J Intell Manuf (2012) 23:955–963
DOI 10.1007/s10845-010-0496-y

Received: 22 April 2010 / Accepted: 3 December 2010 / Published online: 25 December 2010

Ubiquitous storage services oriented to social networks need timely responses to users' requests and high scalability with the increasingly size of storage systems. In this paper, we propose a virtual node based ring-like storage system architecture and data placement scheme for delay-sensitive ubiquitous applications. For improving the fault-tolerant ability of the storage service, we design a data replication approach based on a quorum-fault-tolerant state machine protocol. Moreover, we investigate a three-layered messaging protocol based on the heartbeat mechanism to detect the node failure and maintain the ring topology. The theoretic analysis and experiment results demonstrate the feasibility and high operation efficiency of our ubiquitous storage scheme.

Storage service · Ubiquitous computing · Scalability · Fault tolerance

Ubiquitous computing is creating ambient intelligence and transparent services while people moving around. As a new computing paradigm, essentially, ubiquitous computing environment needs to provide desired information and services for people in the specified QoS requirements anytime anywhere, i.e., data and services are at the center of ubiquitous applications. As a result, a time-efficient storage service is one of the core bases of ubiquitous IT applications, e.g., social network services like Facebook, and YouTube. These networked storage services store peta bytes of digital photos, blog articles and video clips, and build online communities for people with common interests. People have strict requirements on such ubiquitous services, especially on access time, location independence and system scalability. Undesired performance degradation will impact the user experience and accordingly damage the value chain of the storage industry (Celik et al. 2010; Park et al. 2009).

For such ubiquitous storage services, it is an important and challenging issue how to access data efficiently, where data is usually distributed in different networked nodes in the decentralized way because of increasingly large-scale applications and fault-tolerant requirements. More specifically, these services are built over an infrastructure with tens of thousands of servers and network components (e.g., routers). In summary, ubiquitous storage services should have the following characteristics (Gholami and Zandieh 2009; Greer et al. 2008).

- *Low access delay*. Users can access the storage services and finish their operations before the expected deadline.
- *High scalability*. The storage services need to be highly scalable to support the continuous growth of storage requirements with the increase of user activities and data, keeping the constant performance behavior.
- *Fault tolerance*. A storage system should work well in face of failures of some nodes, expect a little performance degradation.

Existing researches on distributed storage have made great progress (Chockler et al. 2009; Jepson 2004; Marcelm-Jimenez et al. 2006; Ramakrishnan and Emer 1989; Yi and Shin 2010; Zeng and Veeravalli 2008), however, their solutions to data access cannot still meet the on-line interac-

F. Tang (✉) · M. Guo · Q. Wang
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, 200240 Shanghai, China
e-mail: tang-fl@cs.sjtu.edu.cn

F. Tang · S. Guo
School of Computer Science and Engineering,
The University of Aizu, Fukushima 965-8580, Japan

tion requirements from ubiquitous applications. Peer-to-Peer (P2P) based distributed storage (Yiu et al. 2007, 2009) is an emerging and promising technology for content sharing applications because of its inherent scalability, fault tolerance, and high performance. These P2P applications provide a lookup service similar to the hash table and support dynamic membership. Current structured P2P overlay network lookup algorithms have been designed to maintain a small amount of routing information, typically in the time complexity $O(\log N)$, where N is the number of nodes in the storage system. But these storage systems built on P2P need the multi-hop lookup so that the communication delay is still too much, say $O(\log N)$ for Chord (Karger et al. 1997). As a result, these P2P based storage services can not be directly applied to delay-sensitive services.

The objective of this paper is to provide an efficient and scalable storage service for future ubiquitous IT society, with *O(1) lookup performance,* i.e., request for a data item needs only one hop. Data is partitioned using the consistent hashing and replicated at several nodes.

The main contributions of this paper are described as follows.

(1) We propose a *virtual node* based ring-like storage model and a data placement scheme. A physical node usually provides a set of services, each with a unique port. We call such a port as a *virtual node*. A virtual node is marked by the IP address of the physical node and the port number. We execute the MD5 algorithm to generate the virtual node's address in the storage ring. Data is mapped and distributed in terms of the consistent hash function, whose randomicity and uniformity guarantee the load balance among networked storage nodes.
(2) For improving the fault-tolerant ability of the storage service, we design a data replication approach based on a quorum-fault-tolerant state machine protocol. If the quorum is N, the storage service is able to recover from N-1 replicated node failure. The algorithm replicates key data to N-1 appropriate nodes with the goal of online data access.
(3) We investigate the node failure detection and ring topology maintenance. The joining and failure of a node can be firstly detected by its subsequent through the heartbeat mechanism. We propose a three-layered messaging protocol based on the heartbeat mechanism. Any joining or failure can be spread to all the storage nodes in at most 4 hops, reducing the data access time significantly.

