♦ Mitigating High Latency Outliers for Cloud-Based Telecommunication Services

Fangzhe Chang, Peter S. Fales, Moritz Steiner, Ramesh Viswanathan, Thomas J. Williams, and Thomas L. Wood

Telecommunication applications are distinguished by their stringent requirements for availability and completion times. A highly available, low-latency, distributed data store is therefore a critical component of cloud-based realizations of telecommunication services. We present a systematic experimental evaluation of state-of-the-art database systems as components of telecommunication applications. We show that while their average latencies are well within the required time scales, the distribution of latencies exhibits a long tail of unacceptably large outliers which may significantly impair meeting the performance requirements of telecommunication applications. To address the observed phenomenon of high latency outliers, we present a new solution that is implemented in a Bell Labs system code named Flurry. Flurry is based on using the first response from a replica rather than waiting for all or a quorum of responses from replicas. To handle incorrect responses arising from message losses, Flurry uses a novel checking algorithm based on vector clocks to determine the correctness of a replica's response. We present experimental evaluation results which show that Flurry significantly reduces both the average response time and the probability of unacceptable response times to values that would allow meeting the availability and completion time thresholds required for telecommunication services. © 2012 Alcatel-Lucent.

Introduction

With the introduction of large-scale data centers and cloud platforms, telecommunication applications are expected to move from being housed on specialized physical equipment to being virtually hosted in the cloud. Initial evidence of this trend can be found in [10] and [13]. The differentiating benefit offered by such cloud-based solutions is the elasticity in utilized resources. Specifically, if there is greater than anticipated demand for a service, additional compute and networking resources can be dynamically leased. A service provider, therefore, eliminates the risk of under-provisioning an offered service and its potentially serious consequences for both company finances and brand image. Similarly, in a case where demand for a service is lower than expected, resources can be released, thereby reducing costs.

As a concrete representative example of a telecommunication application, we consider the mobility

Bell Labs Technical Journal 17(2), 121–142 (2012) © 2012 Alcatel-Lucent. Published by Wiley Periodicals, Inc.
Published online in Wiley Online Library (wileyonlinelibrary.com) • DOI: 10.1002/bltj.21548

Panel 1. Abbreviations, Acronyms, and Terms

API—Application programming interface	LRU—Least recently used
CDF—Cumulative distribution function	LTE—Long Term Evolution
CPU—Central processing unit	MME—Mobility management entity
DHT—Distributed hash tables	POSIX—Portable Operating System Interface
EC2—Elastic Compute Cloud	SQL—Structured Query Language
eNB—Evolved NodeB	TA—Tracking area
ETS—Erlang term storage	TCP—Transmission Control Protocol
GUTI—Globally unique temporary	UDP—User Datagram Protocol
ID—Identifier	UE—User equipment
IMSI—International mobile station identity	VM—Virtual machine

management entity (MME) which serves as the control plane in the Long Term Evolution (LTE) cellular backplane. The MME keeps track of the location (tracking area, or TA) and associated state of a cellular phone (user equipment, or UE) as it moves through the cellular network to complete and maintain network-initiated voice or data connections. Because of power management concerns, LTE UEs spend most of the time in low-power mode with their transceiver turned off. UEs listen at regular intervals to the beacons sent by the local base station (evolved NodeB (eNB)) and explicitly notify the MME of changes in their TA. The MME is charged with keeping all data related to a UE—called the UE context—current while the equipment is idle. The UE context includes values for different UE identifiers. These include the globally unique temporary UE identity (GUTI), international mobile station identity (IMSI), the state of the UE (e.g., idle, connected), security keys (used for authentication and authorization before a UE is connected), subscription data, and its TA. When a call is made to the UE, the MME performs paging by contacting all eNBs in the last known TA in which the UE was detected before widening the scope of the search. Thus, the MME also needs to maintain the association of TAs with eNBs. Requirements for the MME dictate that it be available 99.999 percent of the time and that within 500 ms, networking and interface failures should be detected and traffic re-routed without losing conversations. Consequently, the UE context data and the association of TAs with eNBs must be accessible with low response time and resilient to failure.

More generally, we can observe that telecommunication applications have the following distinguishing characteristics. First, they have stringent requirements on their availability and processing completion times. Second, although the computations performed on data are relatively simple, they are nevertheless data-intensive in that a significant fraction of the message processing logic is tied to querying and updating session data and state. Together, these imply that a critical component common to cloud deployments of most telecommunication applications is a highly available and low-latency distributed data store. The recent advent of NoSQL databases promise excellent response time and scaling characteristics. The first contribution of this paper is a systematic experimental evaluation of existing NoSQL systems as components of telecommunication applications. We study the variation of throughput and latency with respect to several factors including read/write loads, degrees of replication, and the number of network nodes. The systems considered include Apache Cassandra [14], Riak* [3], and memcached [9] deployed both on physical machines and a public cloud of leased virtual machines (Amazon Elastic Compute Cloud (EC2*)). Our results show that the average throughput indeed scales very well with the number of network nodes, and that the average latencies are well within the time scales required for telecommunication applications. However, we also found, somewhat surprisingly, that the distribution of latencies exhibits a long tail of unacceptably large outliers which may significantly impair meeting the performance requirements for telecommunication applications.

Referring again to the example of MME, the requirement of 99.999 percent availability together with a time completion of 500 ms demands that the probability of completion times being more than 500 ms is guaranteed to be less than 0.00001, and our observed magnitude of outlier latencies and their frequency would thwart such a guarantee from being met.

Consequently, a second contribution of this paper is a new solution, a Bell Labs system code named Flurry, which we developed for fault-tolerant replication of state machines that can be applied to implement a reliable data store or, more directly, stateful replicated copies of the standalone telecommunication applications. The key underlying insight behind Flurry is that existing systems need to wait for responses from at least the quorum number of replicas and the overall response time is limited by the worst percentile of latencies among the set of replicas. Our proposed scheme is instead based on using only the first correct response, and the resulting response time is therefore more strongly correlated with the best latency to a replica. The main technical challenge is determining the correctness of a response in the presence of message losses. Flurry adds vector clocks [8, 15] to messages and identifies a checking condition based on vector clocks for addressing this issue. Finally, we present experimental evaluation results which show that Flurry significantly reduces both the average response time and the probability of unacceptable response times to values that would allow meeting the availability and completion time thresholds required of telecommunication services.

The rest of the paper is organized as follows. First, we present background on NoSQL databases and the specific systems that we chose to evaluate. Next, we present our experimental methodology and the evaluation results. We then present the design and implementation of the Flurry system and evaluation results of its performance. We conclude with a summary of our contributions and directions for further work.

Existing NoSQL Databases Studied

A significant number of application states and data in telecom applications can be stored in NoSQL databases which support high availability and scalability. Compared with traditional relational databases offering complex Structured Query Language (SQL) queries and transactions over tables, NoSQL databases are much simpler in that they store key-value pairs and access data only through keys, with or without strict concurrency control mechanisms. NoSQL databases are typically highly available and scalable since they are implemented to take advantage of a cluster of machines with data replicated on different machines. Since it is not always possible for a distributed system to be consistent (C), available (A), and partition-tolerant (P) at the same time (i.e., CAP theorem [12]), NoSQL databases tend to favor availability and partition tolerance over consistency in the presence of machine failure or network partitioning, and rely instead on application-assisted conflict resolution when conflicting data versions are detected.

A set of key-value pairs is often regarded as a hash table or dictionary. Correspondingly, such databases are also called distributed hash tables (DHT). Examples include Dynamo [6], Riak [3], Cassandra [14], memcached [9], and CouchDB [2]. These NoSQL databases are often built with their own perspectives. In this paper, we focus on aspects related to deliver high availability, high scalability, and fast responses, also known as low latency.

Dynamo [6], from Amazon, is a highly available data store that is not publicly available. Dynamo partitions data using consistent hashing onto a circular key space (i.e., ring) such that a node (i.e., machine) is assigned a segment of the ring. In addition, each data item is replicated on a list of nodes (called the preference list) for high availability, with a coordinator (typically the first node on the preference list) that manages read and write operations on all replicas using a sloppy quorum approach. In the standard quorum approach (*c*.f. [11]), when network partitions or nodes crash, the operation can fail or be blocked indefinitely. In this scenario, sloppy quorum diverges from the standard quorum by using the first set of healthy nodes (on the preference list) for the write operation. Dynamo uses hinted handoff to transfer the affected data back to the original nodes once they recover. If conflicts are detected (e.g., due to concurrent transfer-backs from multiple sections of the once-split network, or when two application processes try to update the same data item at the same time), the application will receive all versions of the data at the next read and will be responsible for performing data reconciliation. Conflict detection and reconciliation is based on data versioning. (A vector clock which consists of a list of node-counter pairs provides the version associated with every data item, indicating the number of updates on the node to the corresponding data item). Since data consistency eventually relies on assistance from the application, Dynamo calls it *eventual consistency*. Even though Dynamo supports high availability and scalability, it lacks explicit mechanisms to ensure small and predictive latency bounds. In fact, [6] has reported that data accesses can have 99.9 percentile latency as high as ~200 ms.

Riak [3] is an open source implementation of Dynamo [6], with extended functionality such as links and MapReduce [5]. MapReduce functions specified in JavaScript* or Erlang can spread the processing of a (possibly more advanced) query across many nodes to take advantage of parallel processing power, with the potential to shorten query latency. In addition, Riak allows different storage back ends, e.g., in-memory Erlang term storage (ETS) tables. In-memory back ends avoid disk access, thus making responses faster. Similar to Dynamo, conflicting data versions can occur in Riak, for instance, due to concurrent writes or writes from clients using a stale vector clock obtained from a long-past reading. Applications must select one of the siblings to replace the conflicting data versions.

