

Evaluating DHT Implementations in Complex Environments by Network Emulator

Daishi Kato and Toshiyuki Kamiya
NEC Corporation

Abstract

We propose an evaluation method of large-scale distributed hash tables by network emulator that allows us to evaluate not only DHT algorithms but also DHT implementations. Evaluating DHT implementations is important for DHT application developers since it influences how an application utilizes a DHT. We built a platform to support our method and evaluated four implementations with various scenarios. The evaluation results show that our method is useful for evaluating the performance of DHT implementations in practical and complex environments.

1 Introduction

In the research area of peer-to-peer networks, Distributed Hash Table (DHT) techniques have received much attention. A DHT is a virtual database maintained by nodes communicating with each other in a peer-to-peer network. The features of DHTs include: a) fairness: all nodes perform the same role and can join and leave the network at any time, and b) scalability: the cost of maintaining a network is relatively low even for a large network.

These DHT features allow such usability that one can build a large network without a high-performance server. Examples of DHT applications include DNS-like resolving systems for host lookups, instant messaging systems with user lookups, and file sharing systems with file lookups.

Past work on DHTs has proposed various algorithms and implementations. Popular DHTs that have implementations include Chord [11], Pastry [10], Kademlia [7], and Bamboo [8]. Application developers who want to use a DHT need to select a DHT based on their requirements. Since the basic functionality of these DHTs is identical, they compare DHT performances, for exam-

ple, network bandwidth usage and response delay. Evaluations of DHT implementations are necessary to estimate these performance metrics.

We propose an emulation method to evaluate DHT implementations in a local experimental environment. The emulation method allows DHT implementations to run without modifications and supports many virtual nodes in a few real machines. A network environment is also emulated, and scenarios are used to run evaluations. We developed a platform for our method and did experiments based on it.

The rest of this paper is organized as follows. In Section 2, we explain related work that can be used to evaluate DHTs. In Section 3, we describe our evaluation platform and scenarios for evaluating DHTs. In Section 4, we explain our experiments in which we evaluated existing DHT implementations. In Section 5, we summarize our contribution and describe future work.

2 Related Work

p2psim [5] is a simulator for various DHT algorithms that can compare several DHTs in an identical scenario that is automatically generated by specified parameters. A p2psim user has to write a DHT algorithm in C++ code that is not reusable in real implementations. Nevertheless, the generating scenarios approach is inherited by our platform.

MACEDON [9] is a platform to develop DHTs as well as other overlay networks that provides a special and simple grammar for programming and produces a simulation code and a real implementation code. Hence, a user of MACEDON only needs to write a single code. However, MACEDON is not capable of evaluating pre-existing DHT implementations that are not written in MACEDON grammar.

ModelNet [12] is a large-scale network emulator that can be used to evaluate DHT implementations. It em-

ulates a router network by a single machine or several machines for scalability. A user can run a DHT implementation without modification. However, ModelNet by itself does not include a controller for scenarios to evaluate DHTs.

3 Evaluation Platform

3.1 Motivation

The performance of DHTs is important for application developers who would have difficulty designing applications if they did not know how much data to put into databases.

We propose an evaluation method for DHTs by network emulator. Emulation allows us to run DHT implementations without modifications in a virtual network. To ease the evaluation of DHTs by emulation, we developed a platform to support it that provides four features: a) a definition of a common API of DHTs, b) emulation of nodes and networks, c) generation of scenarios, and d) measurement of performance metrics.

The rest of this section describes these four features in detail.

3.2 API of DHT

We define an API of DHT so that the API is common to all DHTs. Each DHT implementation has to be wrapped to provide the common API. Our aim is to support as many DHTs as possible and to make wrappers very thin. Dabek et al. has proposed a common API earlier [1], but its goal is to abstract several services including a DHT by introducing Key-Based Routing (KBR), and it is not sufficient for our platform in which the common API is to abstract APIs of several implementations.

Our definition of API includes the following four functions:

- *Join* starts a node. It is called with gateway information, which is a node first contacted by the starting node. It returns no value (a.k.a. void).
- *Leave* stops a node. It returns no value.
- *Put* inserts a pair of a *key* and a *value* into a database. It returns no value, which is not intuitive but is less restrictive. *Keys* and *values* are represented in strings.
- *Get* fetches pairs by a *key* and returns a list of pairs whose *keys* are the same as the specified *key*. The

list is returned asynchronously, which is less restrictive. Note that some implementations might return only one pair.

