Decentralized systems need decentralized benchmarks

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

Abstract

Decentralized systems require new benchmarks and new benchmarking techniques. We propose a general methodology for benchmarking the performability of one class of decentralized system: peer-to-peer applications built on top of distributed hash tables (DHTs). Furthermore, we argue that benchmarks for decentralized systems must be designed and implemented with similar concern for scalability and robustness as the systems they are designed to benchmark, implying a need for decentralized load generation, fault injection, and metric collection. These criteria lead us to propose a benchmark implementation that uses a DHT to publish the faultload description and to store collected metrics, and uses a DHT-based relational query engine to analyze benchmark results. Finally, we argue that the fault injection and monitoring mechanisms required to run such benchmarks are reusable for online robustness testing, problem detection, and problem diagnosis, and that they therefore should be provided as infrastructure services.

1. Introduction

Large-scale decentralized systems such as peer-to-peer (P2P) and sensor networks have received increasing attention from researchers and industry over the past few years. For example, researchers have recently proposed decentralized Distributed Hash Tables (DHTs) as the fundamental building block for a new generation of globally distributed applications. Although much attention has been focused on the design of DHTs, techniques for evaluating key properties such as their performance and robustness to evolution and failures have not kept pace. As a result, these systems are commonly evaluated using simulation or implementations with artificial and simplified workloads and faultloads[1]. As DHTs make the transition from theoretical endeavors to standard application building blocks, the importance of characterizing their true performance, scalability, dependability, and performability (combined performance and reliability) increases. The devil is often in the details, and evaluation using standardized workloads and measurement criteria is essential to uncovering those details.

We believe the community would benefit greatly from rich DHT benchmarks, both for designers to evaluate tradeoffs and calibrate models, and for users to compare systems. Because robustness is a key design goal of these systems, benchmarks for them must be *performability* benchmarks, taking both performance and robustness into account. In this paper we make a preliminary proposal for such benchmarks, addressing the key issues of benchmark workload, faultload, and metrics. Further, we argue that benchmarks for decentralized systems must be designed and implemented with similar scalability and robustness criteria as the systems they are designed to benchmark, implying a need for decentralized workload generation, fault injection, and metric collection. Finally, we describe how the fault injection and monitoring mechanisms required to run such benchmarks are reusable for online robustness testing, problem detection, and problem diagnosis, and we argue that they thus should be provided as infrastructure services.

In this paper we target systems of tens of thousands of nodes deployed throughout the world. Examples of such services include P2P filesharing, document searching within worldwide corporate intranets, and mining autonomous local government agency databases. Our goal is to benchmark globally-distributed decentralized systems *in situ* rather than using simulation or cluster-based emulation. This approach provides a level of realism that simulation does not, and it allows us to benchmark systems larger than those that can be emulated on a cluster that a single organization has the resources to build and operate. The tradeoff is the loss of some degree of repeatability. We assume that such a globally-distributed system will experience node and network failures and other perturbations as the benchmark runs, but that these events will not happen often enough and with enough variety to cover the full range of perturbation scenarios of interest to a benchmarker. Thus we need not only to generate a workload, but also to generate perturbations above and beyond those that the system will naturally encounter, and to account for perturbations that occur outside the benchmarker's control.

2. DHT and benchmark model

A DHT is a decentralized data structure that assigns partitions of *keys* to *nodes*. We assume that P2P applications built on top of DHTs have three logical layers. The distributed object location and routing *(DOLR) layer*, which runs on top of the Internet, maps a key to its owner (the node responsible for the key) and routes a message to the node responsible for a key. Built on top of the DOLR layer is the *DHT storage layer*, which adds

[1] Though their true definitions differ slightly, in this paper we use *fault, error, and failure* interchangeably to denote improper behavior of a component.

persistent storage. This layer supports two operations: store a <*key,msg*> pair on the node responsible for the *key*, and retrieve the *msg* associated with a specified *key*. Finally, applications are built on top of the DHT storage layer, though they may use the DOLR layer directly for some operations. We assume the DHT storage layer supports, or can be modified to support, a scalable data replication mechanism that may admit temporary inconsistency. In the remainder of this paper, *DHT* refers to the DHT storage layer running on top of the DOLR layer.