The remainder of this paper is organized as follows. In the next section, we review related work. We, then, present the virtual node based storage model and data placement scheme in "Storage system model and data placement". The data replication approach is described in "Fault tolerance and data replication approach". "Node failure detection and topology maintenance" presents the node failure detection and ring topology maintenance mechanism. The implementation and performance evaluation are presented in "Performance analysis and experiments evaluation". Finally, we conclude the paper in "Conclusions".

Consistent hashing was introduced in Karger et al. (1997) and used for distributed networks to decrease or eliminate the problem of hot spots in the network. Many P2P systems have actively addressed the problem of data storage and distribution. Structured overlay networks, such as Chord (Stoica et al. 2001), Pastry (Rowstron and Druschel 2001), and Tapestry (Zhao et al. 2004), employ consistent hashing to map objects to nodes and attain $O(\log N)$ worst-case lookup performance.

DHT (distributed hash table) based lookup protocols impose strict structures on the overlay networks. Some recently DHT proposals support $O(1)$ lookup performance. Kelips (Gupta et al. 2003) provides $O(1)$ lookup performance by dividing the network into $O(n)$ groups and using a gossip protocol to propagate event notifications. Beehive (Ramasubramanian and Sirer 2004) achieves high performance of serving queries in less than one hop on average and low storage overhead and bandwidth overhead by exploiting the structure of the underlying overly and minimizing the amount of replicated data in the system. Moreover, a dynamic failure detector for P2P storage system was reported in Wan et al. (2009). The detector, which combines heartbeat strategy with unbiased grey prediction model, can improve the failure detection quality of service (QoS) according to the application needs and network environment changes. For improving the search performance and balancing storage load in a P2P network Yamamoto et al. (2006), proposed two replication methods for balancing the load on the storages distributed over P2P networks while limiting the degradation of the search performance within an acceptable level. Compared to these DHT based membership protocols, our system introduces a hierarchical structure to the membership and failure detection protocols for rapidly notifying membership change events but keeping protocol overhead low.

There has been a lot of researches on highly scalable distributed file systems and distributed databases. Distributed file systems usually support hierarchical namespaces. The Coda file system (Satyanarayanan et al. 1990) provides resiliency to server and network failures using of replication and supports disconnected operations at the cost of consistency. Conflicts are processed using conflict resolution procedures and the Coda file system allows disconnected operations. Tian et al. (2007) developed a data placement

scheme called similar-MTTF-MTTR placement, which takes into consideration differences in pees' dynamic characteristics. In this paper, the authors also present a fine-gained analysis model for short-term data availability calculation. Compared to these systems, our system does not support hierarchical namespace or relational schema among data items. Our storage system is designed for latency sensitive applications because it provides $O(1)$ lookup performance at the cost of the full membership memory overhead. The Dynamo key-value storage system is most related to our work but the Dynamo system can only scale up to several hundreds of nodes. Most differently, we introduce hierarchical extensions to the membership protocol to make the system much more scalable.

[illegible]

We target on the storage services oriented to social networks, which have the following characteristics.

- Data is stored in the key-value way. For example, a photo has a unique ID and the ID is associated with a photo file.
- Users require timely even real-time responses.
- The amount of data is dynamic but grows up increasingly. Some users maybe delete parts of their uploaded data from time to time, however, the total data keeps on increasing in the long term.

The above features present the *low access delay*, *high scalability* and *enough reliability* requirements to storage services. We concentrate on these three metrics in designing our storage scheme.

Assumptions

Our storage system consists of a set of servers (storage nodes) which are responsible for holding replicas of data items. A data item in the storage system is uniquely identified with a key. Simple read and write operations can be performed on data items, whereas no relational schema is kept for multiple data items. We assume an asynchronous distributed system in which processes may fail from time to time. Nodes are interconnected by a network, that is, given a host IP address, processes on any other host can (attempt to) communicate with processes on the former directly by sending messages.

We assume dynamic membership behaviors with storage nodes in the system. We can add new storage nodes into the system any time to increase the storage capacity and some nodes may leave the system temporarily or permanently due to different operational reasons. The system employs a full membership model, where each storage node is aware of other peers in the storage system and the data they store. The operation environment of the storage system is assumed to be safe and there are no security related requirements such as authentication and authorization.

Virtual node based ring-like storage model

One of our design goals is the ability to scale up incrementally, i.e., the system can be scaled up by adding a new storage host at a time without degrading the system performance observably. At the same time, after a new storage node is added in the system, the original data will be moved as little as possible and data is evenly distributed in all storage nodes. To achieve high scalability, the storage system favors a decentralized P2P architecture over centralized control. Every storage node in the system takes the same set of responsibilities as its peers. There is no special node which has extra responsibility in the system.

