Benchmarking DHTs with Queries

David Oppenheimer, Joseph M. Hellerstein, Ryan Huebsch, and David A. Patterson
University of California at Berkeley, EECS Computer Science Division
{davidopp, jmh, huebsch, pattrsn}@cs.berkeley.edu

Abstract

The recent proliferation of decentralized distributed hash table (DHT) proposals suggests a need for DHT benchmarks, both to compare existing implementations and to guide future innovation. We argue that a DHT-based query engine provides a unified framework for describing workloads and faultloads, injecting them into a DHT, and recording and analyzing the observed system behavior. To illustrate this argument, we describe the possibilities and challenges of using one such DHT database engine, PIER, to describe and instantiate network dataflow patterns, and to measure and report the resulting system performance. Together, these capabilities form the foundation of a benchmarking tool, while the distributed tracing and analysis facilities alone support debugging.

1. Introduction

Researchers have recently proposed decentralized Distributed Hash Tables (DHTs) as the fundamental building block for a new generation of large-scale distributed applications. Although much attention has been focused on the design of DHTs [2] [3] [4] [5], techniques for evaluating key properties such as their performance, incremental scalability, and robustness to failures have not kept pace. As a result, these systems are commonly evaluated using simulations or implementations that use artificial and simplified workloads and faultloads. As DHTs make the transition from theoretical endeavors to essential application building blocks, it becomes increasingly important to characterize the true performance, dependability, and scalability of DHTs. The devil is often in the details, and evaluation using standardized workloads and measurement criteria is essential to uncovering those details. Thus we believe that the community would benefit greatly from rich DHT benchmarks, both for designers to evaluate design tradeoffs and for users to compare systems.

DHTs are fundamentally content-based routing schemes, delivering (or fetching) data to (or from) a destination specified by a key value associated with the data. Hence a content-centric dataflow system is appropriate for describing and executing DHT benchmarks. Based on this observation, we argue that a DHT-based database query engine is a surprisingly convenient and flexible substrate for benchmarking DHT implementations. Database languages are handy for describing content, distributed database query processing is inherently based on content-based dataflow, and queries are useful for post-processing traces of system behavior.

In this paper we describe how a DHT-based query engine, along with an extensible version of SQL, can be used to express workloads and collect relevant metrics. In particular, we describe how the PIER system [1] can be used to describe a workload, to generate, stage and ship the content that constitutes the workload, and to record and analyze the DHT's behavior with respect to an extensible set of metrics.

Because much of the value of DHTs lies in their robustness to failures and attacks, it is important to evaluate them not just for their performance but also for their dependability. We therefore also mention briefly how our framework can be used to describe, generate, and insert “perturbations” into the system.

2. Context

Traditional database systems are sometimes used as a benchmark application, and they are often used to organize and analyze benchmark results. But we are unaware of past work that uses a query engine as a generic tool for *driving* a benchmark.

Because large-scale geographically-distributed hash table implementations are just starting to emerge, there are no canonical benchmarks or benchmarking techniques targeted to these systems. Existing studies use random workloads [2] [3] [4] [5], occasionally combined with workloads from one or two application domains [5]. Our goal in this early work is to develop a general benchmarking mechanism for DHTs that will allow many kinds of workloads to be expressed and generated using a very compact language. In future work we hope to study the myriad of policy details associated with designing good DHT benchmarks.

3. Elements of a DHT benchmark

We will assume that the elements of a DHT benchmark include a workload, a perturbation load, and metrics. The *workload* is a sequence of calls to the DHT API, which result in a network traffic pattern. This traffic pattern is implicitly defined by the initial distribution of data among nodes, additional data injected into the system by an application, the final destinations for those data items, and the profile of data movement over a time. Ideally, the benchmark consists of a suite of workloads, each representative of an application commonly built atop a DHT. The *perturbation load* is a sequence of out-of-band operations such as node addition and removal, changes in link characteristics, and node and network faults, that are periodically triggered as the workload is inserted. *Metrics* of interest for a DHT benchmark include failure-free performance (*e.g.*, throughput and latency in the absence of failure), performance and data quality in the face of various types of permanent and transient failures, load balance, utilization, and time to recover from failures.

4. PIER overview

The context for this work is the PIER query engine described in [1]. PIER is the "top half" of a distributed database system: a standalone distributed SQL query engine. PIER can access data via DHTs, or via an extensible iterator or "wrapper" that produces a stream of structured data from a local data source. As examples, a wrapper can be configured to scan through the names and metadata of files in a directory tree, to scan the contents of a given file, or to fetch records piped from a program with structured output, *e.g.*, tcpdump.

