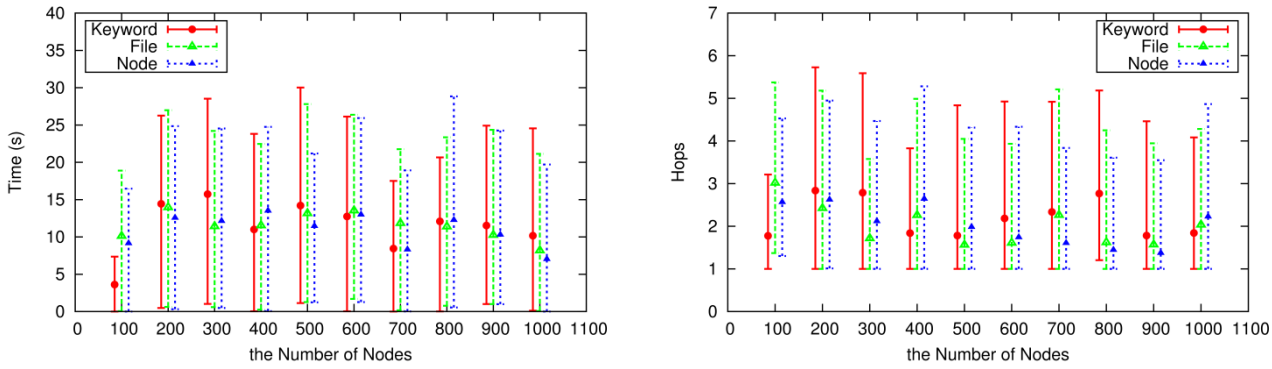
Fig. 4. Efficiency Performance with Different Network Size in Jupiter

B. Robustness

Robustness is also an important metric of routing. High churn is inevitable for nodes in geo-distributed storage systems. Therefore, we need to know if Jupiter can be still on work in this situation. In this experiment, five percent of online nodes are stopped at the moment of the tenth minute. Jupiter comes into a stable status at that moment. Five minutes later, another five percent of online nodes are stopped and existing offline nodes join the Jupiter system again. This procedure is repeated for five times.

Compared with Figure 4(a) and Figure 4(b), Figure 5(a) and Figure 5(b)illustrate that Jupiter achieves a little bit worse results under churn conditions than those under almost stable conditions, but their differences are not obvious. Routing messages are delayed because of nodes' departures,

which degrade lookup performance. Subsequently, lookup performance rises gradually as there are increasingly valid nodes in the routing tables of the remaining nodes in Jupiter due to its routing table maintenance mechanisms.



(a) Lookup Time (b) Lookup Hops

Fig. 5. Robustness Performance with Different Network Size in Jupiter

C. Consistency

In this experiment, we evaluate the consistency of routing. We call consistency: when there are high churn hates in a distributed system, whether any two nodes can return the same routing results for the same routing destination. We use how many nodes are overlapped among K returned nodes to measure the consistency as shown in the following formula, for any two queries for the same file or keyword from two different nodes.

$$Consistency = \frac{S_A \cap S_B}{S_A \cup S_B}$$

where S_A and S_B are the returned sets of nodes for node A and B respectively.

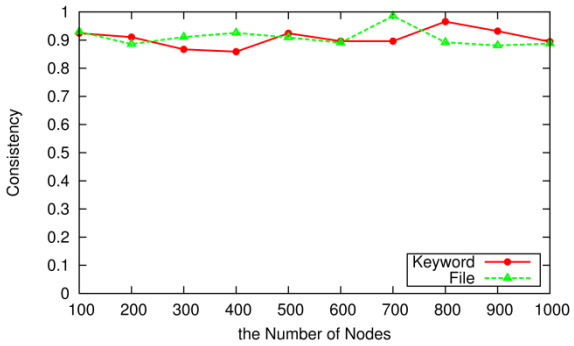
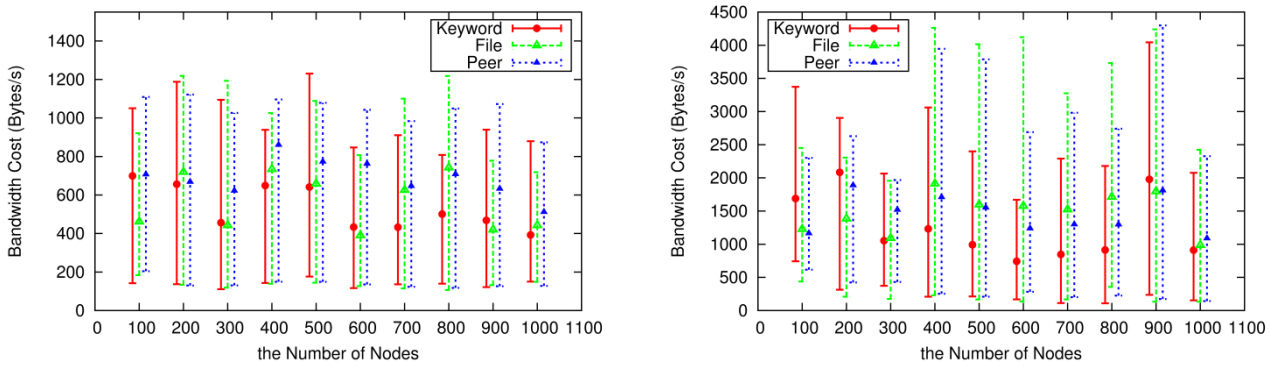