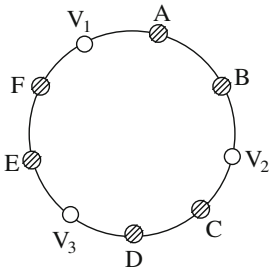
Data is stored in networked nodes. Each node may accommodate a set of different services that run on different ports. We organize the storage system in the *virtual node* (VN). A VN is a specified service running in a physical host, and marked with the combination of the node's IP address and the service's port. We store data in the key-value way, where each data is associated a unique key (called ID). To describe how to organize storage nodes, we firstly define the following annotations.

$K(n)$: the VN n's label, which is the character string *IP address* appended by *service port.*
$H(n)$: the address of the VN n in the storage ring.
$H(n)$ is generated by the MD5 function such that $H(n) = \text{MD5}(K(n)) \bmod 2^m$, where m is the digit number. It is a hash value within the range of m digits.

We organize the storage system using the virtual nodes. According to such the scheme, all virtual nodes are distributed in a ring. Each address is associated with a specified location in the ring. The address space is 2^m(e.g., m=128), arranged from 0 to 2^m–1 clockwisely. Data is actually stored in these distributed virtual nodes.

We assign the number of VNs that run in a physical node based on its computing and storage capacity. Then, we execute MD5 algorithm to generate the $H(n)$ according to the label of the virtual node n. MD5 algorithm ensures that the addresses of VNs are evenly scattered in the storage ring.

Figure 1 depicts the logical view of virtual nodes in the storage system. For example, node D is responsible for storing data items associated with keys in the range $(C, D]$. To add a set of new storage hosts into the system, we randomly choose a number of locations on the ring and assign these locations to the host set V. With high possibility, the key ranges assigned to the V are randomly scattered on the ring. At this moment some existing nodes are responsible for the



1 Virtual node based *ring-like* storage model

key ranges allocated to the V and need to transfer data items associated with these keys to the V. For example, in Fig. 1, if the new storage host set V = {V_1, V_2, V_3} is to be added to the system (i.e., virtual nodes V_1, V_2, and V_3 in the ring), the existing nodes A, C, and E have to transfer an appropriate set of data items to the V_1, V_2, and V_3, respectively.

Our virtual node based storage scheme has the following advantages (Decandia et al. 2007).

- *Balanced data storage when some networked nodes fail*. In our storage model, a physical node provides a set of virtual nodes that are evenly distributed in the ring. If a physical storage node becomes unavailable, the data that is stored in its individual virtual nodes will be evenly dispersed among the other available storage hosts, which makes the storage system balanced.
- *Balanced data storage when new networked nodes join*. Based the same reason above, when the storage host becomes available again or a new storage host joins, its virtual nodes distributed evenly accepts a roughly same amount of load from other available storage hosts.
- *Balance of heterogeneous systems*. With the development of a storage system, hosts involved in the storage system often are heterogeneous, with different capacities. In our storage scheme, however, the number of virtual nodes that a storage host provides is assigned based on its computing and storage capacity. The more powerful a host is, the more storage services it will provide. As a result, the whole storage system works in a balanced way.

Data placement

Data is mapped to corresponding virtual nodes using the consistent hash function. In key-value storage system, each data is associated with a unique key k(data), e.g., the ID of a photo. We use the function $H(k(\text{data}))$ to generate the storage address p(data) of the data such that $p(\text{data})=H(k(\text{data}))$. A possible problem is that p(data) does not always equal to the node address $H(n)$. The following approach solves how to map data to corresponding virtual nodes.

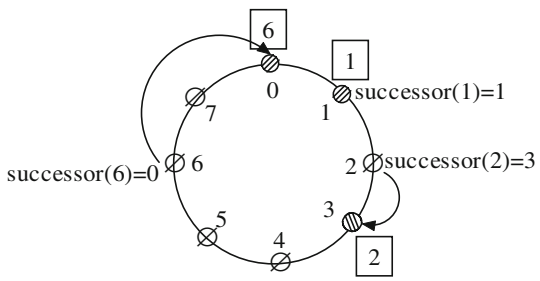
Let SN(data) be a virtual node that actually store the data. The SN(data) is the first existing virtual node with address more than p(data). So, we compare the p(data) with the address of virtual nodes, and store the data in the SN(data). As a result, each virtual node stores the data addressed between the virtual node's predecessor and the virtual node itself.