Cassandra [14], initially developed by Facebook, is an implementation of both Dynamo [6] and Bigtable [4]. Bigtable focuses on storing a large amount of data across commodity servers. Similar to Bigtable, Cassandra structures a value into fields under multiple column families and stores fields from the same column family (spanning different keys) together. As a result, this enhances the query response time for a field of a fixed range of keys. Correspondingly, Cassandra supports order-preserving hash functions in addition to consistent hashing. It also provides several replication policies including "Rack Unaware," "Rack Aware" (within a datacenter), and "Datacenter Aware."

memcached [9] is a key-value store (also known as a hash table or dictionary) combined with cache replacement policy, hosted in the memory of a cluster of server machines. Each memcached server can be regarded as a bucket storing a collection of data. The client side library uses hash function mapping keys to bucket numbers to determine which machine to send requests to. When a bucket is full, subsequent insertions cause older data to be purged in least recently used (LRU) order. memcached uses multiversioning and is lockless and so that no client can block any other client's actions. Data items are not replicated on memcached. Requests for keys on a failed server simply result in a cache miss. Elasticache [1], Amazon's implementation of memcached, supports automatic failure detection and recovery, though it lacks a replication function.

Experimental Evaluation of Existing Database Systems

The following sections describe our experimental methodology and report our test results.

Measurement Methodology

In the following sections, we present our test client, the database systems tested, and the configuration parameters considered in our tests.

Test client. We developed a simple test client to measure the performance of the various database systems. The client was written in C++ (and C). This language was chosen for several reasons:

- We felt it would give us the best control over the low-level details of the system.
- Client libraries were available for all the target database systems.
- We wanted to run tests on a number of different hardware and software platforms, and this minimized the prerequisites that needed to be satisfied on those machines, e.g., no special libraries, or execution environments such as Java*, Erlang, or Eclipse would be required.

While the client itself is custom C and C++ code, we were generally able to take advantage of existing libraries to handle the details of the database application programming interface (API). The client consists of a common *front end* that handles argument parsing, setting up the client threads, executing the tests, calculating statistics, and printing the results; a *middleware* layer that translates generic calls such as "write a key-value" pair to the library API; and a *back* *end,* a library supplied by the database developers or a third-party contributor. The client application starts up a user-specified number of threads (each one intended to simulate a typical "real" client). Each thread sends a request to write a key-value pair into the database, and when that completes, it sends a request to read the value with the same key. It cycles through a user-specified number of distinct keys, and does this either as fast as possible, or throttled back to some lower rate as specified by the user. All these user-specified values and others are passed as command line parameters to the client application which allows it to be easily scripted and run simultaneously on multiple machines. Over the course of the test run, the client accumulates statistics related to the response time to each of the read and write requests (min, max, average, and standard deviation). It can also be instructed to save the response time data. This data can be used to calculate 99.9, 99.99, and 99.999 percentile response times (done automatically by the client), or further analyzed or used to create summary plots using other tools.

The load on the target system can be varied either by sending requests at a fixed rate, or by increasing the number of client threads. With just one thread, the system typically is very lightly loaded since the server spends most of its time waiting for requests and responses to pass through the network. However, with multiple threads, several requests can be run in parallel, and we can load the system down to the point that database throughput becomes the bottleneck.

Database systems. Several representative NoSQL database systems were selected, somewhat arbitrarily, for more in-depth testing. These systems are described below.

- *Apache Cassandra* is an example of a feature-rich, high-availability, high-scalability, system which stores data on disk, and goes to great lengths to obtain a reasonable performance while at the same time minimizing data loss through techniques such as replication, commit logs, hinted-handoff, bootstrapping of failed nodes, and others. It is implemented in Java, and most testing was done using version 0.6.8. The back end uses the Apache Thrift interface for which C++ is one of the supported targets (http://thrift.apache.org/).
- *Riak* is an example of a system which replicates data across nodes for reliability and scaling, but also keeps all data in memory. For the purposes of our tests, all data was kept in memory. We presumed this would improve latency and throughput, though perhaps at some cost to reliability. Riak is implemented in Erlang and we used version 0.14.1. The back end uses a C language client library developed by Piotr Nosek with some local enhancements (https://github.com/fenek/riak-c-driver).
- *memcached* is not a distributed NoSQL system, but we included it for comparison. Since it keeps all data in memory and does not attempt to replicate data across multiple machines, it should serve as an example of the best that can be done in terms of handling read and write operations before adding-in the overhead needed for reliability, scalability, and maintainability. It is written in C and we used version 1.4.5. For the back end, we used libmemcached version 0.44 (http://libmemcached.org/libMemcached.html).
- Other systems which we looked at, though in less detail, include Project Voldemort (http://project-voldemort.com/), and Redis (http://redis.io/).

Test parameters. There are a large number of parameters that need to be considered when running these tests. These include the architecture of the database system (number of nodes, replication factor, and disk-backed versus memory storage), the characteristics of the host systems (disk, memory, CPU cores, and processor speed), and the characteristics of the client (number of threads, size of values, number of unique keys, and read versus write mix). In addition to those, which are common across most of the databases, each NoSQL database typically has a large number of parameters that can be adjusted to tune performance for any particular workload, and for systems implemented in a virtual machine language such as Java or Erlang, there are parameters for tuning the virtual machine. In the data that follows we attempt to show some typical cases, but we don't claim that this is the absolute best performance that could be obtained for any particular database, hardware, and test load.

Typical parameters used for the majority of the tests described here include:

- Three servers.
- Replication factor set to three (one copy stored on each server).
- A test client (running on a different machine) running several client threads (typically 8 to 16) to simulate running multiple "real" client connections to the database. The number of threads was selected to provide a "reasonable" load on the database (not overload, but more than the very light load that is provided on a single-threaded client) with the precise number being fixed as follows. A series of tests was run starting with a very light load, and then increasing the number of threads. Typically the throughput would scale in a close-to-linear fashion up to the point where the system capacity was reached. At that point, the response times would start to increase significantly, and there would be little or no increase in throughput. The test configuration used for these measurements would be a level well below this overload threshold.
- 250,000 unique keys distributed across the servers, with data sizes of 1000 bytes.
- Where supported, "quorum" responses were used for reads and writes. This means that the server handling the request must get a response from a quorum of the distributed nodes before returning a response to the client. (In this case, when the replication factor is three, the quorum value is two.)

Test Results

One test goal was to compare the same tests running on virtual machines versus running on the "bare metal" of a physical server. Though there was little doubt that the performance of the physical machines would be better than running on virtual machines, the penalty for doing so was not as clear. In the following sections, we've included plots of response latency versus time, as well as the corresponding cumulative distribution (CDF). Typically the CDF is better at showing the distribution of the smaller typical response values, while the time sequence plot is better at showing the frequency and magnitude of the larger outliers.

Cassandra on physical machines. This test, illustrated in **Figure 1**, used three database nodes running on three high-end physical machines, and 16 client threads. Though the average response is around 2 milliseconds, the maximum time experienced by a very small number of requests is over 250 milliseconds. The reasons for these fairly large outliers are not well understood, but seem to be common to some degree across the various systems. Tuning can help to address the magnitude and frequency of these outliers, but it's difficult to eliminate them completely.

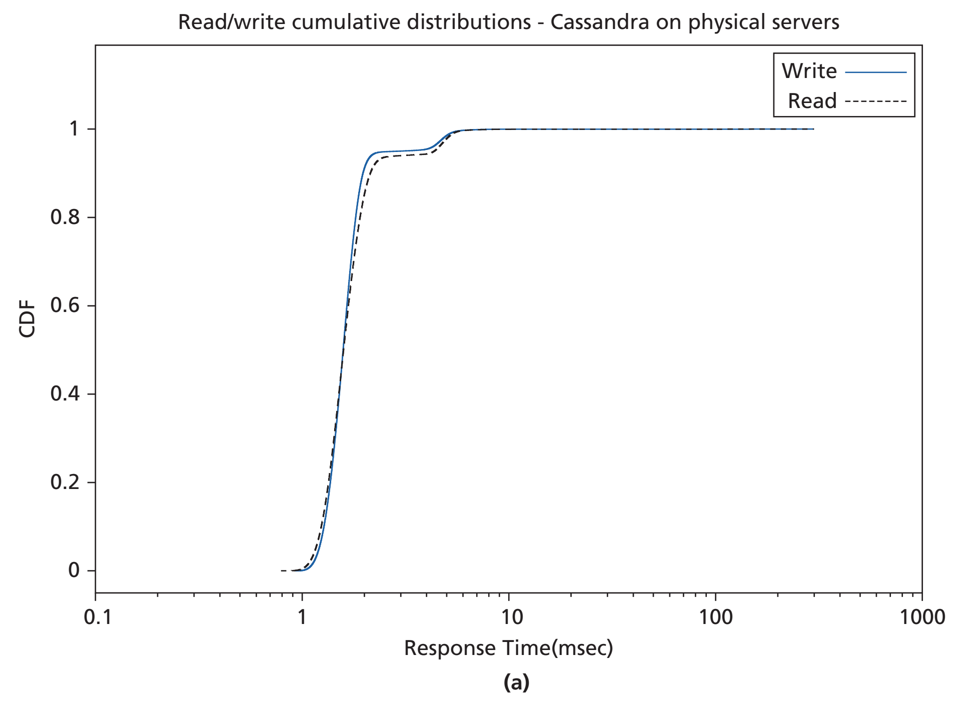
Cassandra on virtual machines. This test, illustrated in **Figure 2**, used three database nodes running on three Amazon EC2 m1.large virtual machines, and 8 client threads. Though the average response is around 2 to 3 milliseconds, the maximum is over 100 times larger, at around 600 milliseconds. There are a non-trivial number of responses in the 20 to 30 millisecond range—some of this is presumed to be due to hypervisor scheduling on the virtual machines.

