Churn Tolerance Improvement Techniques in an Algorithm-neutral DHT

Kazuyuki Shudo

Tokyo Institute of Technology,
2-12-1-W8-43 Ookayama, Meguro, Tokyo, 152-8552 Japan
shudo@computer.org

Abstract

Churn resilience is an important topic in DHT research. In this paper, I present techniques to improve churn resilience and their effects. All the techniques can be implemented in a DHT layer and require no change to underlying routing layers. In other words, they do not depend on a specific routing algorithm and can work with various algorithms.

Keywords: overlay network, DHT, structured overlay, churn, emulation

1 Introduction

Distributed hash tables (DHTs) are distributed systems, by which autonomous nodes provide a single hash table without center servers. There are extensive applications of a DHT because a hash table is a general purpose mechanism and can be applied to various kinds of name resolution.

Decentralized peer-to-peer systems such as a DHT provide the following merits to service providers utilizing it.

- Lower management and providing cost
- Higher scalability
- Higher reliability

If we choose an approach to improve reliability of each node in the same way as using a small number of servers, management cost rises much in proportion to the number of servers and it spoils the merits. Oppositely, it reduces management cost to use clients' computers to provide a service, but we cannot count on them to be reliable and highly-available.

To be reliable and cost-effective enough, a decentralized distributed system has to tolerate churn, the continuous process of node joining and leaving, and keep providing its service with churn.

In this paper, I present churn tolerance techniques for DHTs. Those techniques are independent of underlying routing algorithms. All techniques target the DHT layer and they require no change to underlying routing layers. The independence enables the techniques to work with various routing algorithms,

Proc. 3rd Int'l Conference on Autonomous Infrastructure, Management and Security (AIMS 2009), LNCS 5637, pp.42-55, June 2009

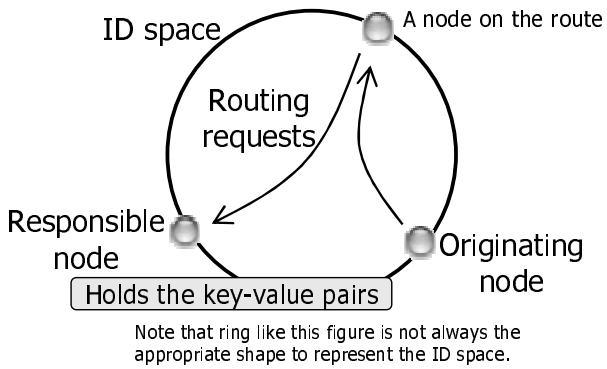


Fig. 1. Routing to put and get to a DHT.

Chord, Kademlia and others, which have their own advantages. The contribution of this paper is a demonstration that those techniques work with various algorithms effectively.

The churn tolerance techniques have been implemented in Overlay Weaver [1, 2], an implementation of structured overlay, which provides multiple routing algorithms. Section 4 presents the effects of the techniques.

2 Churn problem in a DHT

Put and get processes in a DHT are described as follows supposing an abstraction in which a DHT is an application of structured overlay (Figure 1) [3].
put(key,value) A node finds a responsible node by routing with the specified key, and transfers the key-value pair to the responsible node.
get(key) A node finds a responsible node by routing with the specified key, requests values associated with the key from the responsible node, and receives them.

It is possible for the get process to fail on condition that nodes in a DHT have joined and left. It is the churn problem in a DHT. Direct causes of the failure are as follows.

1. The key-value pair disappeared after put (because the responsible node departed).
2. The responsible node does not hold the key-value pairs.
   (a) A newly joined node became the responsible node for the key.
   (b) The routing for the put operation did not reach the suitable responsible node because of incomplete routing tables and other reasons.
3. Nodes in a route left during the routing. (This happens only in recursive routing [4].)