We define the operation for generating the hash value as *successor()*, whose returning result is the label of the virtual node. The storage system uses a variant of consistent hashing (Stoica et al. 2001) as the partitioning scheme to map keys to storage nodes. Responsibilities for maintaining the mapping from names to values are distributed among the storage nodes in a way that a change in the set of participants causes a minimal amount of disruption. This allows the system to scale up to extremely large numbers of nodes and to handle continual node arrivals and departures. The system runs with a 128-bit space where the output range of the hashing function is viewed as a ring or circular space, wrapping around to the smallest hashing value after the largest hashing value. Each storage host in the system is assigned multiple random positions (virtual nodes) on the ring. Each data item is identified with a unique key and assigned to a virtual node by hashing the unique key using the MD5 to yield a position on the ring and walking clockwise along the ring until the first virtual node position. Each virtual node is responsible for the range between itself and its predecessor on the ring.

Figure 2 illustrates a storage ring with the space 2^3(m = 3), where three positions, 0,1 and 3, are placed at corresponding VNs. The data with p(data)=1 is stored in the VN_1, i.e., successor(1) = 1. For the data object with p(data)=2. There is not a virtual node in the position 2 so that the data is stored in the virtual node with position=3, i.e., successor(2)=3. Similarly, the data object with p(data)=6 will be stored in the VN_0 such that successor(6)=0. In summary, each VN stores the data objects with labels between the last virtual node and itself. For example, VN_0 stores data objects with label $\in$ {4,5, 6,7,0}, VN_1 stores the data object with label=1, and similarly VN_3 stores the data objects with label $\in$ {2,3}. If a new VN_5 is inserted in the storage system, the data objects with label=4 and 5 will be removed to the VN_5.

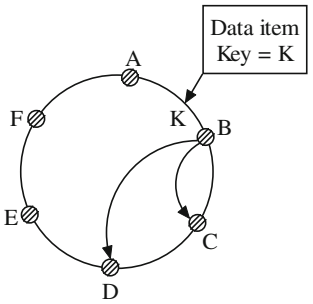
Service interface

The storage service exposes a simple interface to the client and manages data items associated with keys through the



2 Data placement in the storage ring

three operations: *PUT*, *GET*, and *DELETE*. The *PUT(key, data item)* operation determines where the data item should be placed based on the associated key and stores the data item to the disk. The *GET(key)* operation retrieves and fetches the data item associated with the key and the *DELETE(key)* operation removes the data item associated with the key from the storage system. The client accesses data in the storage system through a client library. The client library is ware of the partitioning of the storage system and directly routes client requests to appropriate nodes.



3 Data replication process ($N = 3$)

Social storage services involve large numbers of personal data, e.g., precious photos. When the storage system involves hundreds of physical servers, the servers maybe fail from time to time, losing parts of such the precious data. In this section, we present how to improve the reliability and availability of our storage service in face of node and service failures.

Distributed storage systems use replication to cope with the loss of data, storing data in multiple basic storage units—disks or servers. Such systems provide high availability: The stored data should remain available at least whenever any single server or disk fails; sometimes they tolerate more failures (Chockler et al. 2009).

Replication can improve the data availability through multiple copies. If n independent nodes keep the same data and each node may fail in the probability p, the data will be available in the probability $1 - p^n$. We use the quorum-based replication through a state machine protocol (Satyanarayanan et al. 1990). If the quorum is N, the storage service is able to restore from N-1 replicated node failure. The algorithm replicates key data to N-1 appropriate nodes to improve the system fault-tolerance ability.

Each data item is replicated at N hosts which is configured before the system starts up. For each key, the data item associated with the key is assigned to a virtual node which is the first virtual node from the hash value of the key in the clockwise direction on the ring. We call this virtual node as the *coordinator*. The coordinator stores data items associated with keys within its range and is in charge of replicating these data items at N-1 clockwise successive physical nodes on the ring based on their IP addresses. In Fig. 3, in addition to locally storing the data item identified with the key K, virtual node B also replicates the data item associated with K at virtual nodes C and D (assuming N=3). On the other hand, virtual node D is responsible for storing data items identified with keys that fall in the range $(C, D]$ as well as replicating data items identified with keys in the ranges $(A, B]$ and $(B, C]$.

A storage system involves hundreds even thousands of nodes, where any node may join, fail or exit out of the system. A time-efficient failure detection and topology recovery mechanism needs to be carefully designed.

Global routing

As mentioned above, the most important performance metrics of social network services is the response time. Most of exiting P2P systems cannot provide on-line lookup and communication. For example, the lookup time in Chord and Pastry systems is $O(\sqrt{N})$, where N is the total number of nodes. When there are hundreds of storage nodes, such lookup performance cannot satisfy the basic demands of social storage service. A good storage service oriented social services has to respond users as quick as possible, and at the same time ensure the response time stable even existing nodes fail or new nodes join the system.

Our scheme is to let each storage node maintains a routing table keeping routing information to all other nodes, including IP address, service port and position in the storage ring. In a stable topology, all nodes have the same routing tables. As a result, for any lookup, any node can forward the a lookup to the destination node in one hop, i.e., our storage service always keep $O(1)$ of time complexity.

