How to Run Experiments with Large Peer-to-Peer Data Structures

Erik Buchmann, Klemens Böhm
Otto-von-Guericke Universität, Magdeburg, Germany
{buchmann|kboehm}@iti.cs.uni-magdeburg.de

Abstract

Distributed Hash Tables (DHT) promise to administer huge sets of (key, value)-pairs under high workloads. DHT currently are a hot topic of research in various disciplines of computer science. Experimental results that are convincing require evaluations with large DHT (i.e., more than 100,000 nodes). However, many studies confine themselves to (less convincing) experimental examinations with much fewer nodes. Information on how to run experiments with DHT with many nodes is not available. Based on experience gained with a DHT implementation of our own [4], this article describes how to carry out such experiments successfully. The infrastructure used is a cluster of 32 commodity workstations. The article starts by compiling requirements regarding such experiments. We then identify the various bottlenecks that may be the result of a naive implementation, and we describe their negative effects. The article proposes various countermeasures, e.g., an experiment clock, and a component that maintains persistent network connections between cluster nodes. The features proposed are beneficial: A naive experimental setup allows for 10,000 peers maximum and a total of 20 operations per second, a sophisticated one following our proposal for 1,000,000 peers and 150 operations per second. Furthermore, we say why experimental results gained in such a way are meaningful in many situations.

1 Introduction

Distributed Hash Tables (DHT) promise to administer huge data sets and to cope with very high workloads in a decentralized manner. The number of applications is large, in the context of the WWW and elsewhere. DHT follow the Peer-to-Peer (P2P) paradigm [11]: They consist of independent nodes (peers) that form an overlay network on top of the Internet. Each node acts as server and client at the same time and is responsible for a certain part of data. There are many open questions in the DHT context with regard to load balancing, reputation management, query processing, etc. Researchers are coming up with new proposals at a high rate. With most of such proposals, experimental evaluations are enlightening.

The claim behind DHT is Web scalability. Thus, experiments that are instructive should run in a setup with a large number of independent peers – 100,000 and above. Running such experiments is difficult, be it on a single machine, be it on a cluster of workstations. One difficulty is on the *methodology* level – there currently does not even seem to be a common understanding regarding the requirements on and the characteristics of experiments with DHT. On the *technical* level, various hardware and software limitations are in the way of simply starting many peers in an experimental setting. Examples of these limitations are the boundedness of the numbers of parallel tasks or of parallel network connections per machine. Another difficulty is the development of logging and accounting facilities, adopted to the special needs of DHT. For these reasons, experiments are frequently limited to simulations with some hundred nodes in a simplified environment. However, this does not allow to verify the original claim. Furthermore, one should be able to use the system not only for large-scale experiments – it should be operational as well, in order to run real applications. This requirement, i.e., being able to use the system both for experiments as well as for real applications is difficult to meet. The reason is that the first kind of deployment requires specific tuning measures, while applications require a prototype for stand-alone execution. The article will illustrate this.

We for our part have conducted extensive experiments with a Content-Addressable Network (CAN) [12] implementation of our own with 100,000 to 1,000,000 peers [4]. (A CAN is a prominent variant of DHT.) Our concern is not to describe these specific experiments. Rather, we want to say what one can do in order to carry out experiments with P2P data structures on that scale. Even though we will mention some features of our proprietary CAN implementation, we only do this to ease presentation. The concern of this article is much broader – it should be helpful to anybody trying to run *arbitrary* experiments in a setup similar to the one hinted at so far. The environment we have used is a small cluster of off-the-shelf workstations, a setup that is becoming more and more common. Our cluster consists of 32 PCs equipped with 2 GHz CPU, 2 GB RAM and 100 MBit Ethernet. Our CAN variant is written in Java.

Clearly, the operating-systems and distributed-system communities have proposed a huge number of measures to improve performance of distributed systems and to cope with hardware and software limitations. This article in turn answers the following questions: Taking into account that the prototype should allow for both large-scale experiments as well as real applications, which ones of those measures are relevant? Which ones should be used when it comes to experiments with P2P data structures? What exactly is the impact on performance? These questions have aroused much interest – the group of people doing research work on P2P data structures is large and is growing, in various research communities. In more detail, our contributions are as follows: First, we list requirements regarding experiments with P2P data structures. We then identify various hardware and software bottlenecks that may arise with a naive implementation, and we describe their negative effects. To counter this, we have designed and implemented an experimental framework with several new components, e.g., a so-called *peer manager* that runs on each cluster node[1]. The peer manager allows to share persistent network connections and to bundle threads. Another essential component of our framework is an *experiment clock*, which we will motivate and describe. Furthermore, it is important to keep the code clean from modifications done for experimental purposes only. A precompiler is a simple mechanism to accomplish this. – Again, even though we have used Java 2 SDK 1.4 on Linux, we think that our propositions described here are useful for other runtime environments and settings as well. Our results are positive: while a naive setup would allow for 10,000 peers maximum and 20 operations per second issued by all peers, a setup incorporating our measures allows at least 1,000,000 peers and 150 operations per second.