These functions are simple commands that are written in scenarios and used to invoke wrappers. Adding a new command only incurs defining a new syntax in scenarios and modifying wrappers to accept the new command.

For our evaluation, we built wrappers for Bamboo, Chord, FreePastry [2], and BitTorrent DHT [4].

3.3 Emulation of Nodes and Network

To evaluate a large-scale network in a relatively small environment, we adopted emulation. There are two types of emulation.

First, several nodes are emulated in fewer machines that are connected to each other in a Local Area Network (LAN). This is done by using “IP Aliasing” provided by Linux. Each node runs as a process or a set of processes and uses an assigned virtual IP address to distinguish it from other nodes that run on the same machine. By using IP Aliasing, DHT implementation does not need to be modified except for specifying virtual IP addresses. For implementations written in Java, however, we modified the code so that several nodes run in a single Java Virtual Machine (JVM) that works efficiently in terms of memory usage. We typically run 20-80 nodes per machine.

Second, the network topology of the Internet is emulated in LAN by using “Traffic Control,” again provided by Linux. Packet delays and losses between any two nodes (IP addresses) can be generated by Traffic Control. Since this generation of packet delays and losses is done at all machines, and no single machine grabs all packets, this approach is fairly scalable.

Although ModelNet can also be used for emulating nodes and network, it is not used for our first version of the platform because of the following reasons: a) ModelNet is not well designed to run several nodes in a single JVM and quite a few of target DHTs are implemented in Java. b) Our first evaluation can be done enough with emulating a relatively easy end-to-end topology as opposed to a router network topology that ModelNet can emulate.

3.4 Scenarios for evaluations

Scenarios, which are used to run evaluations, are useful to run one evaluation many times with different DHT

implementations or even with a single DHT implementation, since one can compare different DHT implementations with the same scenario or check if a result of an evaluation by a scenario is reproducible.

The primitive representation of a scenario is a sequence of events in a timeline. An event is either *join*, *leave*, *put*, or *get* of the API functions with node information that invokes the function.

Since writing a scenario word for word is hardly possible for large-scale evaluations, we provide a tool that takes scenario parameters and generates a scenario. There are two scenario models for *join* and *leave* events. One is a *static* model in which all nodes *join* networks at the start of an evaluation and never *leave* until the end of an evaluation. The other is *join/leave* model in which nodes *join* and *leave* randomly throughout an evaluation. This model is the same as the one used in p2psim, and two parameters are specified: *lifemean* is the mean of seconds between *join* and *leave* of a node; *deathmean* is the mean of seconds between *leave* and *join* of a node. Note that when a node re-*joins*, it newly *joins* and no information is kept from the previous *join*. Parameter *joininterval* for both models is also specified for the interval of seconds between two nodes that *join* at the start of an evaluation. For *put* and *get* events, we specify three parameters: *putinterval* is seconds between two *puts*, and *getinterval* is seconds between two *gets*; *putmax* is the number of *puts* that a node calls after the node it joins, and *gets* are summoned after *puts* until the node *leaves*. For simplicity, we keep one node active throughout an evaluation as a gateway used by other nodes when they *join*.

3.5 Performance Metrics

The following are the performance metrics obtained from an evaluation: 1) *get* results, 2) *get* response delays, and 3) network traffic.

All *get* results are logged, and our analyzing tool calculates the percentage of successful *gets* either throughout an evaluation, per node, per period, or per node and per period.

All *get* response delays are logged, and our analyzing tool averages them either throughout evaluation, per node, per period, or per node and per period.

Network traffic is logged at certain periods for each node. The logged information is the total number and total octets of outgoing packets from each IP address within a logging period. Logging all packets is techni-

avg. number of active nodes	316
experiment time	1 hour
<i>lifemean</i>	60 min
<i>deathmean</i>	20 min
<i>joininterval</i>	600 ms
<i>putinterval</i>	20 sec
<i>getinterval</i>	20 sec
<i>putmax</i>	10

Table 1: Basic Parameters

Bamboo	snapshot of 2006/03/03
Chord	CVS snapshot of 2006/01/27
FreePastry	version 1.4.4

Table 2: DHT Implementation Versions

cally possible, but not feasible for large-scale evaluations for performance reasons. The logging interval can be one second at least or bigger, but a smaller interval also creates a performance problem. We use 60-second intervals for moderate evaluations. Typically, we calculate the average bandwidth of the network traffic and investigate more detailed metrics on demand.