Global routing information will be updated whenever a node fails or exits out of the system. Specifically, when a new node is added in the storage system, the new node firstly requests the current routing information from its subsequent node. From then on, the information of the new node will be propagated over the whole system. The propagation algorithm will be presented in the next subsection.

Membership and failure detection

The joining and failure of a node can be firstly detected by its subsequent through the heartbeat mechanism. We propose a three-layered messaging protocol based on the heartbeat

mechanism to maintain the accurate routing information for dynamical topology. Any joining or failure can be spread to all the storage nodes in at most 4 hops, reducing the data access time significantly.

The storage system employs a complete membership model to minimize the lookup latency, that is, each storage node is aware of every other peer in the system and the data it stores so that every node can route a query to the appropriate node that stores the requested data item with $O(1)$ lookup cost. To maintain a full membership and routing information, every storage node adopts a gossip-based membership and failure detection protocol and gossips node arrival and departure events with other nodes in the system. However, our storage system is designed to scale to extremely large system, i.e., a system with tens of thousands of nodes, and this is not trivial because of the overheads of maintaining the routing information. The overheads grow as the system scales up and mainly consist of two parts: the background communication bandwidth usage and the full membership soft state memory usage.

Our hierarchical gossip based membership protocol relies on the proper functions of the group leaders and the subgroup leaders. When a group leader or subgroup leader fails, its successor and predecessor can detect the failure and bootstrap the new group or subgroup leader according to the membership soft state. If a group leader's neighbors find the group leader fails, they tell the closest node to the group boundary to become the new group leader and the new group leader notifies its group members and other group leaders in the system of the membership change. If a subgroup leader fails, its neighbors bootstrap a new subgroup leader, the new subgroup leader notifies its group leader, and the group leader notifies other group members of the event. It takes some time for new nodes to take over the roles of group or subgroup leaders and during this period of time events of node departure or arrival may be missed by the system. As a result, some membership changes will not be notified at all nodes. This does not make a query incorrect but inefficient, i.e., the query can not be bounded at $O(1)$ lookup latency. Since each node carefully maintains correct pointers of its predecessor and successor, any lookup can always be routed to the correct node. Moreover, lookups are also used to eliminate the inconsistency of membership soft states. When a query (encoded in PUT, GET, or DELETE operations) for a data item which is supposed to be stored in node n fails to find the data item on n, the client code will asks one of the group leaders to disseminate the event to other nodes. The notification propagation process is like the process of an ordinary membership change event.

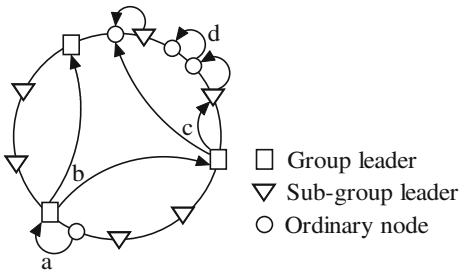
The correctness of our membership protocol is based on the fact that every node correctly maintains its successor and predecessor pointers. Even if the full membership soft states of some nodes contains errors, the query will eventually be routed to the responsible node and the inaccurate membership soft states will be corrected.

Group-based routing table updating

If a node fails or exits the storage system, its subsequent node does not receive the heartbeat message during the last consecutive periods. Every node in the system periodically sends keep-alive messages to its predecessor and successor nodes on the ring after joining the system. The predecessor node p checks if the node sending the keep-alive message, say n, is indeed its successor, and if n is not p's successor, p notifies n of the existence of another node between them. Similarly the successor node of n also checks if n is indeed its predecessor. If either the predecessor or the successor of a node stops sending the keep-alive messages for a period of time, the node declares that its neighbor is dead. Membership change events, nodes joining and leaving the system, are detected by the neighbors at first.

We let every node be aware of every membership change with low propagation delay and low bandwidth consumption. This strategy divides the hash circular space into Q equally sized partitions called *groups*. Since consistent hashing has the virtual nodes uniformly distributed on the hash ring, every group has roughly the same amount of nodes and henceforth is responsible for roughly the same amount of load. Every group has a *group leader* which is the first node from the start of a boundary, but when there is a new node joining the system even closer to the boundary than the current group leader in a group, the group leader does not change until the current group leader is down. Every node in the system knows that which nodes are currently group leaders and when a group leader is declared dead the node which is closest to the boundary becomes the new group leader. Similarly, each group in the system is divided into R equally sized *subgroups*. Each subgroup has a *subgroup leader* which is chosen in the same way as a group leader. When a new node joins the system, it first learns the full membership from its predecessor or successor node. Note that this three-level hierarchical scheme does not violate the symmetry principle mentioned in "Virtual node based ring-like storage model" because every node is capable of becoming a group leader or a subgroup leader, and groups and subgroups are divided in terms of the dissemination of membership change messages instead of the ability to serve client requests.