The remainder of the article is organized as follows: Section 2 describes prerequisites and related work. In Section 3, we discuss requirements on an experimental environment intended to run a large number of peers. In Section 4 we identify potential software and hardware boundaries. Section 5 shows how one can overcome them. Section 6 describes further points necessary to carry out the experiments successfully. Section 7 concludes.

2 Prerequisites and Related Work

Peer-to-Peer (P2P) technology for data management has received much attention in the recent past. The reasons are that it promises Web scalability, avoids single points of failure, and has the potential to distribute the infrastructure costs in a fair way by assigning work to many nodes maintained by independent carriers. *Distributed Hash Tables (DHT)* are an example of this technology, allowing to manage huge volumes of data with a hash table interface. The data are (key, value)-pairs. A key addresses each operation, such as `insert`, `delete`, `query`. A peer invokes an operation on another peer by sending it a respective message, and in what follows, we may speak of the *key of a message*. In analogy to buckets of a hash table, each peer is responsible for a certain part of the key space and knows some other peers. A peer can either process an operation itself or forward it to another peer that is more likely to be able to process it. Note that DHT messages are 'high-level' messages addressed by keys. This is orthogonal to Internet datagrams addressed by IP numbers. Thus, a DHT is an overlay network that is independent from the underlying physical topology.

Content-Addressable Networks (CAN) [12] are an important, well-known instance of DHT. The key space is a torus of Cartesian coordinates in multiple dimensions. In addition to its (key, value)-pairs, a CAN node also knows its neighbors, i.e., nodes responsible for adjacent parts of the key space. A peer forwards a message to another peer by using the Greedy Forwarding Protocol. Greedy Forwarding means that the distance between the key of the message and the key space of the current peer must decrease with each forwarding step. Other DHT use concepts similar to the ones of CAN and will profit from experience with large experiments as well. The DHT variants differ primarily in the topology of the key space and the message passing algorithm [8], but not in the interfaces. *LH** [10] computes the node responsible for a certain key statically by a given hash function. *Chord* [14] organizes the data in a circular one-dimensional key space. *Pastry*, *Tapestry* [13, 15] use a Plaxton Mesh to store and locate the data.

CAN leaves room for improvements in various respects. In [4] we proposed that each peer implements a *contact cache* storing a number of further contacts in addition to its neighbors. This cache is adaptive, i.e. peers may select contacts according to their requirements. As long as a peer knows only its neighbors, messages travel along routes that are static. Following our proposal, experiments [4] show that even a small number of additional contacts decreases the average number of forwarding steps of a message significantly. For instance, a contact cache storing 20 peers in addition to the neighbors reduces the forwarding steps in a CAN of 100,000 peers by more than $^2/_3$. Therefore, evaluation of DHT structures and behavior evolving over time is a relevant topic.

So far, evaluation of DHT is frequently based on simulators. While there are general discrete event simulators like

[1]In what follows, we will refer to cluster nodes as 'host', whenever 'node' is not sufficient to distinguish between 'host' or 'peer'.

simjava [9], other ones are more specialized. For example, the Network Simulator ns-2 [3] or the Stanford Narses simulator [6] target at networking research. However, this is only useful for abstraction levels lower than the DHT protocols. At higher levels, there do exist frameworks as well, but mostly for specific aspects like incentive sharing [5] or document routing in a swarm-like fashion (Anthill [2]).

A recent approach more similar to our topic is JXTA [7]. JXTA defines a network programming environment useful for DHT applications. JXTA is not based on DHT. Instead, every peer propagates resources it wants to share to a so-called Superpeer in a random graph network model. In addition, there is a load-test framework [1] that provides a set of test suites for discovery of new peers, security, and communication between peers. Like our proposal, this test suite tries to limit the resources necessary by running several peers in the same JVM.

3 Experiments with DHT – Requirements

In this article, we differ between *simulations* and *experiments*. In our context, simulations attempt to predict the behavior of DHT data structures by providing an approximate model. In contrast, experiments require a prototype that is fully functional, and that is written in a general-purpose programming language. Following our line of thought, the main difference between a prototype used for experiments and a ready-to-market program is the omission of some features (e.g., Graphical User Interface) that are not needed to assess the performance characteristics of the system.