4 Experiments

We use our platform and evaluate several DHT implementations publicly available on web sites. The evaluations are performed with various scenarios in a virtual network as large as a thousand nodes to clarify differences among DHTs. Our platform eases such comparison experiments.

4.1 Basic Parameters and Target Implementations

We define basic parameters to generate a “standard” scenario. The basic parameters are chosen by preliminary experiments so that a machine load that emulates nodes is about half of the maximum possible load of the machine. Table 1 shows some of the basic parameters. Concerning packet delay, we use “kingdata” [3], as p2psim does, which is a measured delay matrix, and packet loss is uniformly set to be 0.1%.

Target DHT implementations are Bamboo, Chord, Accordion [6], and FreePastry. Chord and Accordion share the same implementation and only differ in configura-

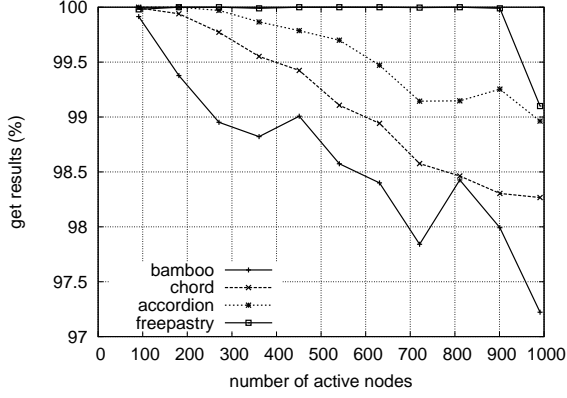


Figure 1: 1) Basic: get results

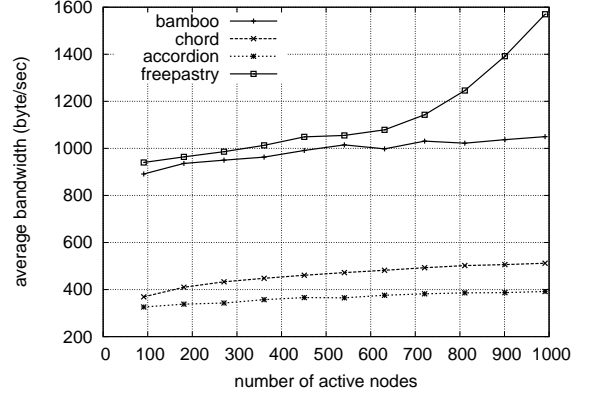


Figure 3: 1) Basic: network traffic

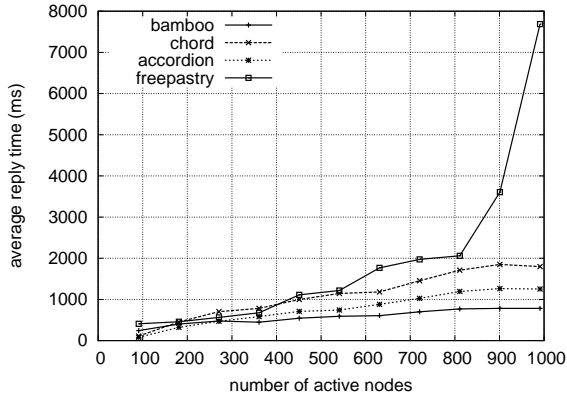


Figure 2: 1) Basic: get response delay

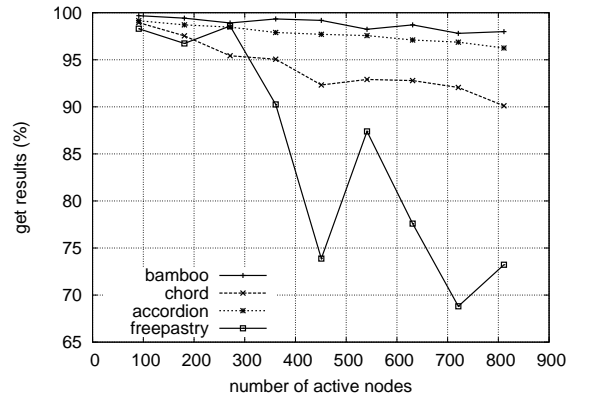


Figure 4: 2) Join/Leave: get results

tion parameters. The versions of the implementations are shown in Table 2.

These DHT implementations have configuration parameters. Some implementations can be configured in detail by the parameters, and the configuration may affect many evaluation results. Hence, we chose configuration parameters of each DHT implementation by preliminary experiments so that the evaluation result with the standard scenario keeps a fairly high rate of successful *gets* (more than 95% on average) and the least network traffic on average.