Figure 4 shows how a membership change event is propagated in the system. When a node detects a new node joins the system or an existing node is dead, it sends a notification to its group leader. After receiving a membership change notification, the group leader notifies other group leaders immediately. Each group leader notifies subgroup leaders in its group about the membership change when receiving the notification. A subgroup leader then notifies its predecessor node and



4 Flows of membership change event notifications. (*a*) An ordinary node detects the event and notifies the group leader. (*b*) The group leader multicasts the notification to other group leaders. (*c*) Each group leader multicasts the notification to the subgroup leaders in its group. (*d*) The notification is brought forth to the predecessor and successor of the subgroup leader. Ordinary nodes then notify their successors or predecessors in one direction

then an ordinary node propagates the message to its predecessor because the message is from the successor. Note that the membership change notifications are delivered in one direction on the ring. Besides, the messages propagated in a subgroup are piggy-backed on the keep-alive messages. In this way, all the nodes receive all the notifications of membership change events. An event notification stops at the boundary of a subgroup and there is no message duplication in the system.

[illegible]

We evaluate our storage scheme through theoretic analysis and simulations, concentrating on the scalability and response time.

Scalability analysis

Our storage service can respond to the user query in the time complexity $O(1)$. In particular, such query performance keeps constant as the number of storage nodes increases. From the flowing description, we can find that network bandwidth consumed by our storage scheme is also scalable.

In our routing algorithm, a VN maintains routing information to all other VNs. So, we first analyze the cost of maintaining routing information and its scalability. As described above, our routing protocol needs to detect the node state through messaging a packet. An ideal solution should consume less network bandwidth and get the feedback as soon as possible. We define a metrics BDP (the product of bandwidth and delay) to measure the performance of routing setting and updating, such that BDP=$B \times T_{\text{detection}}$, where B is the consumed bandwidth and $T_{\text{detection}}$ denotes the response delay.

We analyze and compare our routing algorithm with other heartbeat-based routing updating protocols in terms of BDP. Let the size of a heartbeat packet be m, the number of storage nodes be n, consumed bandwidth for routing updating be B. We define a node failure when its subsequent node does not receive the heartbeat packet in consecutive p periods.

(1) Heartbeat based all-to-all multicast
The detection frequency in all-to-all multicast is $f = \frac{B}{m \times n^2}$. So, the time of node detection is $T_{\text{detection}} = \frac{p}{f} = \frac{p \times m \times n^2}{B}$. Accordingly, the BDP of the heartbeat based all-to-all multicast scheme is BDP $= B \times T_{\text{detection}} = p \times m \times n^2 = O(n^2)$

(2) Heartbeat based gossip
The detection frequency in Gossip is $f = \frac{B}{m \times O(n^2)}$. The time consumed in a detection in Gossip is $O(logn)$ (Renesse et al. 1998). Therefore, we have $T_{\text{detection}} = \frac{O(\log n)}{f} = \frac{O(\log n) \times m \times O(n^2)}{B}$. The BDP of Gossip is BDP $= B \times T_{\text{detection}} = O(n^2 \log n)$.

(3) Our routing maintenance
Our routing scheme is a layered multicast. We assume that each subgroup consists of at most k storage nodes so that the number of the subgroup is $g \approx \frac{n}{k}$. The detection frequency and time are $f = \frac{B}{n \times m + g \times m \times k}$ and $T_{\text{detection}} = \frac{p}{f} = \frac{p \times (n \times m + g \times m \times k)}{B}$, respectively. We can have BDP $= B \times T_{\text{detection}} = p \times (n \times m + g \times m \times k) = O(n)$.

BDP is the cost of a routing setting or updating. From the above analysis, we can find that our routing protocol can detect the node failure within less cost than other two schemes in terms of BDP. Also, the heartbeat based all-to-all multicast and the heartbeat based gossip require BDP $O(n^2)$ and $O(n^2 \log n)$ respectively, while the BDP of our routing protocol grows up linearly with the increase of the storage size such that BDP=$O(n)$.

Lookup analysis

In this section, we present the analytical results of our hierarchical gossip-based membership protocol. Firstly, we define the following parameters characterizing the storage system.

n: the number of virtual nodes in the storage system
f: the fraction of operations that succeed in one hop to all operations.
r: the rate of membership changes (i.e., joining or exiting) in the system
T: the time needed to notify joining or exiting events to all storage nodes.

Our analysis does not take into account message loss and delay since we assume hosts are well connected by the network and the consistent hashing ensures the system has a balanced load distribution so that the network will not be congested by some hosts in the system.