For instance, we cannot retrieve a key-value pair if a responsible node for the key departed after put. But the key-value pair remains on the DHT if it has

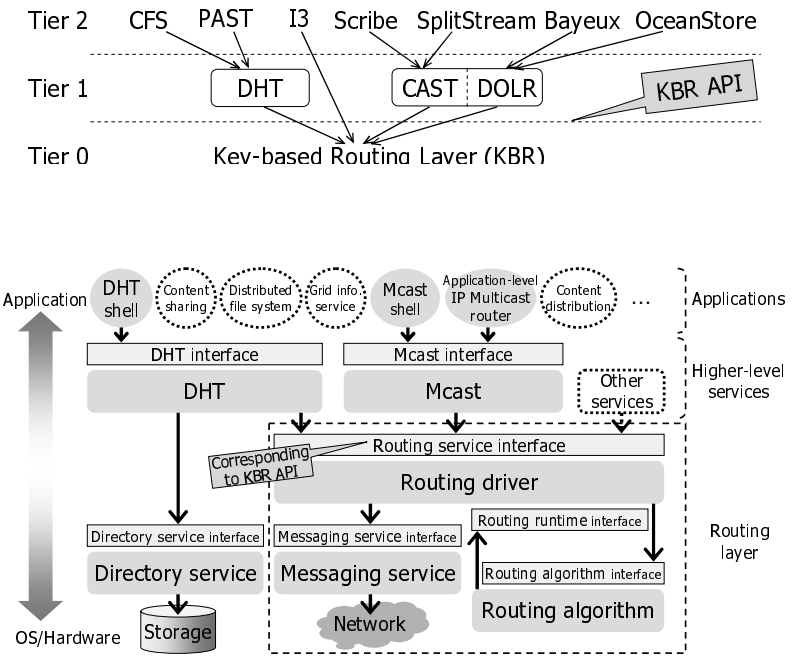


Fig. 3. Components organizing runtime of Overlay Weaver.

been replicated to other nodes in advance of the departure. Section 3 presents these sorts of techniques to improve churn tolerance.

3 Churn tolerance techniques

In this section, I describe techniques to improve churn tolerance of a DHT. I have implemented all the techniques in a DHT implementation of an overlay construction toolkit Overlay Weaver [1, 2], and measured their effects.

Overlay Weaver follows an abstraction of structured overlay proposed by Dabek et al. [3], in which higher level services such as DHT and multicast are built on the basic routing layer (Figure 2). This model enables us to combine arbitrary implementations of each layer and Overlay Weaver demonstrated it. For example, we can choose and combine a suitable implementation of a routing algorithm for an application with a DHT implementation.

If the churn tolerance techniques are implemented only in the DHT layer, they are independent of underlying routing algorithms and work with various algorithms. This flexibility is a nature of churn tolerance techniques. The techniques are apt to refer to internal structures of a routing algorithm (e.g. leaf set in Pastry) and find contacts with other nodes. The techniques can be independent of a routing algorithm by accessing the routing layer only through a carefully-designed and algorithm-neutral interface.

3.1 Routing layer API

In the abstraction proposed by Dabek et al., the routing layer provides a function supporting replication to upper layers including DHT. The function is `replicaSet(key,max_rank)`. It returns an ordered list of nodes which are candidates for the responsible node for the specified key. The order reflects how are the nodes adequate to be the responsible node and have a replica. In other words, after the first node departs, the next node is most adequate for the responsible node.

Overlay Weaver provides a similar mechanism to support churn tolerance techniques. A routing process results in the list of candidates for the responsible node, not just one responsible node. This mechanism is different from Dabek's one which is a function called locally. It serves a number of techniques not only replication. Additionally, Dabek's proposal does not include an empirical proof (Section 6), but this paper demonstrates efficiency of algorithm-neutral churn tolerance techniques.

Within the routing layer (Figure 3), the Routing algorithm component provides a function `rootCandidates(ID target,int maxNumber` to a Routing driver component. The function is similar to Dabek's `replicaSet` function.

Following sections present these four techniques.

- Replication
- Join-time transfer
- Multiple get
- Repeated implicit put

First three techniques utilize the list of responsible node candidates.

3.2 Replication

In a put process, not only the responsible node but also several nodes receive and hold the key-value pair which has been put. Even if the responsible node left after the put, later get requests can complete by obtaining the requested values from the other nodes.

The key-value pair is replicated to several nodes along the list of responsible node candidates, which is obtained as a result of routing. By this replication, even if the responsible node left, later routing reaches the next candidate, which returns the requested value, and get requests complete.