As in any distributed query engine, network traffic in PIER is often the result of joining distributed tables. To perform a distributed join of two tables *R* and *S*, a query engine must arrange for tuples of *R* to appear at the same node as their matches in *S*, a form of set-based rendezvous. PIER supports a number of different join algorithms. Salient to the discussion here are the *fetch-matches* and *put-matches* algorithms, which work when one relation (say *S*) is already stored in the DHT with that relation's join column used as the DHT key. In the fetch-matches algorithm, each node begins to scan its local *R* tuples, and for each tuple it calls the DHT **get()** function to fetch matching tuples of *S* from other nodes. In the put-matches scheme, each node scans its local fragment of *R* and calls the DHT **put()** function to send those tuples to the nodes with matching *S* tuples. While PIER supports other more general-purpose join algorithms, as well as other relational operators, these two algorithms will suffice for our discussion here.

5. The query engine approach to benchmarking

The primary operation performed by a DHT is routing a message from source to destination based on its key. We assume that a basic DHT API is built on top of this routing layer, in particular, a DHT API of

put(key, data): send <*key, data*> to the node responsible for <*key*>

get(key) -> data: retrieve the <*data*> associated with <*key*>

Although an eventual standardized DHT API will likely include additional functions, **put()** and **get()** are the fundamental hash table operations and are therefore the operations that we wish to benchmark. They are also the only network-based operations used by PIER to implement a query plan

Given this API, we wish to design a general query mechanism to generate calls to this API and thus DHT traffic patterns. The workload we generate is a function of the initial configuration of nodes, including the initial distribution if data among them, and the queries that cause invocation of **put()** and **get()** operations. How this DHT workload translates into an underlying IP network workload depends on network topology, network link characteristics, and various DHT details (*e.g.*, routing policy and how keys are mapped to nodes).

5.1. Generating the workload

We begin with a simple example to drive our discussion. Imagine the following query issued to PIER:

```
INSERT INTO localresults
SELECT *
FROM G, D
WHERE G.fkey = D.hashID
```

Assume that *G.fkey* is a foreign key from the *G* table to the *D* table (*i.e.*, each *G* tuple references some tuple of *D* via *G.fkey*). Assume further that we have created *D* to contain exactly one match for each *G* tuple, *i.e.*, *D* has one tuple for every value of *D.hashID* that is referenced by *G.fkey*. Assume that each tuple of *D* is stored as an entry in the hash table, with the *hashID* attribute as the storage key.

To process the above query, PIER can initiate a put-matches join on each node, scanning the *G* table on that node and calling **put(G.fkey, Gtuple)**. This query plan will have the effect of setting up a communication pattern: each tuple of G will, in turn, be shipped to the node responsible for DHT key *G.fkey* specified in that tuple of *G*, where it will be joined with the tuple's columns in the *D* table, and placed into local storage for a table called *localresults*. Because the *D* table has a single match for each *G* tuple, the join is guaranteed to generate exactly one result tuple for every arriving tuple of *G*. Note that the node with the *G* tuple knows the destination *D* node for the tuple, because the join attribute is the DHT storage key used by the D table.

From the above, it should be obvious how to construct a simple workload to exercise the DHT **put()** operation: to have node *X* **put()** tuple *data1* to node *N1*, tuple *data2* to node *N2*, and tuple *data3* to node *N3*, in turn, construct the G table on node *N1* so that tuple *data1* has *fkey=key1* such that *key1* maps to *N1*'s partition of the keyspace, tuple *data2* has *fkey=key2* such that *key2* maps to *N2*'s partition of the keyspace, and tuple *data3* has *fkey=key3* such that *key3* maps to *N3*'s partition of the keyspace. Two effects are produced here: (a) by controlling the order and *fkey* value distribution of *G* on each node, a **put()**-based traffic pattern is generated, and (b) the shipped tuples are recorded in the *localresults* table via the join with the *D* table. (In Section 5.4 we describe how the **get()** operation can be exercised using a PIER query.)

The *localresults* table serves to log the system's behavior. We therefore want a schema for *G* that has columns for every attribute of interest that PIER can usefully track during the benchmark run. For example, PIER can fill in local-time-sent and IP-address-of-sender columns of *G* when the DHT **put()** is invoked, and can fill in local-time-received and IP-address-of-receiver columns of *G* when the message is received by the *D* node.[1] If the DHT API includes the capability to make upcalls on intermediate nodes to allow an application to touch a message as it is being delivered to its destination, additional data can be recorded such as the IP addresses of all nodes that touch a message and the times when the message was touched.