From our perspective, experiments with DHT have some strong advantages over simulations and formal analyses. The latter can become extremely complex, and the number of parameters and 'tuning knobs' easily exceeds several dozens. Since simulations target at a simplified model, it is extremely difficult to say in advance which features are less important when it comes to reduce reality. Both formal analyses and simulations have difficulties with the estimation of resource consumption. Experiments in turn show roughly the same behavior with regard to most of the features we are interested in as a system that is operational. Finally, removing errors from the prototype code and improving the prototype enhances applications immediately.

Having said why experiments are a viable approach to evaluate DHT, we now list the most important requirements on such experiments. Identifying and discussing them is not only useful with regard to the methodology behind the experiments, it is even more important when it comes to the design of an experimental framework.

Repeatability Experiments must be repeatable. This is necessary to assess the effect of changing parameter values or algorithms. Given the complexity of DHT implementations, it also eases debugging and finding errors significantly. In order to repeat experiments with DHT, we must be able to generate the same initial state of the DHT whenever necessary. With 'same initial state of the DHT', we refer to the following DHT characteristics:

Same key spaces With some DHT, the assignment of a peer to a portion of the key space takes place randomly when the peer joins the DHT. Since different assignments of peers to portions of the key space would lead to different forwarding paths and would be in the way of repeatability, our experimental framework must be able to store and recover such an assignment.

Same initial contact information With DHT, each peer 'knows' some other peers, in order to forward messages whenever necessary. With LH* for instance, these 'contacts' directly result from the hash function that maps keys to network addresses. In standard CAN, the initial 'contacts' of a new node result from the current assignment of peers to portions of the key space and from the peer where the new node joins the network. Furthermore, each peer of our CAN implementation has some further contacts in addition to the 'adjacent' peers with standard CAN. On the one hand, this allows to improve the routing performance significantly. On the other hand, it makes the generation of a DHT state identical to a previous one more complex.

Our experimental environment will store this information in a *configuration file*. – To ensure repeatability of experiments, the experimental framework must also be able to run the same sequence of operations repeatedly. I.e., each peer issues the same operation at the same time with the same target key as before. Note that this does not mean that a repeated message travels along the same forwarding path – changed parameter values or modified algorithms may of course change the behavior of the network.

Dynamicity Like any P2P system, a DHT is a dynamic structure. Peers constantly enter and leave the network. With some DHT, the portion of the key space assigned to a peer may change over time. Furthermore, a peer may adapt its view of the world, e.g., by replacing its contacts with other peers. Consequently, experiments must not be confined to individual operations. Instead, they also must examine the effects of processing operations in parallel or consecutively as well as the evolution of system characteristics over time.

Performance Another important goal is to process many operations in short time. As long as a single peer is running on a single host, most hardware resources (except maybe for the network interface) are consumed by the applications and are unlikely to become a bottleneck for the underlying DHT infrastructure. But as soon as some thousand peers are running on the same host, hardware and operating-system limitations will arise, and one must ensure that experiments run with acceptable performance (see Section 4 for a more detailed description of the bottlenecks).

Note that our concern is to speed up the experiment as a whole, as opposed to optimization of individual peers. For instance, proposals such as execution of one peer per CPU on SMP servers or moving the storage layer from the server to a SAN are orthogonal to our current topic.

Logging and Control Obviously, a logging service is an essential component of the experimental framework envisioned. It must enable us to evaluate the performance characteristics both of individual peers as well as of the system as a whole. This must hold even if the system consists of many peers. The characteristics to be observed include the forwarding paths of messages, the contact lists of individual peers and the ways they change over time, and internal parameters that quantify the workload of individual peers, the amount of data administered etc. Finally, this logging service must be as thin as possible to avoid side effects.

Another important requirement is to control the system load of all hosts. In general, different peers process operations at different rates. Without further measures, some hosts will soon experience an overload or will become idle. In consequence, the experimental framework should feature an *experiment manager* component. It adapts the speed for the whole experiment to the current load situation of the hosts without distorting the relations between peers.

Small changes on source code In contrast to the previous requirements, this current requirement does not refer to characteristics of the experimental setup itself. The requirement is that the code running in the experimental framework envisioned shall differ as little as possible from the one of a system that is operational. If the same code base is used, removing errors from the prototype code and improving the prototype leads to direct enhancements of applications.