This is not the only method to select nodes on which a key-value pair is replicated. An example of other methods is generation of multiple keys. For example, a put process can generate another three keys based on the specified key by exclusive-or'ing 01, 10 and 11 to the first 2 bits of the key: $key \oplus 010..$(binary), $key \oplus 100..$, $key \oplus 110...$ But I have not adopted this method because it requires a larger amount of messages between nodes. Replication on responsible node candidates requires routing just once, but this method involve multiple routing the same times as the number of replicas.

Our implementation of replication allows both the node originated the put request and the responsible node to initiate replication. The number of messages sent to make replicas is identical in both cases, but efficiency of those messaging differ if node IDs reflect network proximity. The responsible node is better if the distance between node IDs in the routing algorithm reflects network proximity with proximity identifier selection (PIS) [5]. The efficiency is identical in both cases because our implementation does not take account of network proximity currently.

The implementation of replication has the two following parameters.

- The number of replicas
- Performing node: the node originated the put request or the responsible node

Only the responsible node hold the key-value pair if one is specified as the number of replicas.

Note that we should distinguish replication from caching. A cache is a key-value pair preserved by a node in an earlier put or get process in which the node is involved. It is for the purpose of performance improvement, not for churn tolerance though it contributes to churn tolerance unexpectedly. However, it is possible to cache a key-value pair with the intention of utilizing it also as a replica.

Replica consistency The DHT implementation of Overlay Weaver, on which our research is based, preserves eventual consistency [6, 7] at most even with the churn tolerance techniques in this paper.

One of natural concerns about consistency of replicas is different values associated with the same key according to nodes. It happens in cases that an old value to be overridden remains and almost simultaneous put requests compete.

But these are not the cases in our DHT implementation because multiple values can be associated with a single key and no value is removed by a put request. This is the same policy as a DHT implementation Bamboo [8] and its deployment, OpenDHT [6]. A get request yield all values associated with the specified key.

However, it is possible for a get request to result in an obsolete value which has been removed if a remove operation failed on some replicas. Resolution for such inconsistency is left for an application which uses the DHT. Vector clock in Dynamo [7] is a promising support to resolve such inconsistency even though the current implementation does not provide it.

3.3 Join-time transfer

It is possible for a newly joining node to be the most proper responsible node for existing key-value pairs. In this case, a node holding those pairs transfers them to the newly joining node. This technique is called join-time transfer in this paper. With this technique, the new responsible node can respond to later get requests.

Table 1. Causes of get failure each technique treats.

	Replication	Join-time transfer	Multiple get	Repeated implicit put
• Key-value pairs disappeared	√			√ (requires replication)
• A responsible node does not hold the key-value pairs:				
· A joined node became a responsible node		√	√	√
· A joined node became a responsible node		√	√	√
• Nodes in a route left				

In our implementation, a joining node asks a few nodes which are expected to hold key-value pairs the joining node should have. The asked nodes are responsible node candidates (Section 3.1) for the node ID of the joining node. Routing to join yields the list of the candidates and the joining node asks the specified number of the candidates in order as in the list. An asked node transfers key-value pairs which the joining node is more proper to have and the joining node holds them. Note that the transferring node does not expressly discard those pairs though the node can do it and keeps them.

The only parameter of this technique is the number of nodes a joining node asks.

3.4 Multiple get

In a get process, the requesting node can ask values from multiple nodes, not only the responsible node for the key. The requested nodes are responsible node candidates the same as replication. With this technique, if a get request is routed to a newly joined node which does not have the value, an old responsible node is possible to be also asked and return the value.

A former mentioned technique, join-time transfer, also mitigates the problem where a newly joined node became a responsible node. Effects of the two techniques overlaps but are not identical.

Multiple get compensates suboptimal routing for putting. A routing can reach a node other than the responsible node for the specified key when routing tables are incomplete. In this case a suboptimal node holds the key-value pair, but later get requests can ask the holding node with the multiple get technique.

The number of asked nodes is the parameter of the multiple get technique. Only one responsible node receives a get request in case the parameter is 1.