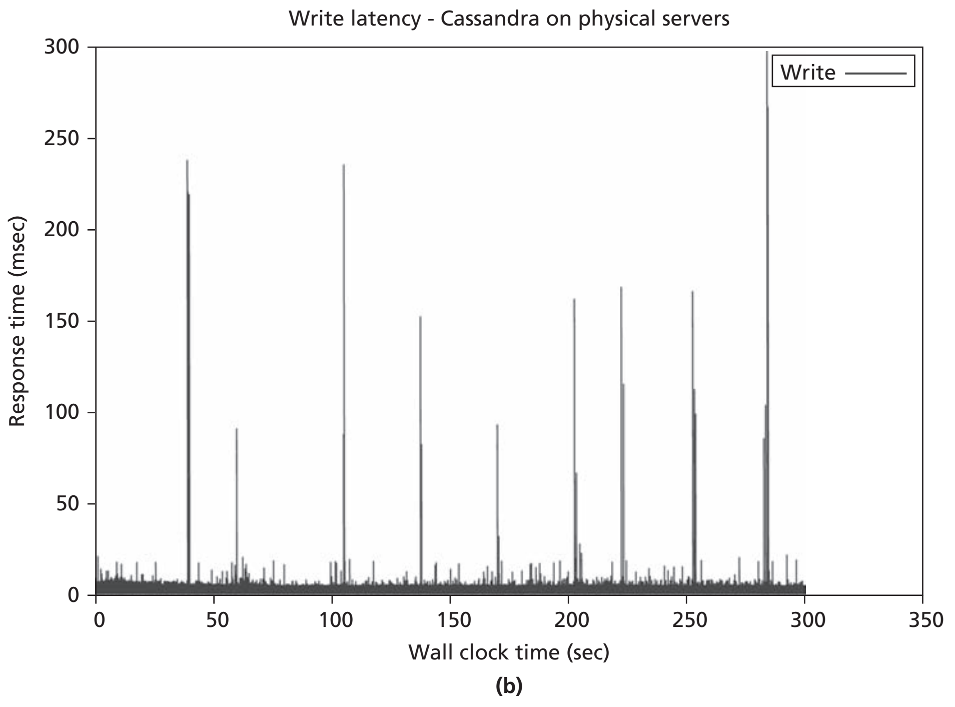
Riak on physical machines. This test, whose results are illustrated in **Figure 3**, used three database nodes, running on high-end (8 core) processors. While there are a very small number of responses in the tens of milliseconds, 99.9 percent are under 3 milliseconds, which is only about twice the average value of approximately 1.7 milliseconds.

Riak on virtual machines. This test, with results illustrated in **Figure 4**, was run using three Amazon EC2 m1.large machines as server nodes. The load is from a single client process running 8 client threads. Here, the average response was around 3 milliseconds and 99.9 percent of the requests complete in under 100 milliseconds, but there are a number that extend out to several hundred milliseconds and a few that take several seconds to complete.

Memcached on physical machines. This test, illustrated in **Figure 5**, was run using a high-end physical machine as the memcached server. The load is from a single client process running 32 client threads. Here again, we see a very small number of responses in the range of tens of milliseconds, compared to an average time of less than 1 millisecond.

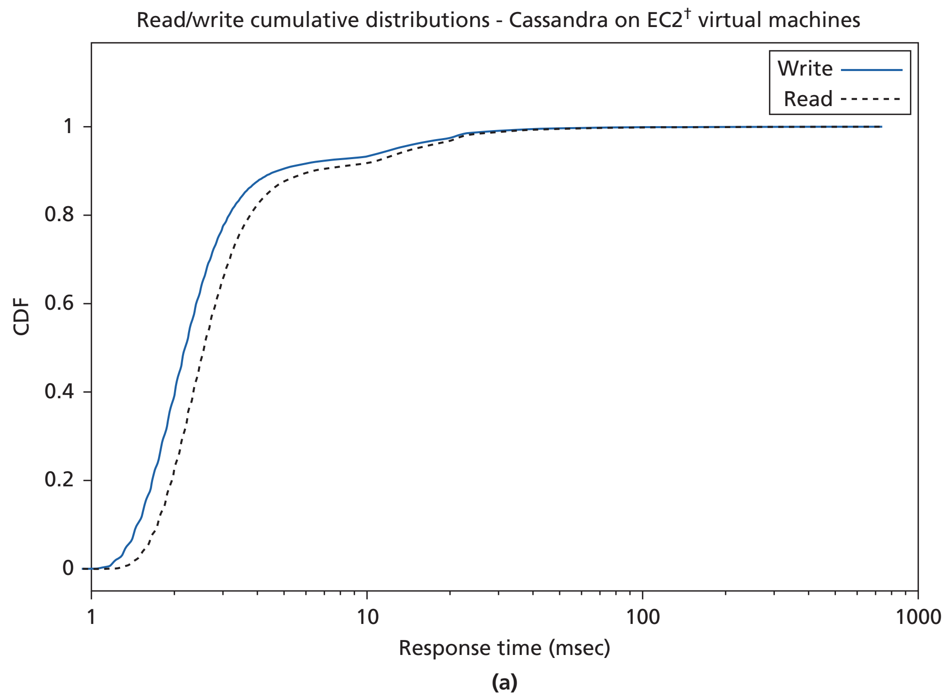
Memcached on virtual machines. This test, illustrated in **Figure 6**, was run using an Amazon EC2 m1.large machine as a server. The load is from a single client process running 16 client threads. The average response time is under 1 millisecond, but there are outliers of 200 milliseconds or more.

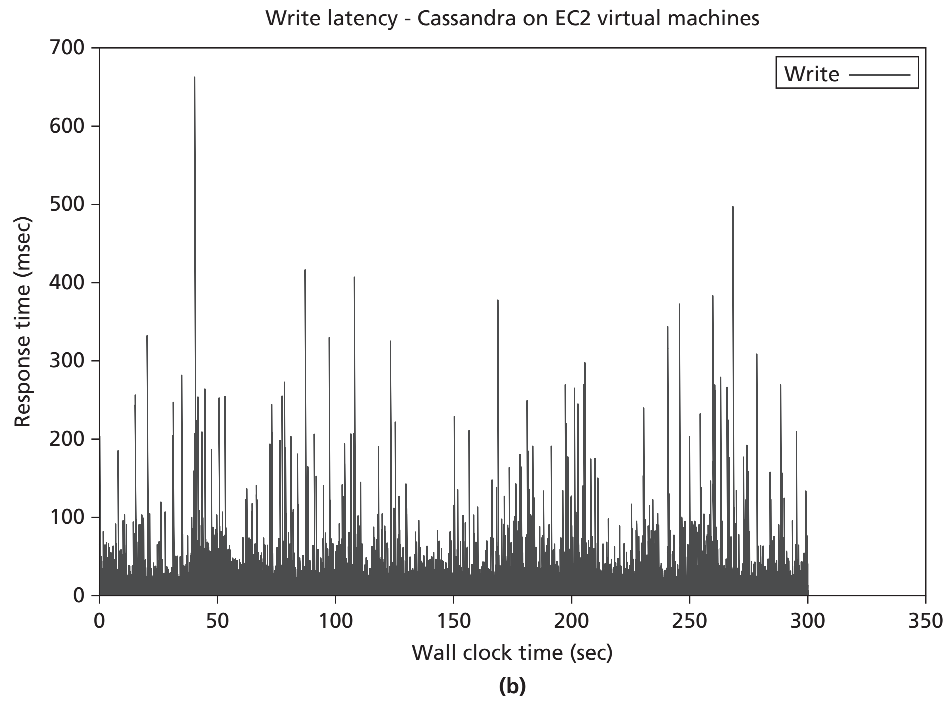




CDF—Cumulative distribution function

Figure 1.
Cassandra on physical machines.

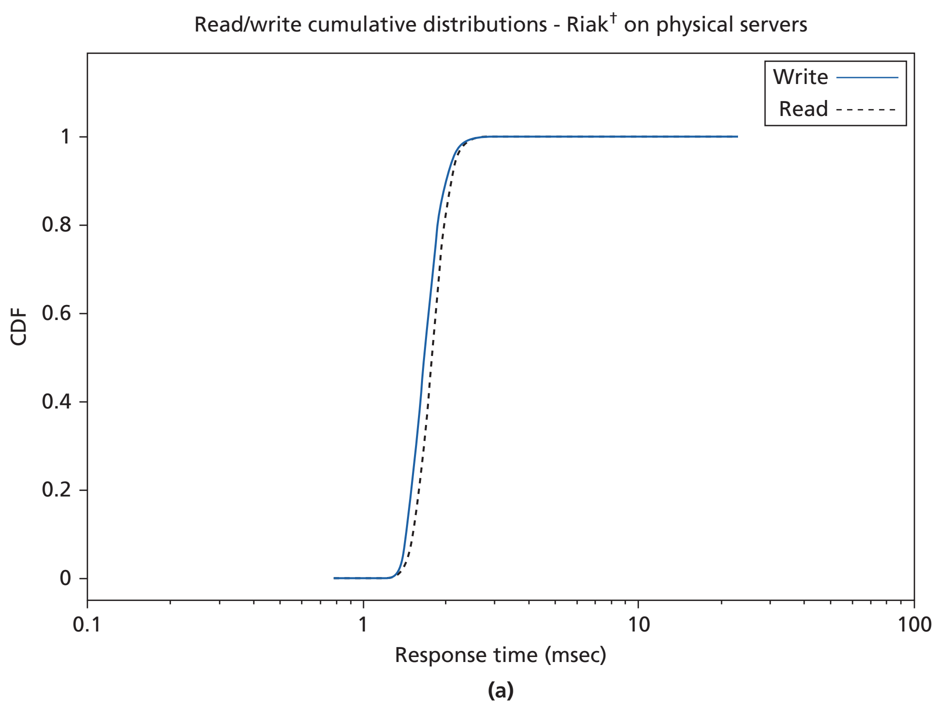


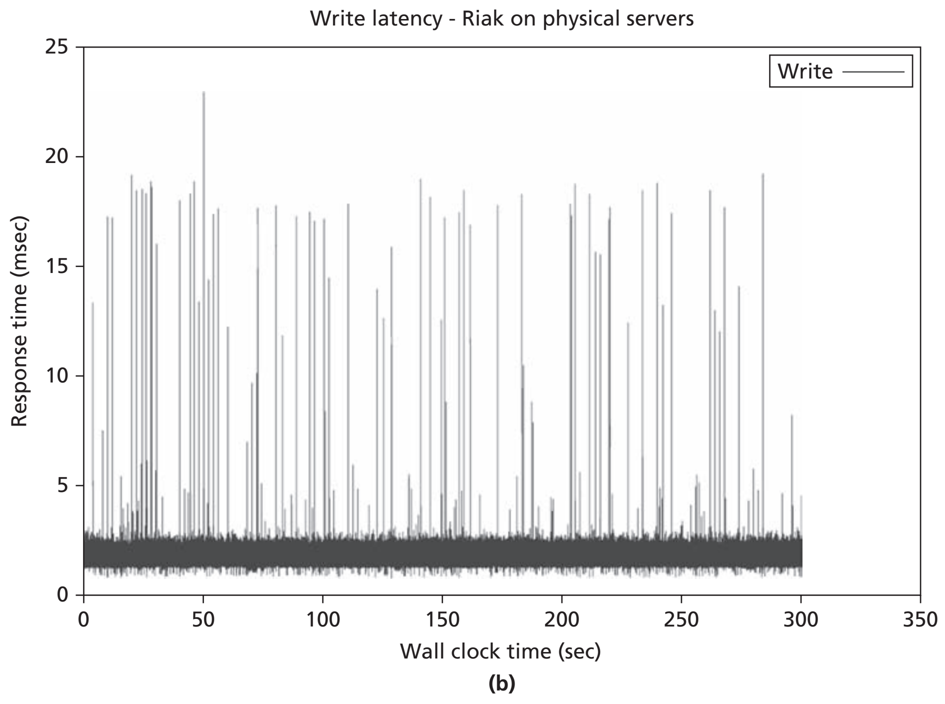


[†]Trademark of Amazon Technologies.