Assuming that the tuples of *G* are tagged with the above statistics, when the benchmark run is finished, queries can be issued to the *localresults* table to compute benchmark statistics such as overall throughput (number of tuples handled divided by benchmark running time), latency for individual **put()** operations, *etc.* Given timestamps, changes in these metrics over the course of the benchmark run can be examined over time. Such an analysis is especially useful for benchmarking DHT dependability, *i.e.*, when perturbations are inserted and it is desired to measure the system's response to those perturbations. More generally, anything that is visible either at the PIER senders, the PIER recipients, or (depending on the DHT API) on intermediate nodes, can be appended to a tuple passing through the system, and any analysis of the result that can be expressed as a SQL query can be later requested. Thus we can use our query engine not only to generate a workload, but also to serve as a simple distributed database that collects execution data and directly generates desired benchmark metrics.

[1]While these columns are part of *G*'s schema, they are virtual in the sense that they are filled in by PIER as the query runs. They are not written to the *G* table until the join is processed at the *D* node.

Note that this mechanism for collecting and analyzing a trace of the DHT's operation can be useful not only for benchmarking, but also for collecting and querying an execution trace for debugging the DHT. For example, lost messages could be detected by comparing the contents of the results table after the benchmark run is finished, with the "correct" contents as specified by the original query and table values. The loss metric can be expressed as a SQL query.

Also note that we are assuming that the query plan generated by our DHT-based query engine behaves as we desire, *e.g.*, it chooses the correct join order and uses the put-matches join algorithm. To guarantee this, we must be able to "force a plan" on the query optimizer. There is syntax to do this in most commercial versions of SQL, and it would be reasonable to incorporate such syntax to indicate how a DHT-based query engine implements a benchmark-driving query.

Note also that this 2-table query involves a single join, which causes a set of one-DHT-overlay-hop communications. By adding more tables like *D* to the query, and by fixing the join order correctly, more hops can be introduced for each *G* tuple.

5.2. Using virtual tables

In the previous section we described a query that uses materialized *G* and *D* tables. A more efficient and convenient approach uses SQL table functions (virtual tables) instead of materialized *G* and *D* tables. We will call these functions **generator()** and **dest()**, respectively, making our query

```
INSERT into localresults
SELECT *
FROM generator(parameters...) AS G,
       dest(parameters...) AS D
WHERE G.fkey = D.hashID
```

Recall that the *G* table controls the contents and destination of the data sent via the DHT **put()** call.

Thus we can use the **generator()** function to generate and control the routing of the data without having to pre-compute and physically store the corresponding database data. Instead of being a materialized table pre-loaded into the DHT, the *G* table is available by calling a function at every node that generates the *G* table data on the fly. Likewise, we use the **dest()** function to avoid having to physically store the *D* table.

In the next sections, we sketch how the parameters to the **generator()** function can be used to specify workload characteristics such as initial data partitioning, final data destination, data size, and the pattern of data movement over the course of the benchmark.

5.3. Tuple attributes and data rates

We next discuss properties the **generator()** parameters must describe, and how those properties might be described, to generate a desired DHT workload.

A DHT benchmark is fundamentally about data movement. Therefore the first important characteristic of the benchmark's workload is the source of each data item. In a real application, some data items will already be stored in the instance of the DHT that resides on the source node, while others will be produced by the application running on the source ("client") node. In the first case the data item will be sent from the node that is responsible for the item's key, while in the second case the data may be sent from any node. Therefore in the first case the initial data distribution might specify node IP addresses directly, and in the second case it may be specified implicitly by the key value associated with each tuple.

In addition to source, each tuple's destination must be specified. This is determined by the *fkey* attribute of each tuple. The **generator()** function must therefore create the appropriate values for this column, reflecting the workload that is to be generated.

The last important attribute of each tuple is its size. The **generator()** function must create appropriate data sizes for the tuples reflective of the desired workload.

In addition to tuple attributes, the **generator()** function must describe workload operation rate, which can change over time. The rate at which each node's instance of PIER scans its local copy of the *G* table determines the rate at which it will call the DHT **get()** and **put()** functions. In the case of a virtual *G* table, this rate is controlled by the rate at which *G* tuples are generated, specified as parameters to **generator()**.

Tuple size, source, and destination might be specified using a statistical distribution, such as uniform, normal, or Zipf, or using a predefined mapping or histogram of the desired values of the attribute. Alternatively, these properties can be determined in a "trace-based" manner, based on a log of **get()** and **put()** operations collected from a real application. In this case the **generator()** function could read from a pre-existing trace file, thereby generating on-the-fly a *G* table that describes the traced workload. The operation rate can be described by a temporal rate distribution and its parameters. In the case of a trace file, we can timestamp the tuples with an elapsed trace time at the time of collection, and instruct our **generator()** function to produce a tuple only when the appropriate amount of time has elapsed.