3.5 Repeated implicit put

Each node composing a DHT puts key-value pairs it holds periodically and autonomically without explicit put requests. The implicit put process also makes replicas as an usually-requested put process.

The number of replicas gets fewer according to nodes leaving even though join-time transfer supplements replicas. The purpose of this technique, repeated implicit put, is supplementation of replicas.

This technique has the same effect as the join-time transfer. A node which joined after a key-value pair was put can receive it if the node is responsible for the key. But its effect is limited as it cannot save a get request issued before an implicit put process runs. There is an interval between implicit put processes.

This technique looks useless in case the number of replicas is 1. It is not correct. Even in the case, the technique has the same effect as join-time transfer and can transfers key-value pairs to the proper responsible node.

The parameter of this technique is an interval of implicit put processes. A node waits for the specified time between the processes. Actual intervals are fluctuated a little with random numbers and the fluctuation prevents synchronized behavior of many nodes, which put a much load on the overlay.

Only this technique does not refer to a list of responsible node candidates (Section 3.1) though other three techniques presented in this paper use the list.

3.6 Targets of each technique

Table 1 illustrates causes of get failure, listed in Section 2, which each technique treats.

Replication prevents disappearance of key-value pairs which happens according to node leaving. Repeated implicit put supplements replicas, however the technique does not take the effect if the number of replicas is 1.

Both join-time transfer and multiple get treat a problematic situation where a node does not hold key-value pairs which the node is responsible for. The former one prevents the situation and the latter one enables a get request to succeed in the situation.

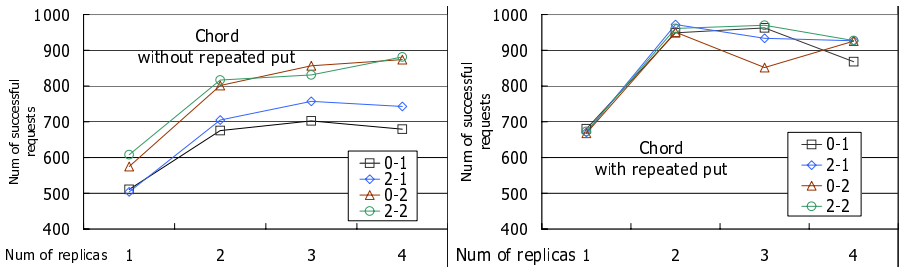
Repeated implicit put is also possible to save the situation (Section 3.5), but it takes effect intermittently.

This paper presents no solution to the last cause in which nodes on the route leaves during a recursive routing. Concurrent multiple routing processes can mitigate this problem.

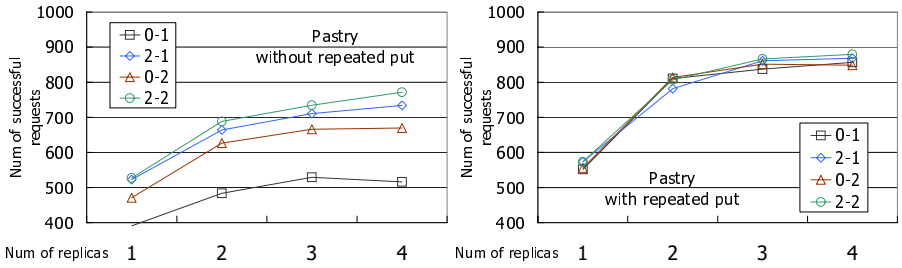
4 Effect of techniques

This section demonstrates the effects of the techniques presented in Section 3. In an experiment, we ran 1000 nodes on a single computer, issued a number of get requests, measured the number of successful requests, and calculated the success rate.

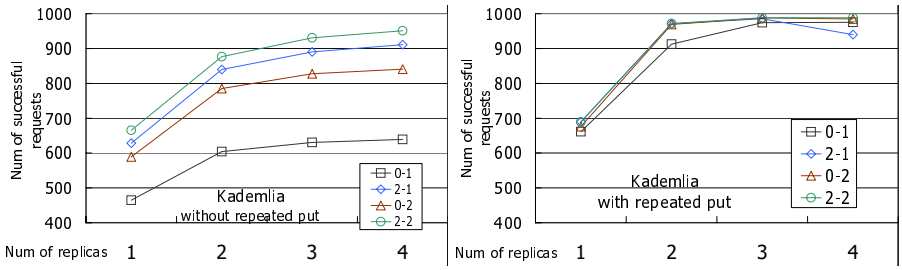
Those nodes ran on a Distributed Environment Emulator, which Overlay Weaver provides. It hosts a large number of nodes and controls them along a given emulation scenario. An Emulator hosts an application which can work on a real network without any modification. The Emulator provides a lightweight



(a) Chord



(b) Pastry



(c) Kademlia

Fig. 4. Effects of presented techniques.