Our benchmarking methodology is to measure various system metrics over time as a workload is applied to some layer of the system. We distinguish between *external* or *Quality of Service* metrics, which are directly visible to the user/caller of a component, and *internal* metrics, which are not. At various points in the benchmark run, we insert one or more *perturbations* and observe how the metrics change in response. *Perturbations* are a generalization of the concept of a fault, including not only faulty component behavior but also normal events such as system evolution as nodes join and depart. Thus a DHT benchmark is characterized by four attributes: system configuration, distributed workload, distributed perturbation load, and benchmark metrics. We address benchmark definition issues in this section, and the corresponding implementation issues in Section 3.

We distinguish between *benchmarking* nodes, which generate load, collect QoS metrics, and specify the perturbation load; and *benchmarked* nodes, which run the benchmarked application, collect internal metrics, and inject and experience injected faults. This is merely a logical division of tasks; a peer-to-peer system node is generally both a "client" and a "server," and is therefore both a "benchmarking" and a "benchmarked" node.

2.1. DHT workload and configuration

A *DHT workload* is the stream of requests inserted into the system, be it at the application layer (*e.g.*, IMAP message operations), DHT storage layer (*e.g.*, *get()* and *put()*), or DOLR layer (*e.g.*, who_is_owner(), send_to_owner()). In contrast to cluster systems, DHT behavior is heavily sensitive to the topology, performance, and reliability of network links among benchmarked nodes. We therefore suggest that much as the TPC benchmarks specify the contents of a pre-existing database on which the workload queries are run, DHT benchmarks should specify the topology and link characteristics of the network used to run the benchmark (in addition to the nature of any pre-existing data)[1].

[1] A"best effort" should be made to use the subset of nodes that most closely matches the requested configuration; the exact configuration used should be reported as part of the benchmark results. How, and whether, to add a layer of emulation to make the geographically-distributed benchmarking network more closely resemble the requested configuration is an open question.

Microbenchmarks exercise a single operation at a particular layer of the system, while *macrobenchmarks* attempt to exercise the entire API of a layer of the system in a manner that reflects the behavior of a typical application. Almost all DHT evaluations to date have been microbenchmarks driven by a random workload., so we focus on macrobenchmarks here. There are two ways to generate the macrobenchmark input workload to a layer of the system: by playing a trace (gathered from a real system or generated synthetically), or using a stochastic model designed to reflect typical application behavior. One approach for defining a macrobenchmark workload is to adapt traces from existing geographically distributed applications that use standard protocols, using information about where in the network the requests came from. For example, HTTP request traces can drive DHT-based content delivery networks, IMAP traces can drive DHT email systems, AFS traces can drive DHT filesystems, and SNMP traces can drive DHT monitoring systems. How to adapt traces from local-area systems is an open question.

2.2. DHT perturbation load

Due to space constraints, we do not fully explore the full range of injectable failures in this paper. Among the fail-stop failures that are relatively straightforward to inject, and which represent the manifestation of a wide variety of errors, are application or DHT runtime crash, hang, or exception; node crash or hang; network interface hang; and network switch or router crash or hang. Among non-fail-stop failures are switch or router overload leading to packet loss. Some failures may have fail-stop or non-fail-stop manifestations, *e.g.*, protocol implementation bugs, routing table corruption (in the DOLR layer or a router), and misconfigurations. Among the evolutionary perturbations of interest are voluntary node join and departure. Incorporating the role of operators in causing and repairing problems (*e.g.*, through modeling or using real humans) is an open question.

Having defined the set of perturbations of interest, we must inject them. Akin to the division of benchmark workloads into micro- and macro-benchmarks, we distinguish micro- and macro- perturbation loads, either of which can be inserted in the midst of a benchmark workload [2]. A micro-load consists of a single event. A macro-load consists of a series of events whose nature, timing, and correlations are played from a trace or are chosen stochastically using a model based on real-world data. The condition(s) for triggering perturbations may depend on system state (*e.g.*, aggregate system load) or events (*e.g.*, reaching a particular point in a program or a certain time), which may in turn require coordination among nodes to decide when to inject the perturbation.

2.3. DHT metrics

Application-layer QoS metrics include latency,

throughput, precision, and recall. Internal application metrics (some of which can be viewed as QoS metrics for lower layers of the system) include time to find an object's owner, time to retrieve an object, time to route a message to an object's owner, degree of load balance, degree of replication, and per-request consumption of CPU, memory, and network resources. Because we observe these metrics while injecting perturbations, we are interested in recovery time of the metrics after a perturbation, and the degradation in the metric between injection of the perturbation and system recovery, as much as we are in their values during perturbation-free operation. Direct measures of the impact of perturbations include number of users affected, amount of data affected, and time to detect, diagnose, and repair problems (be it automatically or by a human). Open questions include how to define an aggregate performability metric for an entire benchmark run, and how to characterize a system by measuring performability metrics across multiple benchmark runs that use different workloads and perturbation loads.