CDF—Cumulative distribution function
EC2—Elastic Compute Cloud

Figure 2.
Cassandra on virtual machines.

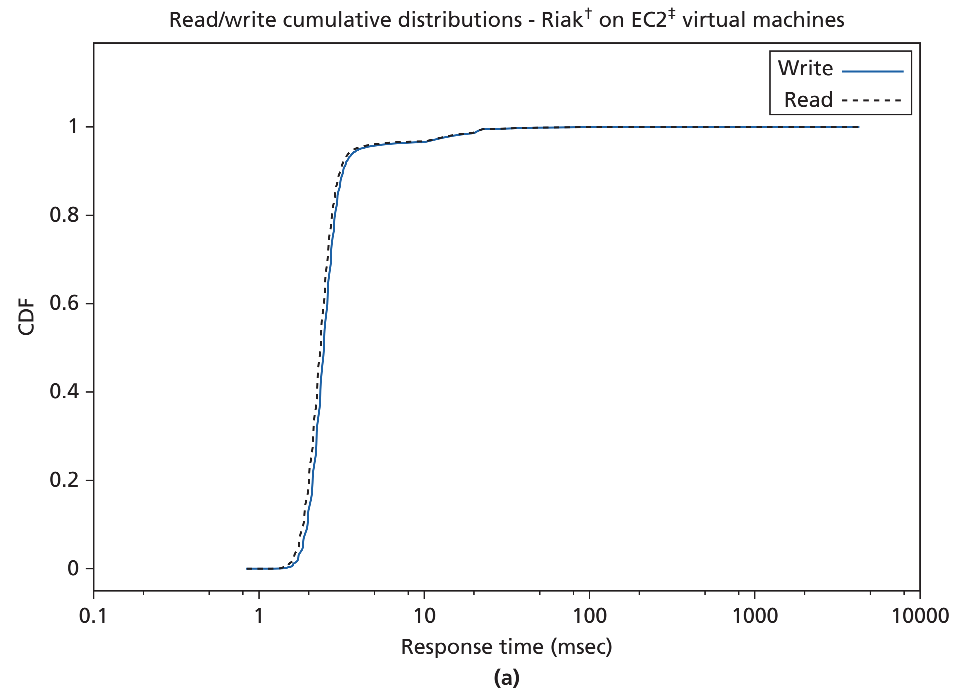


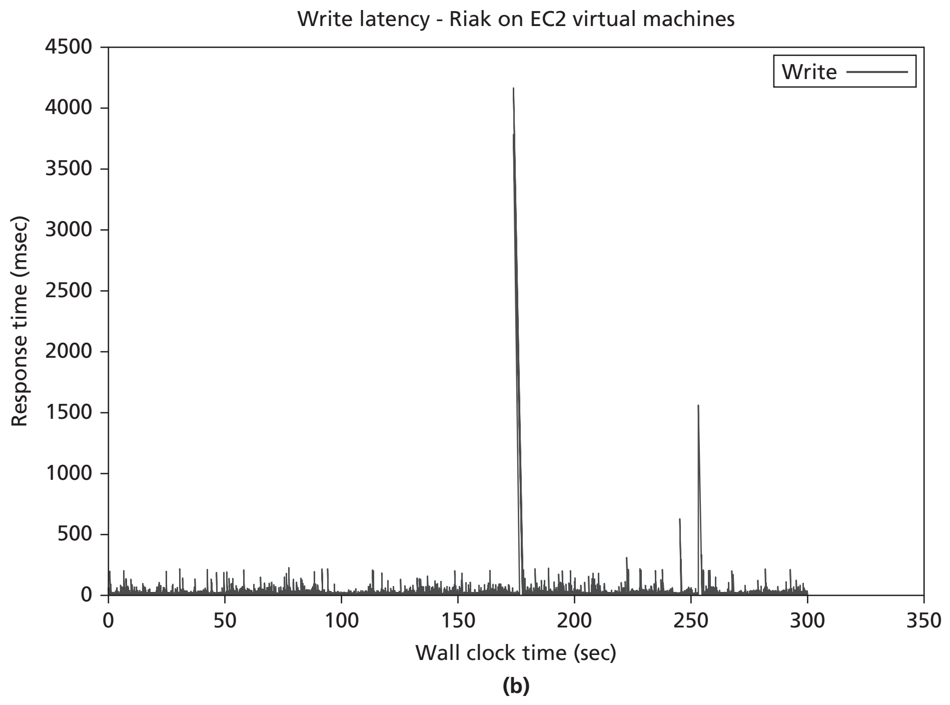


†Registered trademark of Basho Technologies, Inc.

CDF—Cumulative distribution function

Figure 3.
Riak on physical machines.

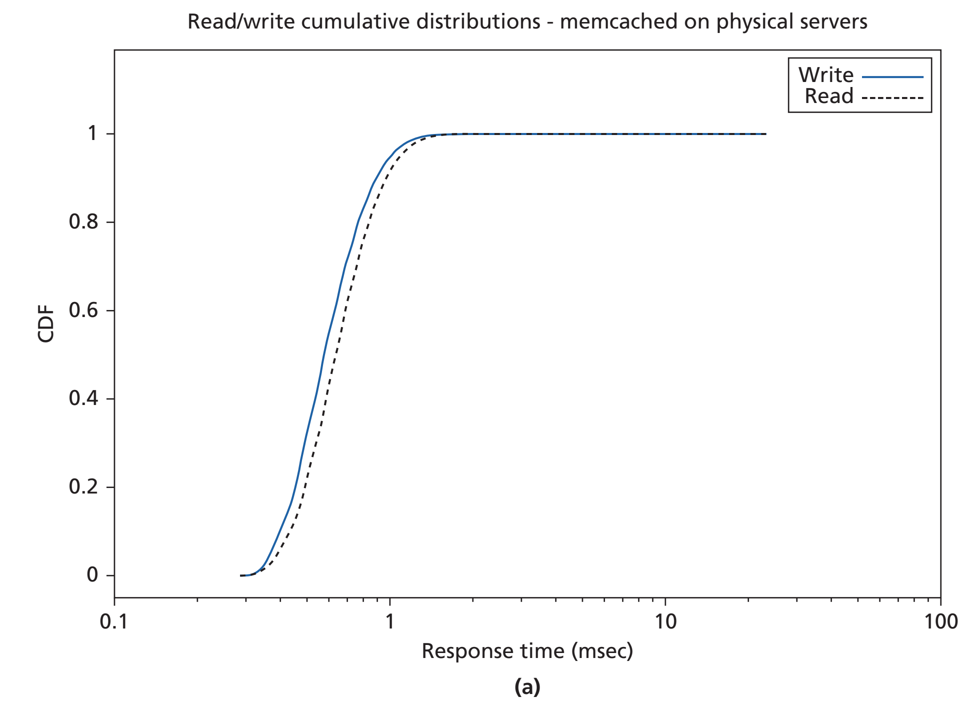


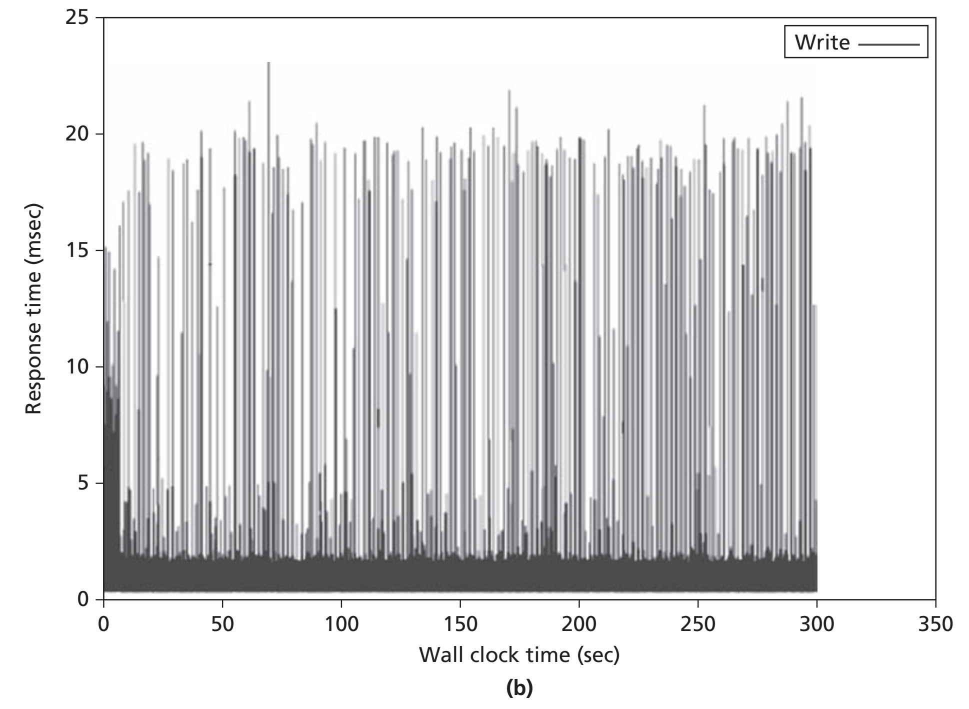


[†]Registered trademark of Basho Technologies, Inc.
[‡]Trademark of Amazon Technologies.