4 Hardware and Software Limitations

A naive approach to experiments with DHT, which resembles our first attempts to evaluate large DHT experimentally, might look as follows. Each peer contains a logging facility that records incoming messages, together with a timestamp. Furthermore, each peer has a random-number generator, in order to issue operations with random keys at random points of time. A script starts as many peers as possible on all machines of the cluster. Each peer now joins the DHT, one after the other, as specified by the DHT protocol. In order to forward messages, peers establish network connections and disconnect on demand. – Unfortunately, all this does not scale, due to the following problems: If peers do not reduce the rate of operations issued, the environment will soon raise exceptions like ”Too many open Connections” or ”Could not start Thread”. Furthermore, even though some machines are overloaded, others are idle all the time. In addition to this, the number of peers that can run on one machine is too small. The reasons are various hardware and software limitations. They need to be identified and eliminated, in order to run meaningful experiments.

CPU Limited computing power manifests itself as follows: the host is unable to process incoming messages as quickly as they arrive. After a short period of time, any queue or network buffer has an overrun, and the experiment breaks down. Such overloads are difficult to avoid, since other hosts generate the load randomly – at least, this is the local perspective of an individual peer. Consequently, it would be good if there was a centralized control component that is allowed to adopt the rate of operations issued by the various peers. Note that such a component does not contradict the peer-to-peer nature of the system under evaluation. It is merely the experimental framework that is not peer-to-peer any more.

Main Memory Normally, a single peer on one host should not exhaust the main memory. But with peers written in a high-level programming language, starting the runtime environment for each peer leads to a significant memory consumption and limits the number of peers per host. The programming language of our choice is Java, and a Java Virtual Machine (JVM) consumes between 2 and 64 MB main memory by default.[2] If the JVM is short of memory, it invokes its garbage collector very frequently, and performance becomes worse.

Timer A very hidden and nearly undocumented limitation comes from the timer. Our CAN implementation uses it for various tasks, e.g., to delay messages. Furthermore, we have experimented with reputation mechanisms where reputation times out after a certain time period. This requires the use of the timer as well. Despite the Java feature to schedule timer events with nanosecond precision (`java.lang.Object.wait(long timeout, int nanos)`), the resolution used depends on the hardware and operating system. For instance, the JVM 1.4.0 shipped from Sun seems to use the system timer. On our

[2]One could question the necessity of using Java to implement the peers. However, application-development complexity in our context and portability issues lead to powerful, but easily usable tools and programming libraries.

machines running x86-Linux, the system timer resolution defined in `include/asm-i386/param.h` is 100 Hz. Other machines or operating systems use different time resolutions. This can result in strange effects during experiments. For instance, the minimum time interval between operations is 10 ms on our Linux machines, but may have other values on other systems. In fact, our machines will process an event timed for 1 ms in 10 ms, other machines may use other times. In many situations, we do not know this value – one reason is that we may not have access to the source code. Thus, explicitly controlling the rate of operations during experiments is problematic.

Network In our setting, the most critical resource is the network connection (TCP/IP over 100 MBit Ethernet). It restricts experiments in various ways: latency, the number of parallel open connections, and the number of available ports. We will now explain these points – they are strongly interwoven. Because ports from 0 to 1023 are reserved for daemons and services, the range of ports usable for non-root users on x86-Linux is from 1024 to 65535. With the naive setup, each peer binds to one port. Every time a peer wants to send a message to another one, the operating system occupies a free port and establishes a connection between that port and the port of the target peer. After the communication has taken place, the previously occupied port remains in the `TIME_WAIT` state for some OS-dependent time. Given the high number of parallel operations in our setup, a host may run out of available ports. The `connect` operation then raises an Exception, and the experiment has to stop until some ports are freed. In addition, a huge number of parallel connections (we have counted more than 20,000 with `netstat | wc -l`) puts a heavy burden on the TCP/IP Stack maintained by the operating system. Furthermore, a connection in a TCP/IP network is established by a three-way handshake. Disconnection is by means of three or four further packets. Thus, connecting and disconnecting causes a significant amount of traffic overhead and needs some milliseconds. This latency limits the number of sequential operations of a single peer to a few per second (cf. Table 1).

	Parallel network connections	Parallel Threads	Operations per second
unoptimized	$>$20,000	$>$2500	20
optimized	62	45	150

Table 1. Comparison of unoptimized and optimized code

A naive approach for solving network problems may be the use of UDP instead of TCP. UDP is a connectionless protocol without handshaking mechanisms or state management datagrams. This reduces the overhead of the network protocol significantly. But UDP has two drawbacks with DHT: first, UDP datagrams are limited in size. Without protocol-inherent fragmentation, our DHT must split larger messages and recombine them ’manually’. Second, in the absence of connection management, datagrams may disappear without any notice. In contrast, TCP uses an internal recovery algorithm to make sure that datagrams arrive at their target host, or the sender gets an error.