Metrics may be expressed in absolute terms or relative to a more traditional implementation. For example, a popular relative metric for expressing the cost of routing through an overlay network is Relative Delay Penalty (# of physical network hops taken when routing through the overlay / # of physical network hops in the shortest IP route to the destination).

3. Benchmark implementation criteria

We now turn our attention to implementing the benchmarking methodology that we have described. Decentralized systems are more difficult to benchmark than centralized systems for three reasons: number of nodes, topological distribution of nodes, and continuous component failure and recovery. These factors place certain requirements on the techniques we use for load generation, perturbation injection, and metric collection.

The *large number of benchmarked nodes in the system* requires scalable load generation (using multiple benchmarking nodes), scalable metric collection (from benchmarked and benchmarking nodes), and scalable coordination of perturbation injection. This demands autonomous operation whenever possible, scalable coordination when required, and using only a representative, perhaps time-varying, subset of nodes as benchmarking and benchmarked nodes.

Because the behavior of decentralized systems is highly dependent on the network topology, placement of nodes in the network relative to that topology, and link performance and reliability characteristics, nodes must be *distributed topologically* to capture the various "views" of the system from different connection points.

Finally, the large number of nodes and the use of a wide-area network mean that *the system is constantly in flux*--some fraction of components will always be down, degraded, or recovering. Our benchmark control and monitoring infrastructure therefore must be robust to failures, just like the systems it is designed to benchmark. This suggests that we use redundancy in load generation, and redundancy and loose consistency in disseminating control and monitoring data. We are willing to allow the benchmarking system to produce an *approximate* set of metrics if some components have experienced uncontrolled failures during the benchmark run, ideally with an indication of the statistical confidence in the result. Likewise, we do not require that all desired perturbations are actually injected, but we would like to maximize the fraction that are injected and about which we are notified. Finally, our data collection must be robust to interference--we must "subtract" or factor out perturbations to the system that we do not control, as well as the overhead of our benchmarking apparatus. To accomplish this first task, we must constantly monitor health and performance of all nodes and network links.

In the remainder of this section we propose initial ideas on mechanisms for decentralized perturbation injection and measurement that aim to achieve the above criteria.

3.1. Decentralized perturbation injection

To add fault injection to a component, a component is instrumented to behave in a faulty way when it is told to do so, its execution environment is instrumented to be perturbed or destroyed, and/or a proxy is interposed in a dataflow path to intercept data and change it to make it faulty. The primary challenges for perturbation injection in decentralized systems are communicating the perturbation load specification to benchmarked nodes, obtaining acknowledgement of when the perturbation injection took place relative to the requested moment, and implementing distributed injection triggers.

The description of *when* to trigger a perturbation must be communicated from a benchmarking node to the relevant benchmarked nodes. We advocate publishing this information in a (separate) DHT, thereby leveraging the DHT's scalability and robustness. Communicating the perturbation load in this way can happen before or during a benchmark run. As an aside, if we are willing to modify the application, DHT storage layer, and/or DOLR layer, we can piggyback a fault injection command on a request. For example, "misroute this message" can be attached to a DOLR-level message, or "corrupt this message" can be attached to a storage-level message. This in-band approach allows us to tie a fault injection command directly to a request, and it ensures that fault injection actions may only be taken on nodes involved in processing the request.

We advocate communicating from benchmarked to benchmarking nodes the acknowledgment that a perturbation took place, and when it took place, by storing this data in a DHT, in a format that can be queried by a DHT-

based P2P query engine. By "joining" the acknowledgement table with the metric tables of interest, the benchmarker can easily correlate variations in metrics with perturbation events. Note that a possible alternative to storing perturbation description and event data in a DHT is to use peer-to-peer publish-subscribe.

How best to determine when a (possibly distributed) perturbation injection trigger's conditions have been met (or met "closely enough," if we allow approximate triggering conditions) is an open question. One approach is to extend the ideas in [4] to work for very large-scale systems in flux.

Finally, benchmark usability would be enhanced by developing a description language to specify perturbations and injection trigger criteria.