CDF—Cumulative distribution function
EC2—Elastic Compute Cloud

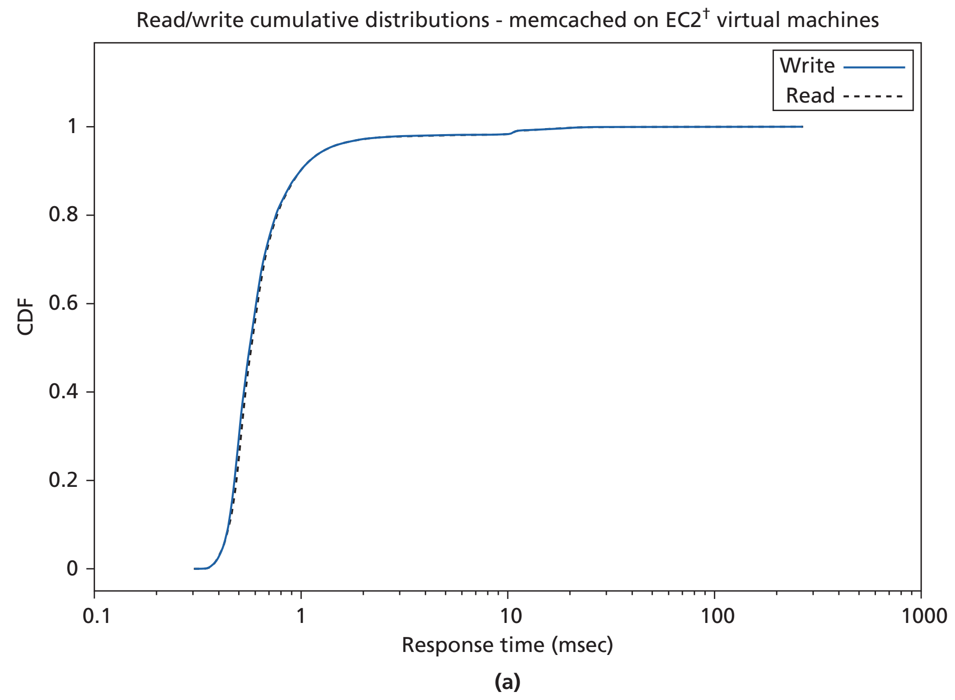
Figure 4.
Riak on virtual machines.

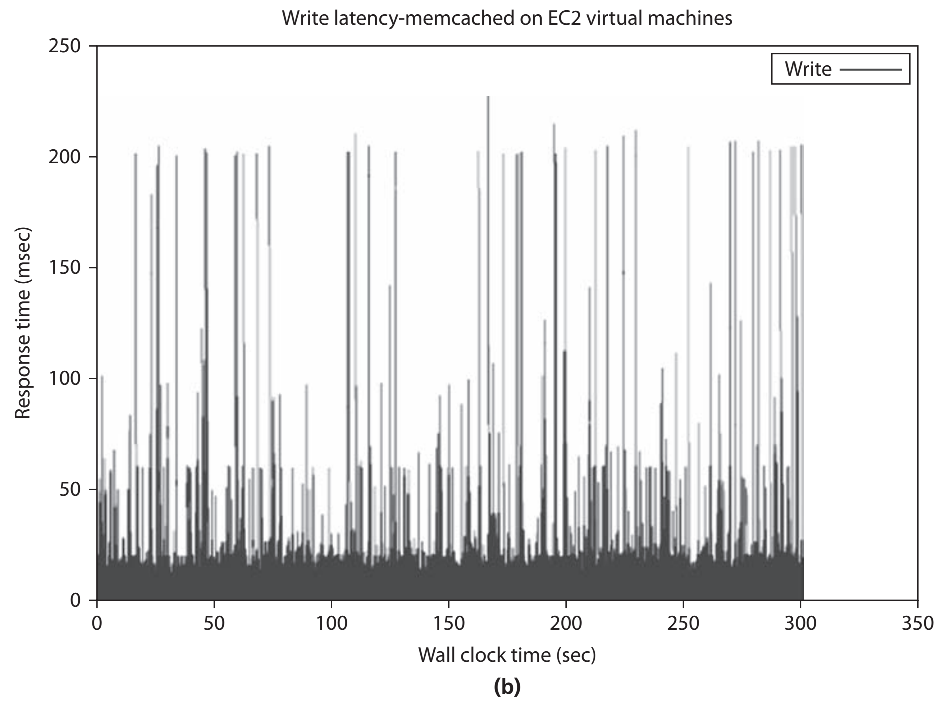




CDF—Cumulative distribution function

Figure 5.
Memcached on physical machines.





†Trademark of Amazon Technologies.

CDF—Cumulative distribution function
EC2—Elastic Compute Cloud

Figure 6.
Memcached on virtual machines.

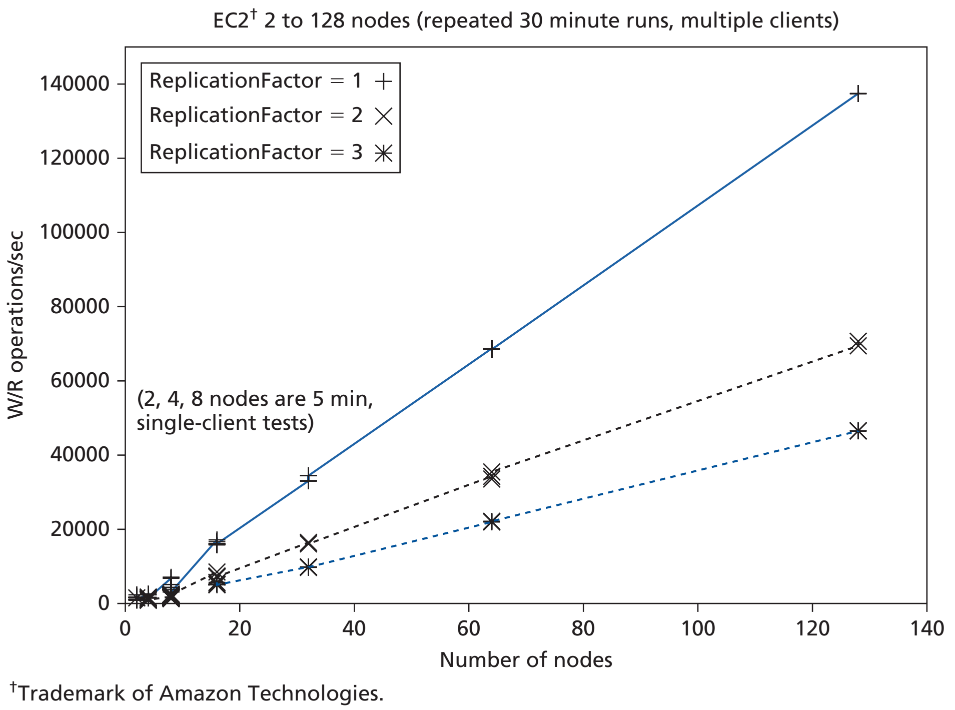
Cassandra scaling. These results, illustrated in **Figure 7**, were obtained by testing Cassandra using a range of cluster sizes (4 to 128) nodes, and a range of replication factors. Machines are m1.large (dual core) EC2 machines. Each node runs both a Cassandra server and a client running 50 threads. All nodes were running in a single EC2 region (US East).

Riak scaling. These results, illustrated in **Figure 8**, were obtained by testing Riak using a range of cluster sizes (4 to 128) nodes, and a range of replication factors, which Riak calls the "n_val." Both read and write are set to "quorum" (r = quorum, w = quorum). The machines are m1.large (dual core) EC2 machines. Each node runs both a Riak server and a client running 50 threads. Using multiple test clients helps the load scale up with the number of nodes, and running local clients simplifies the test procedure. All nodes were running in a single EC2 region (US East). The RF = 2 and RF = 3 curves are fairly close together because for both 2 and 3 the quorum is 2. When the n_val increases to 4, the quorum value increases to 3.

Flurry: A System for Mitigating Latency Outliers

One method of mitigating delays on a reliable distributed data store is to use the first correct response replies from an ensemble of individually unreliable data servers rather than waiting for a quorum of responses or for all responses. This will smooth out temporary delays that may affect a subset of the data servers during the duration of any given distributed operation.

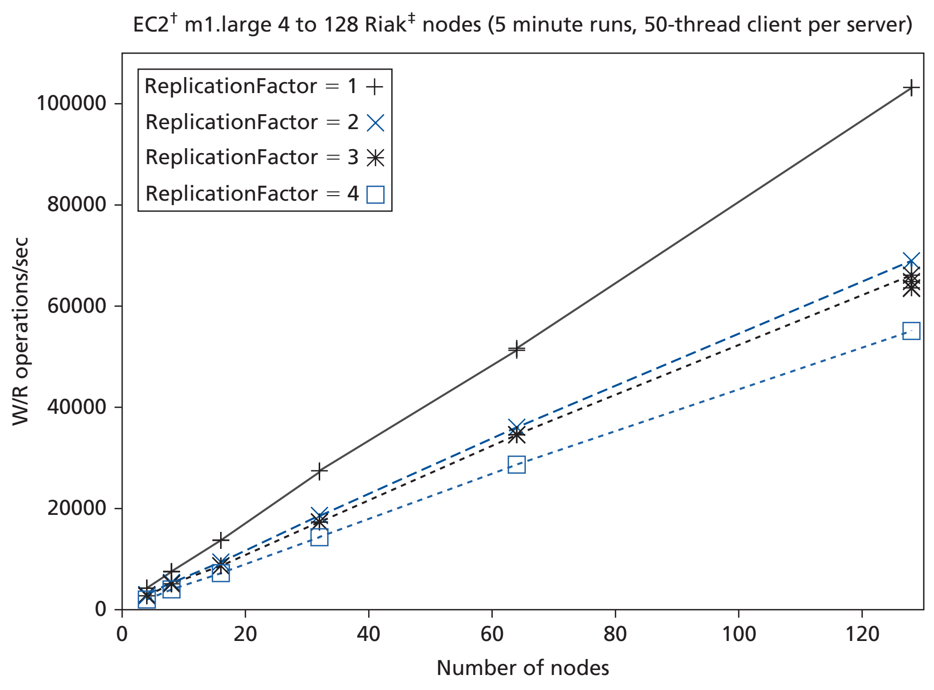
The problem then becomes how to determine whether any particular response from an arbitrary data server in an ensemble of data servers is a correct response. For a subset of potential systems we can use vector clocks to determine whether a message is correct if the following restrictions are observed:



†Trademark of Amazon Technologies.