Our goal is to achieve a desired success rate f for a system with an extremely large n, e.g., a system with 10^5 nodes,

when the expected membership change rate r is moderate. To do this, the system needs to bring forth the membership change events to every node within some time period T. In the worst case, all queries happen immediately after membership change events so that none of the affected soft state entries is corrected and all queries targeted at the down nodes must fail. We assume all the queries would fail at the first attempt before every node in the system get notified about the event. T consists of three parts such that $T = t_1 + t_2 + t_3$. Here t_1 is the delay between the time an event happens and the time the first group leader learns about it, and t_2 is the time from the end of t_1 it takes for all other group leaders get notified about the event. It will also take another t_3 to propagate the event notification to all the subgroups in system. Given the worst case assumption, all queries concerning the incorrect soft state entries must fail at the first attempt and therefore the fraction of failed queries $1 - f = \frac{T.r}{n}$. To ensure that at least a fraction f of queries succeed, we need

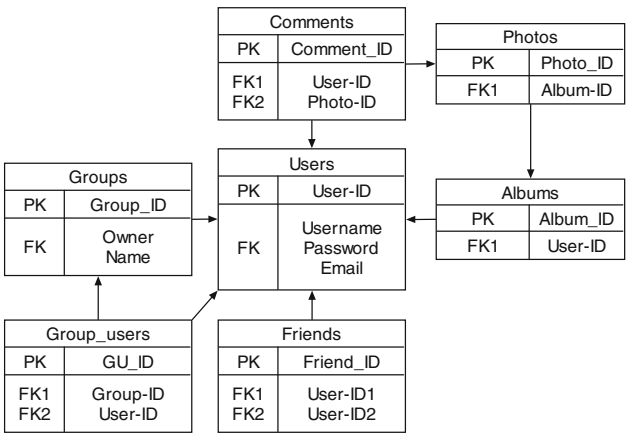
$$T \leq \frac{n.(1-f)}{r}$$

According to above formula, given a system with 10^5 nodes, 100 membership change events every minute, if 99.9% of operations can reach the correct destinations in one hop, the system will propagate the event notifications to all nodes within 6 s.

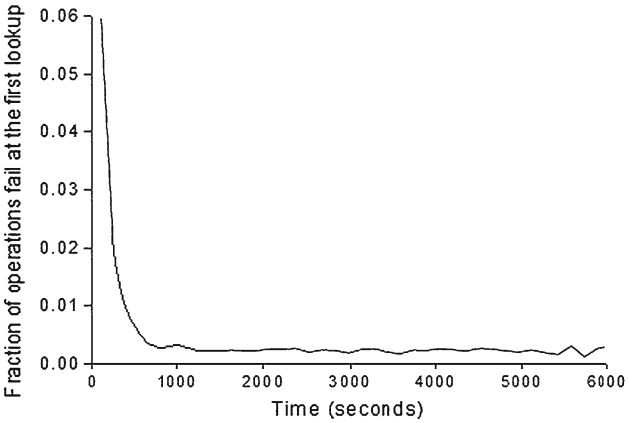
Performance evaluation

Based on the p2psim simulator, we developed a photo-sharing-oriented simulation system with 10^4 storage nodes. The database in our simulation system has the data structure in Fig. 5. We divided the nodes into 100 groups. Furthermore, each group was subdivided into 20 subgroups.

We tested the successful ratio of data lookup where 20 storage nodes joined and exited the simulation system every minute. The experimental result is illustrated in Fig. 6, where



5 Database schema in our photo sharing service



6 One hop lookup failure rate for a storage service with 10^4 nodes

X-axis and Y-axis refer to the testing time and the lookup failure ratio, respectively. After the first 1,000 s, the lookup failure rate stays around 0.03%, that is, 99.97% of the operations succeed in one hop lookup.

We have presented a virtual node based ring-like storage model and data placement scheme, proposed a data replication approach based on a quorum-fault-tolerant state machine protocol and a three-layered messaging protocol based on the heartbeat mechanism to detect the node failure and maintain the ring topology.

Our distributed storage scheme adopts consistent hashing and full membership mechanism that enables one-hop lookup and minimizes the latency. To enable the system to scale up to an extremely large one at the cost of acceptable overhead, we introduce a two-level hierarchical message notification structure to the gossip based membership protocol. Our storage scheme also replicates data items on several nodes (the number of replicas of one data item is configurable) to achieve high availability. We can scale up the system by adding new storage nodes any time without halting the system and the software will ensure that the newly added storage host receives the load from a number of existing nodes in the system. The theoretic analysis and experiment result demonstrate the feasibility and high operation efficiency of our storage scheme.