5.4. Put vs. get

Up to this point, we have described how to use a DHT-based query processor to benchmark a DHT's **put()** function. Because a DHT's **put()** and **get()** calls both initiate content-based message routing, this should be sufficient for benchmarking a DHT's underlying message routing layer. However, it may be insufficient for benchmarking a DHT implementation, since real DHT applications will also use the DHT **get()** interface. We therefore would like a way to incorporate **get()** into the DHT workload. This can be accomplished by telling the PIER optimizer to use a fetch-matches join rather than a put-matches join.

Imagine a query like the one used in Section 5.1:

```
INSERT INTO localresults
SELECT *
FROM driver, data
WHERE driver.fkey = data.hashID
```

We will use the *driver* table to specify the sequence of DHT operations, much as the *G* table was used in Section 5.1, but here the *data* table will contain the data that is to be fetched. As before, assume *fkey* is a foreign key from the driver table to the *data* table, while *hashID* is the storage key for the *data* table. Assume also that the *data* table has exactly one match for each tuple of *driver.*

PIER will, on each node, perform a fetch-matches join, scanning the *driver* table on that node and requesting the corresponding data from the *data* table by calling **get(driver.fkey**). In response, each *data* tuple with *data.hashID* value matching *driver.fkey* will be shipped to the requesting node, where it will be joined with the tuple's columns in the *driver* table. By controlling the contents of *driver.fkey* on each node, we control the order and keys of **get()** requests, and by controlling the data in the *data* table, we control the content that is routed.

5.5. Multiple messages and realistic workloads

Although it is useful to create a workload of purely **put()**s or a workload of purely **get()**s, a real application will drive a DHT with a mixture of **put()**s and **get()**s. One way to generate such a workload is to issue a series of very small **put()**-type and **get()**-type SQL queries interleaved as desired. A less heavy-weight approach is to issue one large **put()**-type query and one large **get()**-type query combined using the SQL UNION operator but sharing a scan of a table *G*; PIER can then control the timing of the generation of *G* tuples to interleave processing of the incrementally generated *G* data, and hence the sequence of **put()** and g**et()** operations produced.

An example of an application that might interleave **get()**s and **put()**s is a network monitoring application that periodically collects system health parameters from the nodes in the system and then publishes various aggregates based on the results. Each node would periodically publish its statistics as a tuple in a DHT table. The monitoring application would periodically scan this distributed table, computing aggregate metrics of interest, thus generating **get()** operations. Assume a small number of nodes have "subscribed" to each receive different types of statistical aggregates. The monitoring application might then inform each subscriber of the desired information using **put()**s.

5.6. Perturbations

As we discussed in Section 3, we believe it is important that DHT benchmarks measure not just performance but also system behavior in the face of failures and evolution. We therefore need a mechanism by which new nodes join the network, existing nodes leave the network, and distributed faults are injected. We also need a way to describe when and how these activities take place--either through a statistical characterization or through a "perturbation trace."

One way to inject a "perturbation trace" along with the workload trace is to write user-defined SQL functions to instruct PIER to simulate perturbations, and to drive those functions with values in the artificially-generated data. For example, let us modify the query from Section 5.1 to read

```
INSERT INTO results
SELECT *, diskhang(G.hanglength)
FROM G, D
WHERE G.fkey = D.hashID
```

We can then fill in the *hanglength* column of a *G* tuple with a nonzero time quantity to specify the duration of a simulated disk hang that is injected into the system before this query is processed. This approach can be generalized by using one column of the *G* table to specify the type of fault and additional columns to specify fault characteristics (*e.g.*, length of time, whether to insert the fault before or after the corresponding tuple is processed, *etc.*).

A new node can be brought online, or an existing node made to depart the network, using a user-defined SQL function that calls an external benchmark supervisor module that starts and stops nodes or processes.

6. Conclusion

We have argued that a distributed query engine provides a natural framework for describing, executing, tracing and analyzing a DHT. We are beginning to implement this methodology to benchmark existing DHT implementations. In the future we hope to extend this work to develop a useful DHT benchmark that captures challenging and relevant workloads and faultloads for a new generation of distributed applications. We also plan to leverage the distributed tracing functionality to facilitate online debugging and monitoring.

References

[1] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proceedings of *ACM SIGCOMM,* 2001.

[3] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware),* Heidelberg, Germany, 2001.

[4] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *ACM SIGCOMM 2001*, 2001.

[5] B. Y. Zhao, J.Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. U. C. Berkeley Technical Report UCB/CSD-01-1141, 2001.