EC2—Elastic Compute Cloud
W/R—Write/read

Figure 7.
Cassandra scaling.



†Trademark of Amazon Technologies.

‡Registered trademark of Basho Technologies, Inc.

EC2—Elastic Compute Cloud
W/R—Write/read

Figure 8.
Risk scaling.

1. Only a single client will access the data for a particular key.
2. The client can provide an ordered sequence identifier (ID) for each of the operations on the data associated with a key.

These restrictions can be met for a class of telecom applications where a single client (e.g., a mobile handset) is interacting with an application where the data about that interaction is maintained as session data and stored with a key unique for that session. The client also needs to provide the application with a sequence number for the message within the session, but this is typically available in many protocols as a method to prevent replay attacks.

The Flurry reliable distributed database test bed was implemented to allow us to conduct experiments which would test whether existing real-world cloud implementation systems can be used as a base for classes of telecom applications that meet the aforementioned restrictions, and will exhibit the latency characteristics which make those architectures feasible. Flurry, which was developed by our team at Bell Labs, is not a fully implemented reliable distributed data store like commercial systems such as Cassandra and Riak, and as such can't be directly compared with those systems. It does, however, allow us to compare the various algorithms used to determine a correct response, and its degree of instrumentation allows us to explore how the algorithms and architectural choices handle failures such as dropped or delayed messages, and network isolation events in a controlled environment by injecting those error situations into the experiments using the test bed code itself.

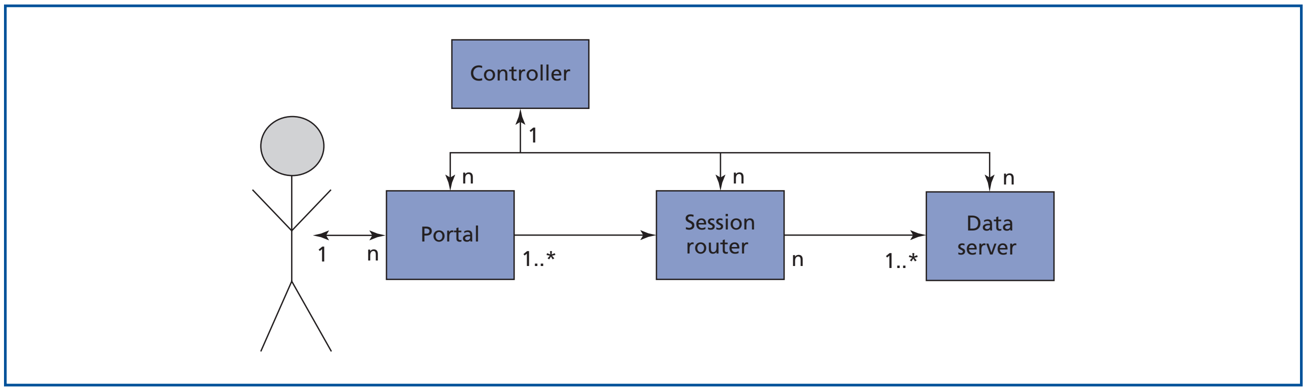


Figure 9.
Components of the Flurry system.

The Flurry design may be described as an object model where major components are implemented as objects and communication occurs by passing a message object ("FlurryPayload") between those components. The FlurryPayload contains not only the data needed to describe the request (e.g., the key, value, and type of command), but also the routing data for that message, as well as instrumentation data such as time stamps and errors that should be introduced when processing the message.

The components of the Flurry system, illustrated in **Figure 9**, include the

- *Portal,* which provides methods for the client to interact with the Flurry system.
- *Session router,* which uses the key for a read/write operation to algorithmically determine (e.g., using distributed hash tables) which set of data servers hold the data for that key. It then sends a copy of the message to each of those data servers. The session router forwards the response from a data server to the portal when it meets the correctness criteria for the specified algorithm (first correct response, quorum, or all-in).
- *Data server,* which provides the physical storage for the key-value store. It also provides the checking (vector clock) to determine whether the operation in the message can be satisfied with its current version of the data stored on that server.
- *Controller,* which provides the mechanism for managing the configuration of the Flurry system, namely providing information on where the various components are being hosted (which system, which port, and which transport mechanism should be used to route a message from one component to another).

Each data server is required to check the vector clock to see if a particular data server is able to satisfy the requested operation on the version of the data stored on that server. If the check fails, an error message is returned to the session router, which can forward the correct response received from a different data server to allow the data server to catch-up to the current version of the data.

Flurry Implementation

The Flurry test bed is implemented as a static library in C++. It uses Google protocol buffers to marshal the data in the messages passed between components, sockets to transport messages between components in different processes, and direct method calls as transport between component instances in the same process. The Flurry library uses pthreads to manage asynchronous operations. It was developed to run in a generic Linux*/POSIX environment.

Each process has a single instance of the FlurryController which will bring up the components that are defined by the configuration to be resident within the process. The FlurryController also spawns a thread for each port on which the process is configured to receive messages. Messages to components which are external to the process are routed over a socket to the process which contains the component. The Flurry test bed is designed to allow for the use of either User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) sockets although the initial experiments were conducted with UDP.

The “session router” is implemented as a separate component from the “portal” to allow for experiments which distribute those behaviors, although for the experiments using the “first correct response,” the “FlurryPortal” and “FlurrySessionRouter” objects are co-located in a process so that they have the same availability. There is one FlurryPortal instance for each client thread doing Flurry queries.

The client can specify which correctness algorithm should be used by Flurry on a per-message (or system default) basis. The FlurrySessionRouter tracks all the messages for each query in a log so performance can be compared between algorithms. That way the client can have its query satisfied by the first response but we still log the information on when the remaining responses arrive so we can also determine how the client would have performed using one of the other algorithms.

For the test bed, the data store was implemented as a simple in-memory hash table without any long term persistence since that wasn’t a focus of the research.

Evaluation of Flurry Experiments

We tested the Flurry distributed database test bed using the same client as the tests run on the commercial distributed databases. Since Flurry also allows us to inject delays and message losses, we are able to simulate the behaviors observed in the commercial databases with respect to the average response characteristics as well as the outliers. Flurry was tested in the same configurations used for the tests of the commercial NoSQL databases described earlier and also tested with simulated delays.

Flurry with simulated server delays. A configuration of three data servers and a single test client were used for this illustrative test run, shown in **Figure 10**. A data replication factor of 3 was used to allow a distinction

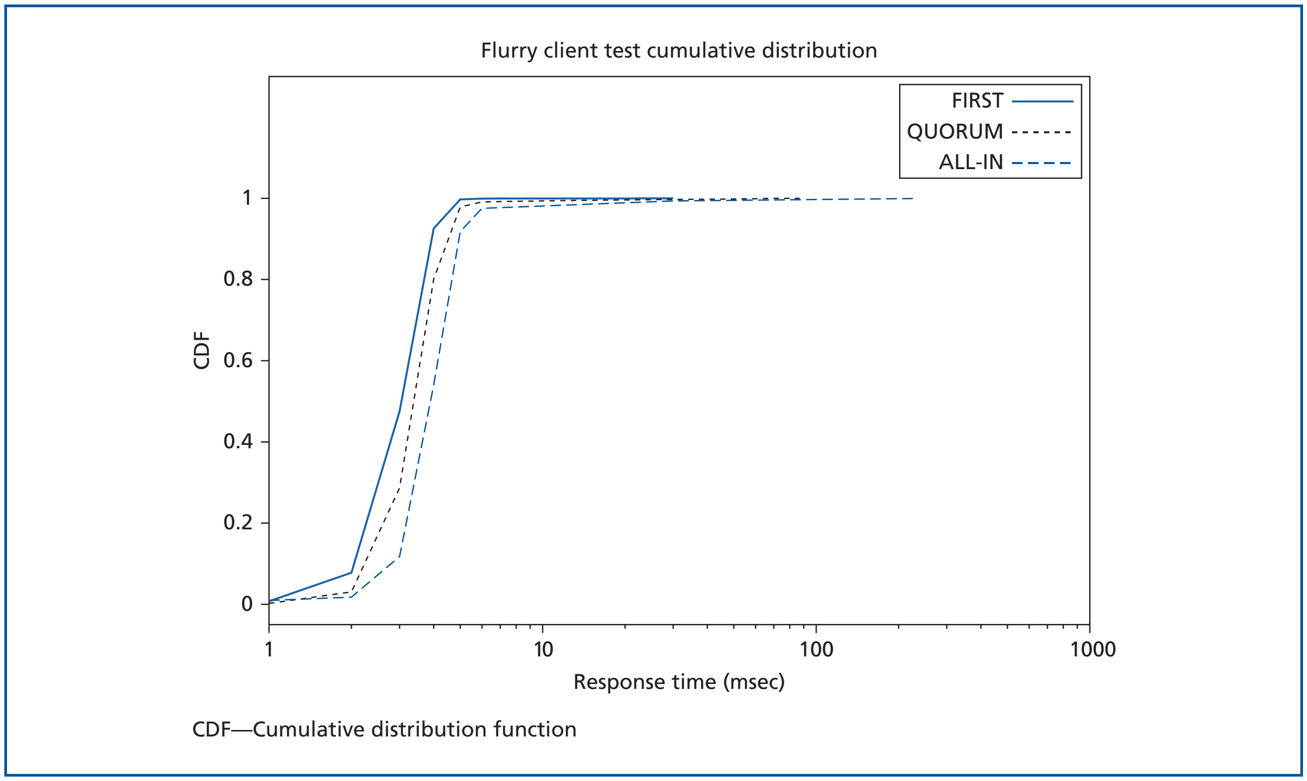


Figure 10.
Flurry with simulated server delays.

between the different algorithms tested. For a sunny day transaction scenario, first response would need one response message, quorum would need two response messages, and all-in would need three response messages.