Messaging service which bypasses a TCP/IP protocol stack and lightens the communication process between hosted nodes though those nodes can use TCP/IP if we want.

The Messaging service delivers a message as fast as the hosting computer can. It does not copy a message body even in memory and then the bandwidth between nodes is infinity in theory. It models an ideal communication medium effectively and not limited by a physical medium. Churn is the only cause of failure of a get request because no message is lost. In other words, the communication medium, the in-memory Messaging service takes no effect on the success rate.

4.1 Conditions

We generated a churn scenario with a generating tool and gave the scenario to hosted 1000 nodes. All the following experiments were made with the identical scenario.

The scenario is as follows.

1. Starts 1000 nodes.
2. Lets all nodes join a DHT, one every 0.15 seconds.
3. Puts different 1000 key-value pairs on the DHT, one every 0.2 seconds.
4. Gets all 1000 key-value pairs from the DHT, one every 0.2 seconds.

Nodes to which put and get were determined at random when the scenario is generated.

Churn lasts from the start of put to the end of get. The scenario keeps the number of nodes as 1000 by letting a node leave and another node join immediately. This model of churn is identical to experiments by Rhea et al. [9], and different from the churn model adopted by Kato et al. [10] In the latter model, the number of nodes varies because another node does not join immediately after a node left.

Churn is timed by a Poisson process and the frequency of it is 2 times a second. Thus the average life time of a node is $1000(nodes)/2(times/sec) = 500seconds$.

Communication timeout is 3 seconds and routing timeout is 10 seconds. Because of it, a put or get request timeouts and fails if tries to contact a failed node 4 times. A node removes a failed node from its routing table.

We used a PC with 2.8 GHz Pentium D processor, Linux 2.6.21 for x86-64 and HotSpot Server VM of Java 2 SE 5.0 Update 12. The version of Overlay Weaver is 0.6.4. All experiments were made 6 times and we adopted the average of middle 4 values as the result.

4.2 Results

Figure 4 shows the results. We conducted experiments with all combinations of all routing algorithms Overlay Weaver provides and two routing styles, iterative and recursive routing. While Overlay Weaver supports various routing algorithms including Chord, Kademlia, Koorde, Pastry, Tapestry and their variations, Figure 4 shows results with representative ones, Chord, Pastry and Kademlia, and the routing style is iterative routing.

The vertical axis indicates the number of successful get requests out of 1000 requests. Numbers closer to 1000 are better. The horizontal axis indicates the number of replicas from 1 to 4. Each algorithm has two graphs. Graphs on the left side shows the results with repeated implicit put enabled, and disabled in the right-hand graphs. The interval of the implicit put processes is 30 seconds.

Four lines in a graph differ in parameters of two techniques, join-time transfer and multiple get. The parameter of the former technique is the number of nodes to which a newly joined node asks. It is 0 (disabled) or 2. The parameter of the

latter technique is the number of target nodes of get requests. It is 1 (disabled) or 2. Legends such as "<number>-<number>" in the graphs mean these two parameters. A couple of numbers means "<join-time transfer>-<multiple get>".

4.3 Observations

We observed the following facts in Figure 4.

- A larger number of replicas resulted in a better success rate.
- Both join-time transfer and multiple get improved success rates.
- In Pastry and Kademlia, join-time transfer (with the parameter 2) was more effective than multiple get (with the parameter 2). Oppositely, in Chord, multiple get was more effective.
- Repeated implicit put filled up the gaps between the results with different parameters of join-time transfer and multiple get. Four lines in the right graphs are very close.
- Replication with 3 replicas showed better results than 4 replicas in a few cases.