3.2. Decentralized collection of metrics

We next discuss the issue of collecting and analyzing benchmark results. We advocate storing in a DHT, in a format that can be queried by a DHT-based P2P query engine, the per-request metrics collected by the components involved in issuing and processing a request. The query engine can then be used to aggregate and analyze results. This approach offers robustness and scalability by using the DHT, and flexibility and extensibility by using declarative queries to analyze benchmark results.

Collected metrics may be inserted into the DHT offline or online. If done *offline*, every benchmarked and benchmarking node records benchmark measurements locally, and then at the end of the benchmark run bulk-loads them into the DHT, which is then queried by the benchmark user. In the *online* version, metrics are inserted into the DHT during the benchmark run. The online approach perturbs the system more than the offline approach because insertions into the DHT may cause network traffic, and this must be factored out of the final benchmark results. But the online approach allows a benchmark user to obtain incremental results as the benchmark runs, and the results are less dependent on which nodes and links are alive at the end of the run.

We believe that SQL queries issued to a distributed query engine are a natural way to flexibly analyze and summarize benchmark results. For example, assume that all benchmarking nodes record in a table *KS* the time when they send each application-level request and in a table *KR* the time they receive each response, tagged with a unique *id* that allows them to correlate requests and responses. Then the query

SELECT avg(KR.time-KS.time) FROM KR, KS
WHERE KR.id = KS.id

produces the average latency for all requests.

More generally, the mechanisms we need for collecting and analyzing benchmark results are also those needed for general system monitoring. If we use online data collection, the difference between analyzing benchmark results, and online problem detection and diagnosis, is simply a matter of what queries we issue.

For problem *detection*, we suggest continuously-running SQL queries (CQ) that generate an alert when predefined anomalous conditions are met. For example, we may wish to know when the average application response time during the last minute is more than 110% of the average response time during the last ten minutes. This is accomplished by building on our previous query:

SELECT "alert" AS result WHERE
(SELECT avg(KR.time-KS.time)
FROM KR[Range 1 Minute], KS[Range 1 Minute]
WHERE KR.id=KS.id) > 1.1 *
(SELECT avg(KR.time-KS.time)
FROM KR[Range 10 Minute], KS[Range 10 Minute]
WHERE KR.id=KS.id)

Once a problem has been detected, an operator *diagnoses* it by issuing *ad hoc* queries to test her hypotheses about possible problem causes. Suppose she decides to investigate the request that experienced the highest latency during the past minute. To find its id, she issues

SELECT KR.id as iid
FROM KR[Range 1 Minute], KS[Range 1 Minute]
WHERE KR.id=KS.id AND KR.time-KS.time = (
SELECT max(KR.time-KS.time)
FROM KR[Range 1 Minute], KS[Range 1 Minute]
WHERE KR.id = KS.id)

Assume that the DOLR layer on each node keeps *IS* and *IR* tables describing IP network level message sends and receives (just as the benchmarking nodes did for application-level sends and receives), and further that the application-level unique identifier is kept with the message as it travels through the network. Assume the *IS* and *IR* tables have the following columns: *time, id,* and *me* (the node recording a fact); and that the *IS* table also has a *next* column indicating where the routing layer intends to send a message next. Now the operator can see how long it took the message from the previous example to get from hop to hop in the overlay with

SELECT IR.time-IS.time, IS.me, IR.me FROM IR,IS
WHERE IR.id=*iid* AND IS.id=*iid* AND IS.next=IR.me

Imagine she finds the latency from node A to node B (*IR.time-IS.time* for *IS.me=A* and *IR.me=B*) is unusually high for that message. She can determine whether this is the case for just that message, or for many recent messages from A to B, by comparing the A-to-B latency from the above query with the result of

SELECT avg(IR.time - IS.time)
FROM IR[Range 1 Minute], IS[Range 1 Minute]
WHERE IS.me='A' AND IR.me='B' AND IS.id=IR.id

If the average latency is high, she has identified an IP network problem. Other queries can investigate alternative hypotheses such as the high application-level latency being due to high CPU load on one or more of the nodes through which the request passed. More generally, queries over the data we collect can be used to find overloaded or misbehaving components. Note that **id** allows us to correlate high- and low-level metrics. Also, the query engine makes it easy to examine system configuration and health from before, during, and after a problem manifests, and to compare to past problems.