System Software Not all limitations are hardware limitations. There is a software-related problem as well. When using x86-Linux, the number of threads available for applications is a bottleneck. In general, the OS scheduler allows non-root users to use only half of the value of `/proc/sys/kernel/threads-max` threads.

```
max_threads =
    mempages / (THREAD_SIZE/PAGE_SIZE) / 8;
init_task.rlim[RLIMIT_NPROC].rlim_cur =
    max_threads/2;
init_task.rlim[RLIMIT_NPROC].rlim_max =
    max_threads/2;
```

Figure 1. Thread Limit (Part of kernel/fork.c)

The value is calculated in `kernel/fork.c` as shown in Figure 1 and depends on the available main memory. Values typically range between 4096 and 16384. Next to the kernel, the `glibc` shared library also limits the number of threads. The line `#define PTHREAD_THREADS_MAX 1024` in `bits/local_lim.h` allows only 1024 threads usable by a program, e.g., a single JVM. Because every peer starts new threads to process incoming messages, the thread limit is in the way of running more than a few hundred peers per JVM.

Number of Threads	Duration in s Linux 2.4.18	Duration in s Windows XP
1	86	62
10	99	66
100	152	211
1,000	121	238

Table 2. Computing the Rabin Hash Function 1,000,000 times with different numbers of threads

Furthermore, the OS scheduler is not optimized for running a large number of parallel threads. A simple test shows that thread scheduling consumes many CPU resources. As shown in Figure 2, we calculated the Rabin Hash Function one million times for a given String with different numbers of parallel threads. The first line of Table 2 stands

```
class Test {
  int c = 100;

  Test() {
    for (int i = 0; i < c; i++) new Helper();
  }

  class Helper extends Thread {
    Helper() {
      start();
    }

    void run() {
      for (int i = 10000000 / c; i > 0; i--)
        RabinHashFunction.getDefaultHashFunction()
          .hash(this.toString().getBytes());
    }
  }
}
```

Figure 2. Estimate the effect of the number of threads

for one thread having carried out all one million computations. Another run was such that 1,000 threads carried out 1,000 calculations each. Table 2 shows that the time to carry out the computations increases by 40 and 280 percent with Linux and Windows XP, respectively, by using 1,000 parallel threads.

5 Eliminating the Bottlenecks

To overcome the limitations from the previous section, we have developed an experimental framework that consists of three components, as shown in Figure 3. A Class `PeerManager` controls resources for all peers running on a certain host. A control component called `ExperimentManager` that is centralized manages the experiment itself. For instance, it tells each peer which query to issue at what time. Finally, a central logging facility named `LogConsole` is responsible for recording events monitored on all peers, together with timestamps. This section will describe these components. Section 6 will discuss further steps necessary to run large experiments, which do not exactly fit under the umbrella 'bottleneck elimination'.

Our CAN implementation is a stand-alone program, to be used by any application. Integration of the experimental framework requires some modifications of the source code. ***Small Changes on Source Code*** states that such modifications must not remain in the code shipped to application developers. Thus, we mark up experiment-related code with precompiler-statements like `#ifdef`, `#else`, `#endif`[3].

[3]Even though there are more powerful ways to accomplish this, for example Aspect Oriented Programming, we have settled for a precompiler because of its simplicity and adequateness.

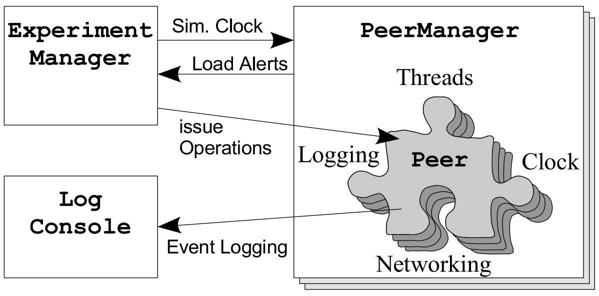

Figure 3. Experimental framework architecture

```
# Input files are located in ./base/p2pnet
# Output files are written to ./p2pnet
createoutputdir:
    @echo "removing old output directory"
    rm -rf ./p2pnet
    @echo "creating new directory tree"
    mkdir -p ‘find ./base -name "*.java" |sed -e \
      "s/^\.\/base/./;s/\/[a-zA-Z1-9]*\.java//"‘

derive_experiment: createoutputdir
    @echo "deriving code for experiment purposes"
    cd ./base ; find p2pnet -name "*.java" |      \
      xargs --replace=X gcc -E -nostdinc -I- -C  \
      -P -D _experiments_ -x c X -o ../X
```

Figure 4. Makefile to separate experiment-related code

The problem is that Java does not support such precompilation. Fortunately, the current `gcc` allows to parse files in other formats. The flag `-x c` tells the precompiler to treat input files as if they were written in 'C'. This allows for precompiler statements. Note that this is feasible only because of the narrow similarity of the syntax of C and Java – for instance, if '#' was a Java symbol, all this would not work. Figure 4 shows a part of our `Makefile` to separate experiment-related code from system code. The statement `make derive_experiment` simply creates a new empty directory tree and uses the `gcc` to parse all java files from the `./base/p2pnet` directory into `./p2pnet`.

