Cassandra - A Decentralized Structured Storage System

Avinash Lakshman
Facebook

Prashant Malik
Facebook

ABSTRACT

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers). At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

1. INTRODUCTION

Facebook runs the largest social networking platform that serves hundreds of millions users at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Facebook's platform in terms of performance, reliability and efficiency, and to support *continuous growth* the platform needs to be highly scalable. Dealing with failures in an infrastructure comprised of thousands of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such, the software systems need to be constructed in a manner that treats failures as the norm rather than the exception. To meet the reliability and scalability needs described above Facebook has developed Cassandra.

Cassandra uses a synthesis of well known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. Inbox Search is a feature that enables users to search through their Facebook Inbox. At Facebook this meant the system was required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. Since users are served from data centers that are geographically distributed, being able to replicate data across data centers was key to keep search latencies down. Inbox Search was launched in June of 2008 for around 100 million users and today we are at over 250 million users and Cassandra has kept up the promise so far. Cassandra is now deployed as the backend storage system for multiple services within Facebook.

This paper is structured as follows. Section 2 talks about related work, some of which has been very influential on our design. Section 3 presents the data model in more detail. Section 4 presents the overview of the client API. Section 5 presents the system design and the distributed algorithms that make Cassandra work. Section 6 details the experiences of making Cassandra work and refinements to improve performance. In Section 6.1 we describe how one of the applications in the Facebook platform uses Cassandra. Finally Section 7 concludes with future work on Cassandra.

2. RELATED WORK

Distributing data for performance, availability and durability has been widely studied in the file system and database communities. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus[14] and Coda[16] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. Farsite[2] is a distributed file system that does not use any centralized server. Farsite achieves high availability and scalability using replication. The Google File System (GFS)[9] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunk servers. However the GFS master is now made fault tolerant using the Chubby[3] abstraction. Bayou[18] is a distributed relational database system that allows disconnected operations and provides eventual data consistency. Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them however, guarantee eventual consistency. Similar to these systems, Dynamo[6] allows read and write operations to continue even during network partitions and resolves update conflicts using different conflict resolution mechanisms, some client driven. Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency of replicated data. Although strong consistency provides the application writer a con-

venient programming model, these systems are limited in scalability and availability [10]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

Dynamo[6] is a storage system that is used by Amazon to store and retrieve user shopping carts. Dynamo's Gossip based membership algorithm helps every node maintain information about every other node. Dynamo can be defined as a structured overlay with at most one-hop request routing. Dynamo detects updated conflicts using a vector clock scheme, but prefers a client side conflict resolution mechanism. A write operation in Dynamo also requires a read to be performed for managing the vector timestamps. This is can be very limiting in environments where systems need to handle a very high write throughput. Bigtable[4] provides both structure and data distribution but relies on a distributed file system for its durability.

3. DATA MODEL

A table in Cassandra is a distributed multi dimensional map indexed by a key. The value is an object which is highly structured. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Columns are grouped together into sets called column families very much similar to what happens in the Bigtable[4] system. Cassandra exposes two kinds of columns families, Simple and Super column families. Super column families can be visualized as a column family within a column family.

Furthermore, applications can specify the sort order of columns within a Super Column or Simple Column family. The system allows columns to be sorted either by time or by name. Time sorting of columns is exploited by application like Inbox Search where the results are always displayed in time sorted order. Any column within a column family is accessed using the convention $column\_family : column$ and any column within a column family that is of type super is accessed using the convention $column\_family : super\_column : column$. A very good example of the super column family abstraction power is given in Section 6.1. Typically applications use a dedicated Cassandra cluster and manage them as part of their service. Although the system supports the notion of multiple tables all deployments have only one table in their schema.

4. API

The Cassandra API consists of the following three simple methods.

- $insert(table, key, rowMutation)$
- $get(table, key, columnName)$
- $delete(table, key, columnName)$

$columnName$ can refer to a specific column within a column family, a column family, a super column family, or a column within a super column.

5. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have the following characteristics; scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is beyond the scope of this paper, so we will focus on the core distributed systems techniques used in Cassandra: partitioning, replication, membership, failure handling and scaling. All these modules work in synchrony to handle read/write requests. Typically a read/write request for a key gets routed to any node in the Cassandra cluster. The node then determines the replicas for this particular key. For writes, the system routes the requests to the replicas and waits for a quorum of replicas to acknowledge the completion of the writes. For reads, based on the consistency guarantees required by the client, the system either routes the requests to the closest replica or routes the requests to all replicas and waits for a quorum of responses.