4.2 Evaluation Patterns

In this section, we describe the five patterns of evaluations.

1) Basic: As the most basic evaluation, we experimented with DHTs in *static* models. To confirm the scalability of DHTs, the parameter of “avg. number of active nodes” was varied from 91 to 991.

2) Join/Leave: Next, we experimented with DHTs in

join/leave models. In this experiment, since most nodes that *join* at the beginning will *leave* during the experiment, we can confirm that nodes correctly take over data. The parameter of “avg. number of active nodes” was varied from 91 to 991.

3) Multi-Value: In the standard scenario, nodes *put* a *value* with a unique *key*. This experiment lets nodes *put* multiple *values* for each *key*. This confirms that a DHT implementation supports multiple *values*.

4) Separate-Put-Get: In the standard scenario, parameters are identical for all nodes. In other words, nodes act homogeneously. This experiment introduces heterogeneity; one group of nodes only *puts* data, while the other group of nodes only *gets* data. A varied parameter is the ratio of the two groups.

5) Separate-Put&Get-Dummy: This experiment also introduces heterogeneity; one group of nodes *puts* and *gets* data, while the other group of nodes does not *put* or *get*. Even in this scenario, the nodes in the second group hold data that the nodes in the first group *put*. A varied parameter is the ratio of the two groups.

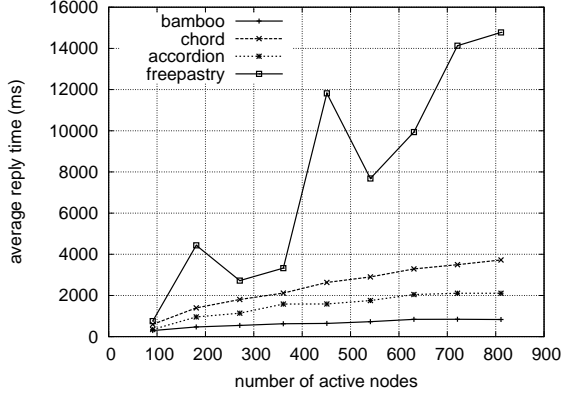


Figure 5: 2) Join/Leave: get response delay

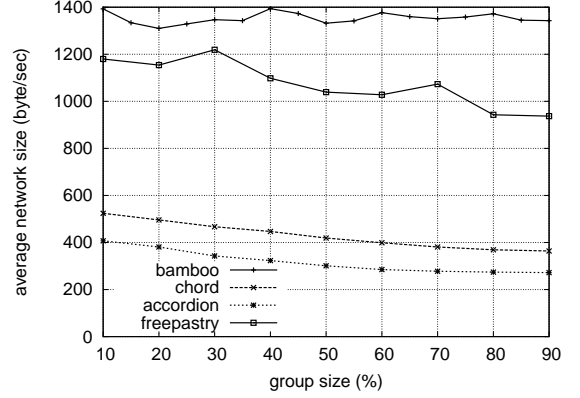


Figure 7: 4) Separate-Put-Get: network traffic

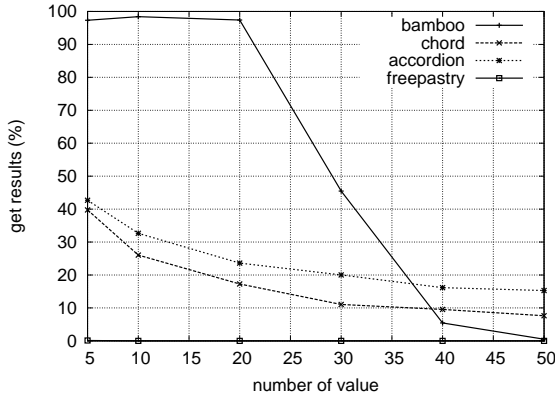


Figure 6: 3) Multi-Value: get results

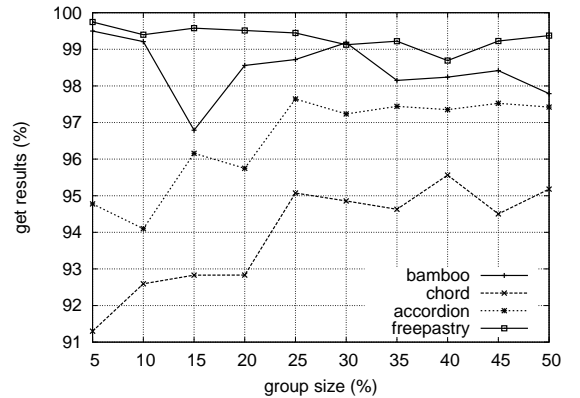


Figure 8: 5) Separate-Put&Get-Dummy: get results

We ran a single experiment for these five patterns of evaluations and plotted the results in figures.