In general, we can reduce resource consumption by lowering parameter values and by using smaller data objects, for instance changing peer names from fully featured String Objects to sequential numbers stored in an Integer. But simple solutions like this will not solve all those problems. We now say how we have overcome those various limitations.

Several Peers in one JVM Because every JVM comes with some further functionality like Garbage Collection, running each peer with its own JVM consumes a lot of free memory and computing power. Running several peers with one JVM allows to overcome the Bottlenecks ***Main Memory*** and ***CPU*** in a relatively simple way. One has to make sure that the peer class can be invoked from outside by calling its constructor, and that shutting down a peer does not shut down the whole JVM. This would happen with the usual `System.exit()` call. Besides that, one has to ensure that the source code does not contain static objects that are not intended to be shared between peers. Class `PeerManager` can now read the command line parameters identifying a configuration file and create the peers. Figure 5 illustrates this by means of some sample code.

```
class Peer extends Thread {

  boolean standalone = false;
  boolean isRunning = true;

  static void main(String[] args) {
    standalone = true;
    new Peer(args);
  }

  Peer(String[] args) {
    ...
  }

  void shutdown() {
    ...
    isRunning = false;
    if (standalone) System.exit(0);
  }
}

class PeerManager {

  PeerManager() {
    FileReader in = new FileReader("start.dat"));
    String s = in.readLine();
    while(s != null) {
      new Peer(stringToArray(s));
      s = in.readLine();
    }
  }
}
```

Figure 5. Running several peers with one JVM

Persistent network connections As described above, connecting and disconnecting network sockets for each transferred message consumes a lot of time, causes unnecessary network overhead and leads to a large number of parallel connections maintained by the OS. Our way around the Bottleneck ***Network*** is to use persistent connections, i.e., connections are established and kept open during the experiment. But while persistent connections from one peer to its most important contacts are feasible with one peer per host, this is not possible for some thousands of them. Thus, we have built a *peer manager* that controls all connections to and from peers running on the particular host. An important requirement is that the peer manager does not affect the 'natural' behavior of peers in any way. One can achieve this as follows: suppose that the peer class owns a method named `send(Message, Target)` to send messages and one named `receive()` to wait for incoming ones. `receive()` invokes a third method `process(Message)`.

```
class Peer extends Thread {

  void receive() {
#ifndef _experiments_
    ServerSocket s = new ServerSocket(port);
    while (isRunning) {
      process(s.readObject());
    }
#endif
  }

  void send(PeerDescriptor p, Message m) {
#ifdef _experiments_
    PeerManager.send(p, m);
#else
    openSocket(p).writeObject(m);
#endif
  }

  void process(Message m) {
    ...
  }
}

class PeerManager {
  Map peers;
  Map connections;

  void receive() {
    ServerSocket s = new ServerSocket(port);
    while (isRunning) {
      Message m = s.readObject();
      Peer p = peers.get(m.targetPeer);
      p.process(m);
    }
  }

  void send(PeerDescriptor p, Message m) {
    if (peers.contains(p)) {
      Peer n = peers.get(p);
      n.process(m.deepClone());
    } else {
      Socket s = connections.get(p.address);
      s.writeObject(m);
    }
  }
}
```

Figure 6. Connection management

Let us first consider the case that the peer is standalone. This happens if the peer is part of a system that is operational. Then `send()` opens a new connection for each message, sends it, and closes the connection afterwards. For experiments on the other hand, we replace that piece of code with a call to Class `PeerManager`. It does the networking for all peers in the JVM. The class maintains persistent connections to all other instances of this class running on other hosts, and passes the message to the one responsible for the peer addressed. As soon as a `PeerManager` receives a message, it extracts the name of the target peer and invokes `process(Message)`. Figure 6 shows all related parts of our source code.

Class `PeerManager` handles connections between peers running in the same JVM internally, without invoking network sockets. But simply transferring the message internally to the target peer has side effects: in Java, it is not the content of data object that is transferred, but the object ID. If the first peer changes the message object after having sent it, the object is altered at peers who have received it as well. One way around this problem is to deep-clone the message. In Java, `Object.clone()` only makes copies of the object itself, not of other objects contained in deep structures like vectors or maps. So one has to implement a deep clone oneself. A general way to do this is to serialize the whole message to a `ByteArray` stream and read it. Furthermore, addressing a target peer by its host name and port number is ambiguous if the peer manager receives messages for a lot of peers by using only one address. Thus, each peer must have a globally unique name in addition.