Feilong Tang would like to thank The Japan Society for the Promotion of Science (JSPS) and The University of Aizu (UoA), Japan for providing the excellent research environment during his JSPS Postdoctoral Fellow Program (ID No. P 09059) in UoA, Japan. This work was supported by the National Natural Science Foundation of China (NSFC) (Grant Nos. 60773089 and 61073148), the National High Technology Research and Development 863 Program of China (Grant No. 2006AA01Z172) and Hong Kong RGC (Grant No. HKU 717909E).

Celik, N., Nageshwaraniyer, S. S., & Son, Y. J. (2010). Impact of information sharing in hierarchical decision-making framework in manufacturing supply chains. *Journal of Intelligent Manufacturing*, online.

Chockler, G., Guerraoui, R., Keidar, I., et al. (2009). Reliable distributed storage. *Computer*, *42*(4), 60–67.

Decandia, G., Hastorun, D., Jampani, M., et al. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles (SOSP '07)* (pp. 205–220). New York, USA: ACM Press, 2007.

Gholami, M., & Zandieh, M. (2009). Integrating simulation and genetic algorithm to schedule a dynamic flexible job shop. *Journal of Intelligent Manufacturing, 20*, 481–498.

Greer, K., Stewart, J. R., & McCollum, B. (2008). Comparison of a centralised and distributed approach for a generic scheduling system. *Journal of Intelligent Manufacturing, 19*, 119–129.

Gupta, I., Birman, K., Linga, P., et al. (2003). Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *Proceedings of 2nd international workshop on peer-to-peer systems (IPTPS '03)*, 2003.

Jepson, T. C. (2004). The basics of reliable distributed storage networks. *IT Professional, 6*(3), 18–24.

Karger, D., Lehman, E., Leighton, T., et al. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of ACM symposium on theory of computing*, May 1997, pp. 654–663.

Marcelm-Jimenez, R., Rajsbaum, S., & Stevens, B. (2006). Cyclic storage for fault-tolerant distributed executions. *IEEE Transactions on Parallel and Distributed Systems, 17*(9), 1028–1036.

Park, J. H., Lee, S., Lim, J., & Yang, L. T. (2009). U-HMS: Hybrid system for secure intelligent multimedia data services in UbiHome. *Journal of Intelligent Manufacturing, 20*, 337–346.

Ramakrishnan, K. K., & Emer, J. S. (1989). Performance analysis of mass storage service alternatives for distributed systems. *IEEE Transactions on Software Engineering, 15*(2), 120–133.

Ramasubramanian, V., & Sirer, E. G. (2004). Beehive: O(1)lookup performance for power-law query distributions in peer-topeer overlays. In *Proceedings of the 1st conference on symposium on networked systems design and implementation*, San Francisco, CA, March 29–31, 2004.

Renesse, R. V., Minsky, Y., & Hayden, M. (1998). A gossip-style failure detection service. *Technical Report*, TR98–1687.

Rowstron, A., & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Lecture notes in computer science*, Vol. 2218, pp. 329–350, 2001.

Satyanarayanan, M., Kistler, J. J., & Kumar, P. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers, 39*(4), 447–459.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Survey, 22*(4), 299–319.

Stoica, I., Morris, R., Karger, D., et al. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)* (pp. 149–160). New York, NY, USA: ACM Press, October 2001.

Tian, J., Yang, Z., & Dai, Y. F. (2007). A data placement scheme with time-related model for P2P storages. In *Proceedings of seventh IEEE international conference on peer-to-peer computing (P2P 2007)*, 2007, pp. 151–158.

Wan, Y. P., Luo, Y., & Liu, L. (2009). A dynamic failure detector for P2P storage system. In *Proceedings of 2009 international conference on new trends in information and service science* (pp. 15–19). Beijing, China, June 30-July 02, 2009.

Yamamoto, H., Maruta, D., & Oie, Y. J. (2006). Replication methods for load balancing on distributed storages in P2P networks. *IEICE Transactions on Information and Systems, E89-D(1)*, 171–180.

Yang, C. T., Chen, H. Y., Huang, C. L., et al. (2009). A distributed file storage with Replica management in peer-to-peer environments. In *Proceedings of ninth IEEE international conference on computer and information technology (CIT '09)*, 2009, pp. 75–80.

Yi, S. Y., & Shin, H. (2010). A hybrid scheduling scheme for data broadcast over a single channel in mobile environments. *Journal of Intelligent Manufacturing*, online.

Yiu, W. P. K., Xing, J., & Chan, S. H. G. (2007). VMesh: Distributed segment storage for peer-to-peer interactive video streaming. *IEEE Journal on Selected Areas in Communications, 25*(9), 1717–1731.

Zeng, Z., & Veeravalli, B. (2008). On the design of distributed object placement and load balancing strategies in large-scale networked multimedia storage systems. *IEEE Transactions on Knowledge and Data Engineering, 20*(3), 369–382.

Zhao, B. Y., Huang, L., & Stribling, J., et al. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications, 22*(1), 41–53.