Fig. 6. Consistency with Different Network Size in Jupiter

As shown in Figure 6, the minimum and maximum value of consistency are 85.86% and 96.54% respectively when performing searches for keywords and the numbers of nodes are 400 and 800 respectively. They are 88.11% and 98.56% respectively when performing searches for files and the number of nodes are 900 and 700 respectively. When any two nodes in Jupiter lookup the same file or keyword, there are only one or two different nodes in returned results at the beginnings and the other nodes are the same, which reflects the excellent convergence of Jupiter.

D. Bandwidth

In this experiment, we are aware of bandwidth in routing. We can measure bandwidth cost with a metric: sent bytes per second. An active node initializes a random lookup every half a minute, and an inactive node only maintains its routing table and responds the requests from other nodes. Here the minimum and maximum bandwidth consumptions are the mean of values of the lower and upper ten percent of bandwidth consumptions respectively. As shown in Figure 7(a), for an inactive node, its bandwidth consumption responding other requests is low, even the maximum bandwidth consumption only reaches 1.2KB/s as performing searches for files (the number of nodes is 800). Maintaining a routing table consumes some bandwidth, but this bandwidth consumption does not vary much with the increment of network size. For an active node, its maintenance cost rises in that it continuously sends routing requests, which incur that its routing table is larger than that of an inactive node. Therefore, for its routing table, the maintenance cost rises. In a word, bandwidth consumption of a node, even an active node in Jupiter, is acceptable: average bandwidth consumption of about 1.5KB/s is no problem for a node in the current Internet.



(a) Inactive Nodes (b) Active Nodes

Fig. 7. Bandwidth Cost with Different Network Size in Jupiter

VIII. Conclusion

In this paper, we present Jupiter, a DHT middleware system for building geo-distributed storage systems. Jupiter provides robust and efficient routing mechanisms under geo-distributed environments. The key innovation is the integration of two concepts: robustness and efficiency. We have prototyped it and deployed it on a network of Linux machines. We confirm the effectiveness and efficiency of Jupiter by conducting an extensive performance benchmark measured by efficiency, robustness and consistency. In future, to deploy Concurrent Regeneration codes with Local reconstruction (CRL) [36] in Jupiter is a possible work to reduce storage space consumption.

References

[1] Q. Xu, R. V. Arumugam, K. L. Yong, and S. Mahadevan, “DROP: Facilitating distributed metadata management in EB-scale storage systems,” in IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA. IEEE, 2013, pp. 1–10.

[2] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, andS. Quinlan, “Availability in globally distributed storage systems,” in 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings, 2010, pp. 61–74.

[3] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: facebook's distributed data store for the social graph," in 2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013, 2013, pp. 49–60.

[4] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in SIGCOMM, 2001, pp. 149–160.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, 2007, pp. 205–220.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," Operating Systems Review, vol. 44, no. 2, pp. 35–40, 2010.

[7] Q. Xu, H. T. Shen, Y. Dai, B. Cui, and X. Zhou, "Achieving effective multi-term queries for fast DHT information retrieval," in Proceedings of the 9th International Conference on Web Information Systems Engineering (WISE) (2008), Springer-Verlag, pp. 20–35.