These trends were also observed in the results with recursive routing. Tapestry, another routing algorithm, shows the same trend as Pastry as expected. They share an important part of their routing algorithm and this result looks natural.

Part of get requests failed even though the churn tolerance techniques were applied. The reasons of those failure are as follows.

- The techniques and the parameters were not enough to compensate such degree of churn completely.
  For example, all replicas disappeared or a get request was issued before the responsible node received a replica by implicit put.
- Routing requests did not reach the responsible node because of incomplete routing tables.
- Timeouts happened many times in communication and a routing could not finish (Section 4.1).

Note that the results shown in this section do not support relative superiority of each routing algorithm. Each algorithm has its specific parameters and they have an effect on churn tolerance. In this paper, all parameters of routing algorithms were the default values of Overlay Weaver 0.6.4 and were not adjusted.

This section could demonstrate that algorithm-neutral churn tolerance techniques take effect. The next problem is which technique and what parameter we choose. The decision should be based on cost performance, not only performance. In the next section, I discuss cost performance of churn tolerance techniques.

5 Cost performance of churn tolerance techniques

In this section, I discuss how to calculate cost performance of churn tolerance techniques. It is rational that the effect is represented by a function which takes

the success rate of get requests as an input, because the effect appears as the success rate.

Next, let me consider the cost. There are various kinds of cost of the churn tolerance techniques as follows.

- Traffic and the number of times of communication
- Time required to join, put and get
- Memory and storage consumption
- Processing time

Preceding work [9, 10] focused on communication traffic and time to complete get operations. Other kinds of cost have drawn less attention and one of the reasons for it is that the other resources are seldom a bottleneck on today's Internet.

Nevertheless, replication with n replicas consumes n times larger storage or memory. It is fairly expensive with a large amount of data or on an embedded node with small resources. Therefore it is necessary to premise an application and an applied environment when considering the cost.

Table 2. DHT processes in which each technique is involved.

	join	put	get	ordinary
Replication		√		
Join-time transfer	√			
Multiple get			√	
Repeated implicit put				√

It is also necessary to take account of the behavior of the system which depends on an application. Table 2 shows timing on which each churn tolerance technique works. This table indicates that replication is expensive with many put operations and multiple get is expensive with many get operations. Repeated implicit put is a process which keeps running and takes its cost continuously even without put or get requests. The cost in a time unit increases according to the number of key-value pairs in the DHT.

Suppose that the application is the domain name system (DNS), the frequency of get operations is much higher than other operations such as join and put. In this case, multiple get is expensive but replication and join-time transfer are relatively inexpensive. It may be possible to use an expensive parameter for such lower frequent operations. For example, an application like the DNS may allow a higher number of replicas. On the other hand, a different type of application shows different properties. A sensor network on a DHT will receive more put requests than the DNS and we cannot neglect the communication and storage cost of replication.

As shown here, we cannot calculate the cost of churn tolerance techniques without premising a concrete application. By making such a premise, we can estimate the frequency of each operations on a DHT, join, put, get and remove. If we

have real-world traces of an application or a scenario which reflects application behavior, we can calculate the determinate cost based on them.

The emulation scenario for experiments shown in Section 4 is artificial. It does not make significant sense to inspect the cost in the experiments.

6 Related work

A layered model of structured overlay proposed by Dabek et al. (Figure 2) [3] includes an API to implement replication (Section 3.1). The authors proposed the model and APIs between the layers. The paper includes no empirical proof of those proposals.

In contrast to it, I demonstrated that it is possible to implement a number of churn tolerance techniques such as join-time transfer, multiple get and repeated implicit put in addition to replication. Those techniques were implemented based on a single mechanism similar to Dabek's `replicaSet` function. The mechanism provides the DHT layer with the list of responsible node candidates. Section 4 showed the effects of those techniques combined with various routing algorithms.

Rhea et al. evaluated churn tolerance of Bamboo [8], that is their DHT implementation [9]. The authors used a network emulator ModelNet and ran 1000 nodes on 40 PCs. On the emulated environment, they measured the effects of methods to recover node failures, reactive recovery and periodic recovery. They also measured and evaluated techniques to reduce the required time to perform routing. Those techniques include TCP-style timeout calculation and proximity neighbor selection (PNS) [5].