4.3 Evaluation Results

In this section, we describe the results of the five evaluations.

1) Basic: The percentage of successful *gets*, shown in Figure 1, remains high (more than 95%) for all DHTs. FreePastry works relatively well in this scenario, staying at almost 100%. *Get* response delay (shown in Figure 2) increases similarly for all DHTs, while DHTs can be ordered for smaller delay: Bamboo, Accordion, Chord, and then FreePastry. The delay of FreePastry jumps up at 900 of the x-axis and over. We assume a kind of limitation in the FreePastry implementation. Network traffic (shown in Figure 3) stays relatively low for Chord and Accordion compared to Bamboo and FreePastry.

2) Join/Leave: The percentage of successful *gets* (shown in Figure 4) drops down overall compared to 1) Basic Evaluation. Especially in the case of FreePastry, it

drops down extremely and fluctuates greatly, which indicates that *lifemean* and *deathmean* of this scenario are too small for FreePastry. *Get* response delay (shown in Figure 5) also indicates that FreePastry does not work well compared to 1) Basic Evaluation.

3) Multi-Value: Figure 6 shows the percentage of successful *gets* with the number of *values* for each *key* (x-axis). FreePastry does not support multiple *values*, since the percentage remains 0% for all plots. Chord and Accordion seem to support multiple *values*, but the percentage is very low compared to 2) Join/Leave Evaluation. Bamboo works pretty well up to 20 *values* per *key*, but the percentage becomes almost 0% at 50 *values* per *key*.

4) Separate-Put-Get: Figure 7 shows network traffic with the ratio of the *put* group (x-axis). It remains stable for Bamboo, while it slightly decreases for the other three. We assume this decrease reflects that the total packet size for *put* is smaller than *get* for Chord, Accordion, and FreePastry.

5) Separate-Put&Get-Dummy: Figure 8 shows the percentage of successful *gets* with the ratio of the *put/get*

group (x-axis). The percentage remains high (more than 90%) for all DHTs, and FreePastry is the most stable.

4.4 Overall Results

Our findings with evaluations are described in the following.

Bamboo works most stably for all evaluations, and note that the *get* response delay is the smallest. However, network traffic is rather big; hence, it is appropriate for applications that require stability without bandwidth limitation.

Chord and Accordion achieve far smaller network traffic compared to Bamboo, but the *get* response delay is bigger. Accordion works better than Chord for all evaluations, and it is suitable for applications that require low bandwidth.

FreePastry works fine in a *static* model or if the group of nodes that does *puts* and *gets* is small. Hence, it might be usable for applications in a static network.

5 Summary and Future Work

We proposed an emulation method for evaluating DHT implementations and built a platform for our method. The platform allows running DHT implementations without modifications in a large-scale virtual network.

Evaluations by emulation point out facts that may not be found by simulation. For example, Bamboo and FreePastry are based on the same algorithm “Pastry,” but our results show that their performances are greatly different.

Scenarios that drive our platform are so straightforward that new evaluations can easily be performed by only generating new scenarios. Future work should extend the scenario generator to support new features, such as random gateway, where a newly joining node picks up a random gateway instead of a static gateway.

Other future work includes a) coping with ModelNet to emulate a router network, and b) evaluating other services in overlay networks, such as Multicast and Search. We hope DHT evaluations by emulation will contribute to further development of DHT applications.

References

[1] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Struc-

- ured Peer-to-Peer Overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, February 2003.
- [2] <http://freepastry.org/>.
- [3] <http://pdos.csail.mit.edu/p2psim/kingdata/>.
- [4] <http://www.bittorrent.org/>.
- [5] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, February 2004.
- [6] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth efficient management of DHT routing tables. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, May 2005.
- [7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, March 2002.
- [8] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the 2004 USENIX Annual Technical Conference (USENIX '04)*, June 2004.
- [9] A. Rodriguez, C. Killian, S. Bhat, D. Kotic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, March 2004.
- [10] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.
- [11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM '01 Conference*, August 2001.
- [12] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kotic, J. Chase, and D. Becker. Scalability and Accuracy in a LargeScale Network Emulator. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.