The average values for the read and write transactions, shown in **Table I**, are very similar across all three algorithms, with the first response algorithm being slightly better than the quorum.

Table I. Average values for read and write transactions.

	First response	Quorum	All-in
Average response (milliseconds)	3.5	4.1	5.2
Maximum response (milliseconds)	30	87	247

In this case with simulated network and data server delays, the first response algorithm is able to mitigate the response latency caused by data server delays.

Flurry on virtual machines. With flurry running on Amazon EC2 m1.large machines, the CDF graph provided in **Figure 11** shows the first correct response algorithm performing better than the quorum or all-in algorithms in this illustrative three server cluster.

We can observe a number of outliers in the plot of the write latency, shown in **Figure 12**, when using the quorum algorithm.

When we look at the same data set, this time using the first correct response algorithm for processing the data, we can see that several of the outliers have been removed, as shown in **Figure 13**.

With the runs in the Amazon EC2 cloud, we observed that about 20 percent of the outlier latencies reported with the quorum operator are removed when using the first response algorithm. In the data

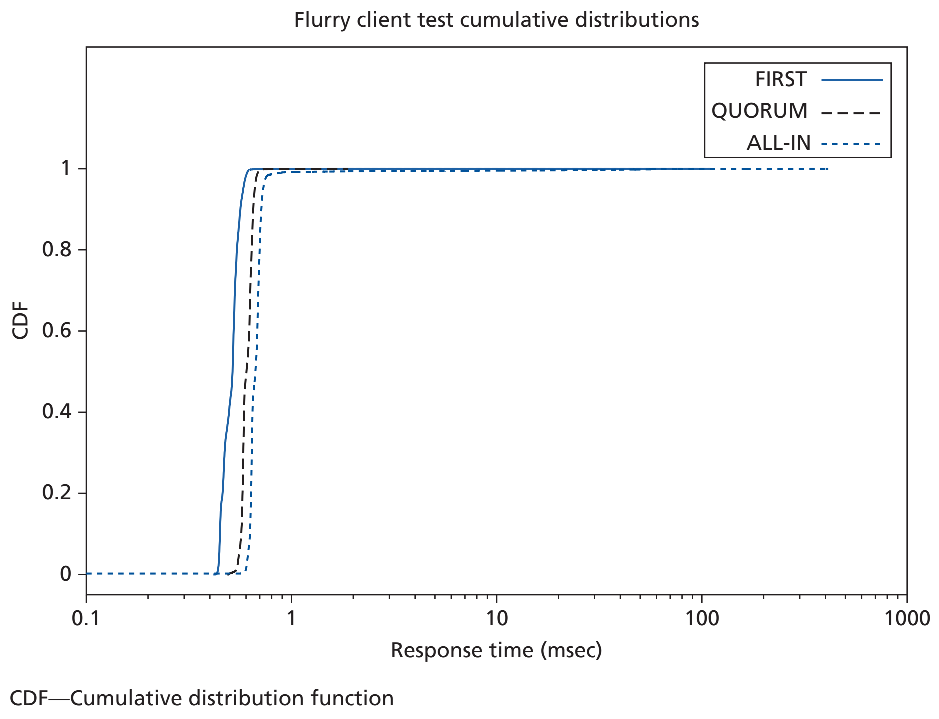
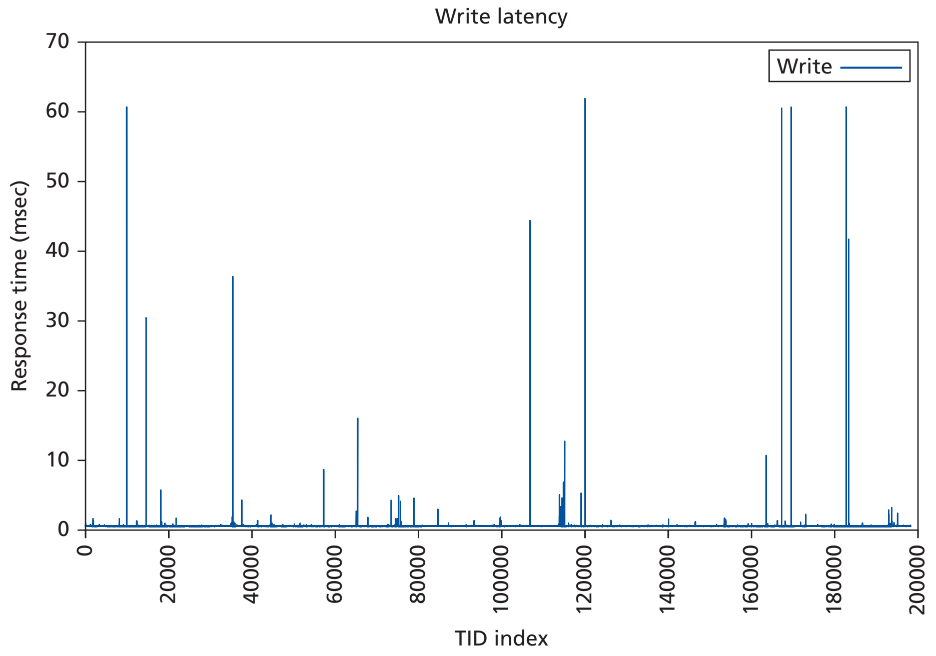
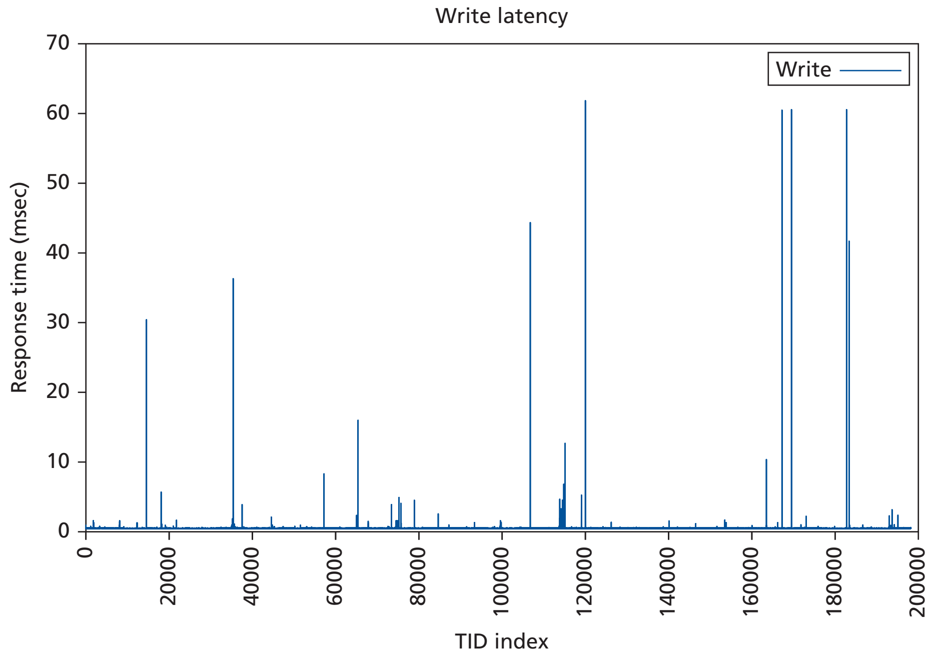


Figure 11.
Flurry on virtual machines.



TID—Tuple-identifier

Figure 12.
Outliers when using quorum.



TID—Tuple-identifier

Figure 13.
Outliers removed with use of first correct response algorithm.

Table II. Transactions with the highest latency.

TID	First response	Quorum	All-in
71255	36.3	36.4	68.7
367575	41.7	41.7	45.7
214333	44.4	44.4	44.8
76466	0.6	60.3	133.3
335343	60.5	60.5	60.6
339919	60.6	60.7	60.7
366369	60.6	60.7	60.8
20273	0.5	60.7	60.8
240813	61.8	61.9	92.1
73190	110.1	110.1	110.2

TID—Tuple-identifier index

set for the latency plots, the 10 transactions with the highest latency are shown in **Table II**, along with the amount of time (in milliseconds) that each of those transactions would have taken with the three algorithms tested.

The first correct response algorithm does a good job of mitigating delays that affect individual data servers, such as those caused by VM context changes or dropped UDP packets, although even using first response, we were still seeing enough outliers when running on Amazon EC2 to prevent us from meeting the 99.999 percent availability within the budgeted time. The number of outliers is related to the load placed on the system in that as we increased the number of clients reading and writing data, we saw the number of outliers increase, but even with a very light load the occurrence of outliers did not go to zero.

We instrumented Flurry to record the time stamp when the kernel posted the UDP packet to the socket, as well as when the Flurry application received the packet for processing as suggested by earlier performance studies [17, 18] on the Amazon EC2 cloud. This allowed us to observe that the outliers that remained after we applied the first response algorithm were caused by delays in the client code. We saw instances of several hundred milliseconds between when the kernel time-stamped the arriving UDP packet and posted it to the socket, and when the application completed the "recvfrom" system call on the socket to process the packet. The measurements were recorded on both the clients and the data servers.

In this example run on Amazon EC2, "Hop1" refers to time on the data server from the receipt of the request packet and posting to socket by the kernel and the processing of that packet by the Flurry application. "Hop2" refers to the time on the client machine from the receipt of the response packet until the Flurry application was able to process that packet.

Looking at 6,758,460 messages, we observed:

Hop1: Min = .012 milliseconds Max = 62.015 milliseconds

Hop2: Min = .013 milliseconds Max = 756.626 milliseconds

When running the Flurry client on physical machines and the data servers on virtual machines, we see the number of outliers drop off dramatically.