All techniques in Rhea et al. are implemented in the routing layer of the Dabek's model [3], while the techniques described in this paper target the DHT layer. Rhea et al. premises their Bamboo implementation and its particular algorithm derived from Pastry. In contrast to it, the techniques in this paper naturally work in combination with various routing algorithms because their target is the DHT layer, which is independent of routing algorithms. The most significant contribution of this paper is empirical demonstrations of the combinations of the techniques and the routing algorithms shown in Section 4.

The techniques in this paper work together with all techniques shown in Rhea et al. because their target layers are different and they do not conflict. The communication layer of Overlay Weaver actually implements TCP-style timeout calculation proposed in Rhea et al.

Kato et al. evaluated 4 DHT algorithms, Bamboo, Chord, Accordion and FreePastry on their network emulator peeremu [10]. They conducted experiments in which up to 1000 nodes ran on 10 or 20 PCs. Their metrics were success rate of get requests and time to complete them.

There is a different type of approach to build a churn tolerant system, supernodes. The distributed system itself elects supernodes suitable for composing an overlay based on their attributes such as computation performance, bandwidth and running time so far. Only the elected supernodes maintain a DHT (an overlay) and they serve other ordinary nodes.

This supernodes architecture relaxes churn tolerance required for a DHT. It introduces another merit, by which the system qualifies convenient nodes capable of composing a DHT, for example, capable of bi-directional communication. Conventional routing algorithms of structured overlay require all nodes on an overlay to be able to communicate each other. It makes DHT construction easier to choose supernodes as they can communicate in bi-direction, for example, not behind a NAT. With supernodes, churn tolerance is still an important property because even supernodes suffer churn.

7 Conclusion

I presented a number of churn tolerance techniques for DHT and demonstrated their effects. They work with various routing algorithms of structured overlay because they all target the DHT layer of the Dabek's model. I implemented those techniques in Overlay Weaver, measured their effects with all the algorithms Overlay Weaver provides, and showed the effects with Chord, Pastry and Kademlia in Section 4. The most significant contribution of this paper is the empirical demonstration of those routing-algorithm-neutral churn tolerance techniques.

In Section 5, I discussed how we can determine which technique and what parameter we choose. The decision should be based on cost performance of the techniques and I concluded that we need an emulation scenario which reflects a concrete application because the cost is heavily dependent on the frequency of join, get, put and remove operations on a DHT.

A promising next step is measurement and calculation of cost performance based on concrete applications such as DNS and sensor networks. It leads to the establishment of methodology by which we construct a churn-tolerant system with appropriate techniques and their parameters.

Acknowledgments

I would like to thank Daishi Kato, Youki Kadobayashi, Yusuke Doi, Akito Fujii, Mikio Yoshida and members of IDEON working group in WIDE project for insightful discussions.

References

1. Shudo, K., Tanaka, Y., Sekiguchi, S.: Overlay Weaver: An overlay construction toolkit. Computer Communicatios (Special Issue on Foundations of Peer-to-Peer Computing) **31**(2) (2008) 402–412
2. Shudo, K.: Overlay Weaver: An overlay construction toolkit (2006) http://overlayweaver.sf.net/.
3. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: Proc. IPTPS'03. (2003)

4. Rhea, S., Chun, B.G., Kubiatowicz, J., Shenker, S.: Fixing the embarrassing slowness of OpenDHT on PlanetLab. In: Proc. WORLDS '05. (2005)
5. Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., Stoica, I.: The impact of DHT routing geometry on resilience and proximity. In: Proc. SIGCOMM 2003. (2003)
6. Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: OpenDHT: A public DHT service and its uses. In: Proc. ACM SIGCOMM 2005. (2005)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proc. SOSP 2007. (2007)
8. Rhea, S.C.: The Bamboo distributed hash table (2003) http://www.bamboo-dht.org/.
9. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proc. USENIX '04. (2004)
10. Kato, D., Kamiya, T.: Evaluating DHT implementations in complex environments by network emulator. In: Proc. IPTPS 2007. (2007)