Open questions include how to minimize monitoring overhead; generate statistically meaningful results using inconsistent, missing, or stale data, *e.g.,* without using data from all nodes or all requests (for performance reasons or due to node failures or partitions); improve CQ efficiency by pushing predicates and aggregation operations into the network; scope queries, and aggregate query results, based on network topology or administrative domain (*e.g.*, to receive fine-grained data about nearby nodes and coarse-grained or aggregated data about distant nodes); use aggregation to archive historical data; integrate last-N-events queries with time-based queries; choose the appropriate design point between centralized and decentralized monitoring; and replace hard-coded CQ triggers with an adaptive anomaly detection mechanism layered on top of the query engine.

4. Related work

The system we have proposed draws on related work in benchmarking, fault injection, and system monitoring. Benchmarking performability by measuring QoS as faults are injected is not new [2] [6], and work on performance benchmarking of DHTs has begun [8]. In this paper we describe a general framework for benchmarking DHT-based applications, raising issues of decentralized fault injection and data collection not addressed in previous benchmarking work. We also describe workloads, perturbation loads, and metrics in more detail than previous DHT-related work.

Fault injection has a rich history. The projects most relevant to ours are NFTAPE [9] and Loki [4], two systems for injecting faults in local-area distributed systems. In moving fault injection to the wide-area, we must tackle additional issues of scale and perturbations outside the benchmarker's control. We also propose to integrate our fault injection methodology with a general benchmarking system rather than using it for testing.

Researchers have built distributed query engines for P2P systems and have suggested their usefulness for network monitoring [5] [7]. We do not propose innovation over these systems, but rather point out their usefulness in recording and analyzing benchmark metrics and in drilling down to diagnose problems. Tagging requests for automated problem diagnosis in a single-node system was implemented in [3].

Continuous queries are a new, active area of research in the database community. The continuous query syntax we present here is loosely based on [1]. We believe distributed monitoring is a natural application of this technology, as it allows declarative specification of actionable conditions without needing to run general-purpose code outside the query engine. CQ are not unlike publish/subscribe systems, which may be an appropriate mechanism for broadcasting CQ results.

5. Conclusion and status

We have outlined a general methodology for benchmarking large-scale decentralized systems, and have explored preliminary design challenges that stem from the scale and frequency of failures in such systems. Central to our proposal is using a separate DHT to distribute the workload and perturbation load specification, and to collect metrics. Two components our proposed system--decentralized perturbation injection and decentralized metric collection--are reusable (the former for testing fault detection and recovery mechanisms in deployed systems, the latter for online monitoring and problem diagnosis), arguing for their provision as a generic platform service. Indeed, the fault injection system can be used to test the monitoring system, and the benchmarking system can be used to benchmark both. Of course a generic service raises issues not addressed here of isolation, fairness, usability, and standardizing an API.

We have left as open questions many other important issues, including the design of a declarative benchmarking/monitoring language, or GUI, on top of SQL that is integrated with the workload/perturbation load specification; how to intelligently explore the "perturbation space" of applications; and how to accurately correct for the overhead of data collection and perturbations from uncontrolled failures. We are beginning to implement a framework based on the ideas presented here, using the PIER distributed query engine [5] to collect benchmark metrics and monitoring data. We are also beginning to run *ad hoc* benchmarks to gain experience with workloads, perturbation loads, and metrics. We invite collaboration with other researchers in exploring the ideas presented in this paper, and in evaluating their systems.

References

[1] A. Arasu, *et al*. An abstract semantics and concrete language for continuous queries over streams and relations. http://dbpubs.stanford.edu/pub/2002-57

[2] A. Brown, *et al*. Towards availability benchmarks: A case study of software RAID systems. *Proc. 2000 USENIX Annual Technical Conference*, 2000.

[3] M. Chen, *et al.* Pinpoint: Problem determination in large, dynamic systems. *2002 Intl. Conference on Dependable Systems and Networks*, 2002.

[4] M. Cukier, *et al.* Fault injection based on a partial view of the global state of a distributed system. *18th IEEE Symposium on Reliable Dist. Sys.*, 1999.

[5] M. Harren, *et al.* Complex queries in DHT-based peer-to-peer systems. *IPTPS '02*, 2002.

[6] K. Nagaraja, *et al*. Using fault injection and modeling to evaluate the performability of cluster-based services. To appear in *4th USENIX Symposium on Internet 'Technologies and Systems*, 2003.

[7] R. van Renesse, *et al*. Scalable management and data mining using Astrolabe. *IPTPS '02*, 2002.

[8] S. Rhea, *et al*. CANs need application-driven benchmarks. To appear in *IPTPS '03*, 2003.

[9] D.T. Stott, *et al.* NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. *4th IEEE Intl. Computer Performance. and Dependability Symp.*, 2000.