Controlling the Number of Threads Up to now, each peer is an instance of `Thread`. The rationale is to prevent the peer from blocking the application while waiting for incoming messages. This allows to run a few hundred peers in one JVM (around half of the glibc limit), and a few thousand if we depended only on the OS limit per host. To increase the number of peers, i.e., to do away with Bottleneck ***System Software***, we have to utilize the threads available as much as possible. We therefore propose the following: We observe that peers in our experimental framework only process incoming messages. They even do not issue queries themselves any more. Instead, the experiment manager sends them messages asking to invoke a particular query at a certain point of time. (Section 6 will provide more details on this feature.) Having said this, it is not necessary that a peer is constantly up and running. Instead, the peer manager now generates a thread corresponding to a particular peer only if there is an event which the peer must react to. This requires that the peers do not inherit from `Thread`. Furthermore, they may not invoke its method that listens on a network socket and waits for connections. Note that this optimization is only feasible within our experimental framework. Namely, it is only here that there is a peer manager responsible for the creation of new threads, instead of the peers themselves.

The modifications described so far are only 'minimally invasive'. Nevertheless, the savings with regard to resources and the total number of DHT operations per second increase significantly, as shown in Table 1. We have obtained these values regarding connections and threads from a single host communicating with 31 other hosts in the cluster. Operations per second refers to the experiment as a whole. While the numbers are of course experiment-specific, one can expect similar improvements in other contexts.

6 Managing Experiments

Eliminating the bottlenecks on a rather technical level is not yet sufficient in order to run large experiments with DHT. The current section will show this, together with a description of the measures necessary.

Startup of Experiments Requirement ***Repeatability*** states that the experimental setup must allow to rerun the same experiment from exactly the same system state. If the experiments do not require that the DHT actually maintains data at startup, peers (be it with CAN, be it with most other DHT architectures following the Document Routing Model) are sufficiently described by their portion of the key space and their contact list. As described in the last section, the peer manager is capable of starting an arbitrary number of peers. It does so by reading the configuration file and generating command line parameters. To achieve a repeatable startup, the configuration file also contains parameters that specify key spaces and peer contacts of individual peers.

Next to repeatability, being able to start experiments from a certain configuration has further advantages. For instance, this allows to create specific distributions of key spaces in order to focus on load balancing mechanisms. As another advantage, it allows to speed up the startup process. This is worthwhile if one wants to carry out experiments regarding the behavior of a running system. Our setup mechanism as part of the peer manager allows to bypass the conventional startup process, where peers join the DHT network [12]. Letting 100,000 peers join the network one by one using the join takes more than 90 minutes, in contrast to only five minutes when using our setup mechanism. Thus, another task for the precompiler is to switch between new nodes joining the network in the 'conventional' way and fetching all relevant information from the configuration files to start the network.

Issuing Operations It is not only the experimental setup that must be predictable and repeatable, but also the operations issued during an experiment. For example, specific distributions of key values, of arrival times of operations, and of the peers issuing them would allow to investigate DHT both in worst-case and in real settings. In this context, it is important that the code needed to run stand-alone applications remains unchanged in spite of possible changes to the prototype. Thus, we emulate a large number of operators using the same interfaces as real applications do. To ensure repeatability of experiments, we for our part have introduced a special type of message. The central experiment manager issues this kind of message. It instructs the target peer to carry out the operation attached to the message. The experiment manager reads a script file whose lines say which peer has to issue a certain operation. For example, one line of the script might be ”Peer 8632: issue a query for Point (0.2; -0.4; 0.4).”.

Load Supervision The average number of messages generated to evaluate a single query may change over time. For instance, [4] describes adaptation mechanisms for CAN and respective experiments. With these mechanisms, the number of forwarding steps of a query decreases significantly over time. If the rate of queries issued was always the same, machines would become idle. Performance of experiments would be low, and Requirement ***Dynamicity*** would not be met. Consequently, we want to speed up the global rate of queries issued. A naive way to do this would be to specify the rate of queries issued as a function of time elapsed. However, this is tedious, and the specification work needs to be carried out anew for each experiment. Monitoring the CPU load on every node in the cluster is not straightforward as well. The reason is that peers are mostly engaged in sending and receiving network packets: as long as a peer communicates, it does not consume any CPU resources. According to our experience, the most useful quantity to monitor for load supervision is the number of parallel threads active on each host. This parameter is easily observable by the peer manager. In our case, a peer manager sends an Alert Packet to the experiment manager if the number of parallel threads exceeds a certain limit – we use a value of 50 in our experiments. The experiment manager then decreases the global rate of new operations issued. On the other hand, if there are no alert packets arriving for some time, it increases this rate.