[8] M. J. Freedman, "Experiences with coral CDN: A five-year operational view," in Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA, 2010, pp. 95–110.

[9] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004, Boston Marriott Copley Place, Boston, MA, USA, 2004, pp. 127–140.

[10] P. K. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of DHT routing geometry on resilience and proximity," in SIGCOMM, 2003, pp. 381–394.

[11] Q. Xu, R. V. Arumugam, K. L. Yong, and S. Mahadevan,"Efficient and scalable metadata management in eb-scale file systems," IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 11, pp. 2840–2850, 2014.

[12] Q. Xu, L. Zhao, M. Xiao, A. Liu, and Y. Dai, "YuruBackup: A space-efficient and highly scalable incremental backup system in the cloud," International Journal of Parallel Programming, vol.43, no.3, pp.316– 338, 2015.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, 2012, pp. 261–264.

[14] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, 2011, pp. 223–234.

[15] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete,"MDCC: multi-data center consistency," in Eighth Eurosys Conference 2013, Prague, Czech Republic, April 14-17, 2013, Z. Hanz´alek, H. H¨artig, M. Castro, and M. F. Kaashoek, Eds. ACM, 2013, pp. 113–126.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: yahoo!'s hosted data serving platform," PVLDB, vol.1,no.2,pp.1277– 1288, 2008.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-distributed storage," in Proceedings of the 10th USENIX Symposium on

Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, 2013, pp. 313–328.

[18] ——, “Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS,” in Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011, 2011, pp. 401–416.

[19] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional storage for geo-distributed systems,” in Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011, 2011, pp. 385–400.

[20] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguic¸a, and R. Rodrigues, “Making geo-distributed systems fast as possible, consistent when necessary,” in 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012, 2012, pp. 265–278.

[21] N. Tran, M. Aguilera, and M. Balakrishnan, “Online migration for geo-distributed storage systems,” in USENIX ATC, 2011.

[22] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, “Volley: Automated data placement for geo-distributed cloud services,” in NSDI, 2010.

[23] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, “Consistency-based service level agreements for cloud storage,” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 309–324.

[24] M. S. Ardekani and D. B. Terry, “A self-configurable geo-replicated cloud storage system,” in the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014, pp. 367–381,

[25] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. Lau, “Scaling social media applications into geo-distributed clouds,” in INFOCOM, 2012.

[26] M. Castro, M. Costa, and A. I. T. Rowstron, “Performance and dependability of structured peer-to-peer overlays,” in 2004 International Conference on Dependable Systems and Networks (DSN2004),28June -1 July 2004, Florence, Italy, Proceedings, 2004, pp. 9–18.

[27] S. Deb, P. Linga, R. Rastogi, and A. Srinivasan, “Accelerating lookups in P2P systems using peer caching,” in Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 712, 2008, Canc´exico, G. Alonso, J. A. Blakeley, and A. L. P. un, M´Chen, Eds. IEEE, 2008, pp. 1003–1012.

[28] B. Awerbuchand C. Scheideler, “Towards a scalable and robust DHT,” in SPAA2006: Proceedings of the18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 -August 2, 2006, P. B. Gibbons and U. Vishkin, Eds. ACM, 2006, pp. 318–327.

[29] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, “Bandwidth-efficient management of DHT routing tables,” in 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings., 2005.

[30] A. R. Bharambe, M. Agrawal, and S. Seshan, “Mercury: supporting scalable multi-attribute range queries,” in Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA, 2004, pp. 353–366.

[31] Q. Xu, H. T. Shen, B. Cui, X. Hou, and Y. Dai, “A novel content distribution mechanism in DHT networks,” in Networking,2009, pp.742– 755.

[32] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, “Sharing the data center network,” in Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI, Boston, MA, USA, March 30 - April 1, 2011.

[33] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The ocean store prototype," in Proceedings of the FAST Conference on File and Storage Technologies, March 31 - April 2, 2003, San Francisco, California, USA, pp. 1-14.

[34] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a DHT for low latency and high throughput," in NSDI, 2004, pp. 85–98.

[35] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," University of California at Berkeley, Computer Science Department, Tech. Rep., 2001, UCB/CSD011141.

[36] Q. Xu, W. Xi, K. L. Yong, and C. Jin, "Concurrent regeneration code with local reconstruction in distributed storage systems," in The 9th International Conference on Multimedia and Ubiquitous Engineering, 2015, pp. 415–422.