Processed 3,037,986 messages:

Hop1: Max = 542.226 milliseconds (136 outliers above 100 milliseconds)

Hop2: Max = 205.603 milliseconds (5 outliers above 100 milliseconds)

Since the Hop1 latency was distributed between the data servers, using the quorum and first response algorithms ensured that the system was not affected by those latencies. The Hop2 latency shown was not affected by the choice of algorithm, but occurred infrequently enough to meet our budget.

Conclusion

A highly-available low-latency distributed data store is critical to a cloud-based implementation for most telecommunication applications. We considered several existing database systems that were selected to comprehensively cover the most promising state-of-the-art solutions, and we conducted experiments to thoroughly evaluate their scaling and latency characteristics. Our results confirm their excellent performance with respect to scaling and *average* latencies. However, we also show, somewhat surprisingly, that the 99.999th percentile of latencies can be worse than 10 times the average latencies. To our knowledge, this is the first study of the fine-grained *distribution* of latencies. In recent work [16], the impact of the latency performance of distributed database systems has been experimentally studied—however, that work considers worst-case (as opposed to *probabilistic*) latencies and the solutions proposed are based on real-time scheduling. We presented a new system which we call Flurry that uses the first response from a replica and a checking algorithm based on vector clocks to determine the correctness of a response in the presence of message losses. While the notion of vector clocks is not particularly new, previous applications have been limited to determining causality, and our application for handling message losses seems novel. While the idea of reducing the number of replicas accessed was previously considered in [7], its application was limited to reads with writes still being performed on all replicas. Flurry is not yet as robust or mature in comparison to commercial systems. However, our experimental evaluation of Flurry shows that the idea of using first response, besides improving average latencies, can significantly improve the distribution characteristics of latencies.

We have identified a class of systems for which the Flurry vector-checking algorithm is applicable. Specifically, these are client-server systems with redundancy and high availability limited to the server. In future work, we plan to devise extensions of the checking algorithm to the more general setting of fully distributed peer-to-peer systems with a more formal analysis of its correctness properties. More generally, we are investigating the end-to-end design of a cloud-based system for achieving low latencies with high availability. The simplest way to use a reliable data store directly is to decouple the message processing from the data processing by having a set of replicated stateless message processors that process incoming messages and use the reliable data store for reading and updating the session state. This design achieves efficient parallelism in dispatching and processing incoming messages but its overall performance is limited by the latency characteristics of the data store. We are therefore also investigating an alternate design where the data is co-located with its processing elements as a set of replicated stateful components with no data sharing among different components. In such a system, the data access times are significantly reduced, but more elaborate replication algorithms need to be devised and any resulting improvement in the overall performance still requires evaluation.

***Trademarks**

Amazon EC2 is a trademark of Amazon Technologies.
Java and JavaScript are trademarks of Sun Microsystems, Inc.
Linux is a trademark of Linus Torvalds.
Riak is a registered trademark of Basho Technologies, Inc.

References

[1] Amazon Web Services, "Amazon ElastiCache: Getting Started Guide," API Version 2011-07-15, 2011, <http://awsdocs.s3.amazonaws.com/ElastiCache/latest/elasticache-gsg.pdf>.

[2] J. C. Anderson, J. Lehnardt, and N. Slater, CouchDB: The Definitive Guide, O'Reilly Media, Sebastopol, CA, 2010.

[3] Basho Technologies, "Riak", <http://wiki.basho.com/Riak.html>.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," Proc. 7th USENIX Symp. on Operating Syst. Design and Implementation (OSDI '06) (Seattle, WA, 2006).

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proc. 6th Symp. on Operating Syst. Design and Implementation (OSDI '04) (San Francisco, CA, 2004).

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," Proc. 21st ACM SIGOPS Symp. on Operating Syst. Principles (SOSP '07) (Stevenson, WA, 2007), pp. 205–220.

[7] A. El Abbadi, D. Skeen, and F. Cristian, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Syst. (PODS '85) (Portland, OR, 1985), pp. 215–229.

[8] C. J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88) (Brisbane, Aus., 1988), pp. 56–66.

[9] B. Fitzpatrick, "Distributed Caching with Memcached," Linux J., Aug. 1, 2004,

<http://www.linuxjournal.com/article/7451?page=0,0>.

[10] J. Gabrielsson, O. Hubertsson, I. Más, and R. Skog, "Cloud Computing in Telecommunications," Ericsson Rev., 1 (2010), 29–33, <http://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2010/cloudcomputing.pdf>.

[11] D. K. Gifford, "Weighted Voting for Replicated Data," Proc. 7th ACM Symp. on Operating Syst. Principles (SOSP '79) (Pacific Grove, CA, 1979), pp. 150–162.

[12] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," ACM SIGACT News, 33:2 (2002), 51–59.

[13] IBM, "SK Telecom Builds Cloud Computing Platform with IBM," Press Release, Dec. 16, 2009, <http://www-03.ibm.com/press/us/en/pressrelease/29041.wss>.

[14] A. Lakshman and P. Malik, "Cassandra – A Decentralized Structured Storage System," ACM SIGOPS Operating Syst. Rev., 44:2 (2010), 35–40.

[15] F. Mattern, "Virtual Time and Global States of Distributed Systems," Proc. Internat. Workshop on Parallel and Distrib. Algorithms (Chateau de Bonas, Gers, Fra., 1988), pp. 215–226.

[16] Y. J. Singh, Y. S. Singh, A. Gaikwad, and S. C. Mehrotra, "Dynamic Management of Transactions in Distributed Real-Time Processing System," Internat. J. Database Management Syst., 2.2 (2010), 161–170.

[17] G. Wang and T. S. E. Ng, "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," Proc. 29th IEEE Internat. Conf. on Comput. Commun. (INFOCOM '10) (San Diego, CA, 2010).

[18] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining Packet Delays Under Virtualization," ACM SIGCOMM Comput. Commun. Rev., 41:1 (2011), 38–44.

(Manuscript approved March 2012)

FANGZHE CHANG is a member of technical staff at Bell Labs in Murray Hill, New Jersey. His current research focuses on distributed computing, service composition, and networking systems. Dr.Chang received his bachelor's degree from the Changsha Institute of Technology and his master's degree from the Institute of Software, Academia Sinica, both in the Peoples Republic of China, and received his Ph.D. in computer science from the Courant Institute of Mathematical Sciences at New York University in New York City.

PETER S. FALES is a member of technical staff in Bell Labs Service Infrastructure research department and is based in Naperville, Illinois. He has a bachelor's degree in electrical engineering with computer science from the University of Colorado in Boulder, Colorado, and a master's degree in electrical engineering from Stanford University in Palo Alto, California. Mr. Fales has been with AT&T, Lucent Technologies, and Alcatel-Lucent for 30 years, and began his career in AT&T's Computer System Division. He has worked in software development areas associated with both wireline and wireless switching systems and for the past 10 years he has been the Central Administrator for Alcatel-Lucent Exptools, a large collection of open-source and proprietary tools provided collaboratively and used by developers throughout Alcatel-Lucent. His interests include open-source software, network applications, and ways to use software tools to improve productivity.

MORITZ STEINER is a member of technical staff at Bell Labs in Murray Hill, New Jersey. He received his M.S. degree (Diplom) in computer science from the University of Mannheim in Germany, and his Ph.D. degree in computer networks from jointly from Telecom ParisTech, France and the University of Mannheim. His doctoral thesis investigates how to build virtual network environments from unstructured peer-to-peer networks. It also introduced measurement techniques and presented extensive measurement results on a real world, large-scale, structured peer-to-peer file sharing network, named Kad. His research interests and project activities are in the areas of analysis and design of peer-to-peer networks and cloud computing.

RAMESH VISWANATHAN is a member of technical staff in Bell Labs' Enabling Computing Technologies research domain, and is based in Murray Hill, New Jersey. He is broadly interested in the application of mathematical logic and formal methods to deriving precise and systematic solutions for problems arising in the practice of software systems and *networks. His current work focuses on cloud deployment of telecommunication services and specification logics, synthesis and verification for automatic service composition. Previously, he has worked on semantics for functional, imperative, and object-oriented languages; virtual multimedia environments for supporting collaboration; alarm correlation for network management; topology discovery for public Internet Protocol (IP) networks; logics for compositional verification; online monitoring techniques for detecting and locating faults in deployed networks; analysis of Border Gateway Protocol (BGP) convergence; and protocols for inter-domain quality of service (QoS)-aware routing. He received a B.Tech in computer science and engineering from the Indian Institute of Technology in Kanpur, and a Ph.D. in computer science from Stanford University in California. Dr. Viswanathan was a Rosenbaum Fellow at the Isaac Newton Institute for Mathematical Sciences in Cambridge University, UK, from 1995 to 1996.*

THOMAS J. WILLIAMS is a distinguished member of technical staff in Bell Labs' Service Infrastructure Research Domain, and is based in Columbus, Ohio. He has a B.S. in computer science from Ohio University, Athens, Ohio and a M.S. in computer science from Case Western Reserve University in Cleveland. He began his career almost 30 years ago with AT&T's Western Electric division, and worked in operations support and network management systems software development before moving to Bell Labs. Over the past dozen years, he has held various research positions in the Bell Labs Advanced Technologies Software Technology Center, in Bell Labs Ventures, and in Bell Labs Research. Mr. Williams' holds one patent. His interests include software architecture, database systems, and agile development techniques, and his current focus is on cloud-based distributed real time data services and architectures.

THOMAS L. WOOD is a director in Bell Labs' Enabling Computing Technologies research domain and is based in Holmdel, New Jersey. Hired into Bell Labs' Government Communication Center, he has been with the company for over 25 years, and has worked on a variety of projects including large-scale control systems, image processing, and real time media processing. He led a team that created Voice over Internet Protocol (VoIP), IP traffic-shaping technology, and a hardware architecture that was deployed as part of a fiber-to-the-home solution. The technology was adapted and deployed as part of the company's Line Access Gateway product. Mr. Wood also served as a Brookings Congressional Fellow in the office of Senator Bill Frist. He has a B.S.E.E. from Rensselaer Polytechnic Institute in Troy, New York, and an M.S.C.S. from Columbia University in New York City. ♦