Experiment Clock Running experiments at an adaptive rate prevents us from thinking in absolute times, i.e., we should not use the system clock. For instance, message expiration after a certain fixed period of time would be counterproductive. This is because the number of operations taking place during this period of time is not predictable and typically varies. This is why we use the count of operations issued as experiment clock. The experiment manager program sends this number every few operations (we use a value of 100 to save network overhead) to each `PeerManager` instance through a network connection. Each peer manager puts it in a public, static attribute variable. All peers hosted by this class can access it. If a peer does not run within our experimental setup, it must again obtain the system time – the code fragment in Figure 7 accomplishes exactly this.

```
#ifdef _experiments_
  long l = PeerManager.currTime;
#else
  long l = System.currentTimeMillis();
#endif
```

Figure 7. Switching between experiment clock and system clock

Discussion Our experimental framework allows to focus on relevant aspects of DHT behavior and gives us verifiable, accurate results. Side effects of the framework on the behavior of peers are small: the chronological order of messages arriving a certain peer may be affected a little. The reason is that message transfer between peers running on the same host is faster than the transfer via network connections. But compared to the situation found on common Internet connections, we can leave aside these effects.

The first series of experiments with our prototype has measured the benefit of contact caches [4], as mentioned in Section 2. Without our experimental framework, the results were quite unsatisfactory, i.e., only 10,000 peers and a few operations. With our framework in turn, only the available memory and the time needed for a meaningful number of operations limit the number of peers. Some of the results obtained with regard to contact caches are interesting and were unexpected for us: First, a very small cache size already yields a significant benefit. The simple replacement strategy RANDOM is competitive with more sophisticated ones like LRU, LRD or LFU. (See [4] for more information and explanations.)

7 Conclusion

Given that peer-to-peer data structures are important, experiments to evaluate new ideas and proposals in this field are important as well. This article has made a strong point for large experiments in DHT networks. It has provided a clear list of requirements that such experiments and experimental frameworks should fulfill. We have described bottlenecks that materialize when approaching the problem (the one of carrying out large experiments) in a naive way.

The article has then introduced countermeasures that help to get rid of these limitations. These countermeasures do not affect those parts of the prototype needed by real-world applications. The countermeasures include the use of an experiment clock and adaption of the rate of operations issued. The results of our measures are pleasingly good: A naive experimental setup allows for 1,000 peers maximum and 20 operations per second, a sophisticated one following our proposal in turn allows for 1,000,000 peers and 150 operations per second. We point out that this is not yet the 'end of the road'. We are confident that we could have many more peers and approximately the same experiment runtimes without encountering a bottleneck if we had a bigger cluster available. (The cluster currently consists of 32 hosts.)

Future work will deploy the experimental setup described here for sophisticated experiments with DHT.

References

[1] Bench Project: A JXTA Load Test Framework. http://bench.jxta.org/LoadTest.html, 2003.

[2] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems, 2002.

[3] L. Breslau et al. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[4] E. Buchmann and K. Böhm. Efficient Routing in Distributed Scalable Data Structures (in German). In *Proceedings of the 10th Conference on Database Systems for Business, Technology and Web*, Feb. 2003.

[5] C. Buragohain, D. Agrawal, and S. Suri. A Game-Theoretic Framework for Incentives in P2P Systems. In *Proceedings of P2P2003, Sept. 1-3, Linkoping, Sweden*, 2003.

[6] T. Giuli and M. Baker. Narses: A Scalable Flow-Based Network Simulator. Technical Report cs.PF/0211024, http://arXiv.org/, Nov. 20 2002.

[7] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5:88–95, 2001.

[8] K. Gummadi et al. The impact of dht routing geometry on resilience and proximity. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394. ACM Press, 2003.

[9] F. Howell and R. McNab. simjava: A discrete event simulation library for java, 1998.

[10] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - Linear Hashing for Distributed Files. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 327–336. ACM Press, 1993.

[11] D. S. Milojicic et al. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, Palo Alto, Mar. 2002.

[12] S. Ratnasamy et al. A Scalable Content-Addressable Network. In R. Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 161–172, New York, Aug. 2001. ACM Press.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, 2001.

[14] I. Stoica et al. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, New York, Aug. 2001. ACM Press.

[15] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: an infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, Apr. 2001.