5.1 Partitioning

One of the key design features for Cassandra is the ability to scale incrementally. This requires, the ability to dynamically partition the data over the set of nodes (i.e., storage hosts) in the cluster. Cassandra partitions data across the cluster using consistent hashing [11] but uses an order preserving hash function to do so. In consistent hashing the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its *position* on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. This node is deemed the coordinator for this key. The application specifies this key and the Cassandra uses it to route requests. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principal advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected. The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. Typically there exist two ways to address this issue: One is for nodes to get assigned to multiple positions in the circle (like in Dynamo), and the second is to analyze load information on the ring and have lightly loaded nodes move on the ring to alleviate heavily loaded nodes as described in [17]. Cassandra opts for the latter as it makes the design and implementation very tractable and helps to make very deterministic choices about load balancing.

5.2 Replication

Cassandra uses replication to achieve high availability and durability. Each data item is replicated at N hosts, where N

is the replication factor configured "per-instance". Each key, k, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides the client with various options for how data needs to be replicated. Cassandra provides various replication policies such as "Rack Unaware", "Rack Aware" (within a datacenter) and "Datacenter Aware". Replicas are chosen based on the replication policy chosen by the application. If certain application chooses "Rack Unaware" replication strategy then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring. For "Rack Aware" and "Datacenter Aware" strategies the algorithm is slightly more involved. Cassandra system elects a leader amongst its nodes using a system called Zookeeper[13]. All nodes on joining the cluster contact the leader who tells them for what ranges they are replicas for and leader makes a concerted effort to maintain the invariant that no node is responsible for more than N-1 ranges in the ring. The metadata about the ranges a node is responsible is cached locally at each node and in a fault-tolerant manner inside Zookeeper - this way a node that crashes and comes back up knows what ranges it was responsible for. We borrow from Dynamo parlance and deem the nodes that are responsible for a given range the "preference list" for the range.

As is explained in Section 5.1 every node is aware of every other node in the system and hence the range they are responsible for. Cassandra provides durability guarantees in the presence of node failures and network partitions by relaxing the quorum requirements as described in Section5.2. Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Cassandra is configured such that each row is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple datacenters. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without any outage.

5.3 Membership

Cluster membership in Cassandra is based on Scuttlebutt[19], a very efficient anti-entropy Gossip based mechanism. The salient feature of Scuttlebutt is that it has very efficient CPU utilization and very efficient utilization of the gossip channel. Within the Cassandra system Gossip is not only used for membership but also to disseminate other system related control state.

5.3.1 Failure Detection

Failure detection is a mechanism by which a node can locally determine if any other node in the system is up or down. In Cassandra failure detection is also used to avoid attempts to communicate with unreachable nodes during various operations. Cassandra uses a modified version of the Φ Accrual Failure Detector[8]. The idea of an Accrual Failure Detection is that the failure detection module doesn't emit a Boolean value stating a node is up or down. Instead the failure detection module emits a value which represents a suspicion level for each of monitored nodes. This value is defined as Φ. The basic idea is to express the value of Φ on a scale that is dynamically adjusted to reflect network and load conditions at the monitored nodes.

Φ has the following meaning: Given some threshold Φ, and assuming that we decide to suspect a node A when $\Phi = 1$, then the likelihood that we will make a mistake (i.e., the decision will be contradicted in the future by the reception of a late heartbeat) is about 10%. The likelihood is about 1% with $\Phi = 2$, 0.1% with $\Phi = 3$, and so on. Every node in the system maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster. The distribution of these inter-arrival times is determined and Φ is calculated. Although the original paper suggests that the distribution is approximated by the Gaussian distribution we found the Exponential Distribution to be a better approximation, because of the nature of the gossip channel and its impact on latency. To our knowledge our implementation of the Accrual Failure Detection in a Gossip based setting is the first of its kind. Accrual Failure Detectors are very good in both their accuracy and their speed and they also adjust well to network conditions and server load conditions.

5.4 Bootstrapping

When a node starts for the first time, it chooses a random token for its position in the ring. For fault tolerance, the mapping is persisted to disk locally and also in Zookeeper. The token information is then gossiped around the cluster. This is how we know about all nodes and their respective positions in the ring. This enables any node to route a request for a key to the correct node in the cluster. In the bootstrap case, when a node needs to join a cluster, it reads its configuration file which contains a list of a few contact points within the cluster. We call these initial contact points, seeds of the cluster. Seeds can also come from a configuration service like Zookeeper.

In Facebook's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. Failures can be of various forms such as disk failures, bad CPU etc. A node outage rarely signifies a permanent departure and therefore should not result in re-balancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Cassandra nodes. To that effect every message contains the cluster name of each Cassandra instance. If a manual error in configuration led to a node trying to join a wrong Cassandra instance it can thwarted based on the cluster name. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Cassandra instance. An administrator uses a command line tool or a browser to connect to a Cassandra node and issue a membership change to join or leave the cluster.

5.5 Scaling the Cluster

When a new node is added into the system, it gets assigned a token such that it can alleviate a heavily loaded node. This results in the new node splitting a range that some other node was previously responsible for. The Cassandra bootstrap algorithm is initiated from any other node in the system by an operator using either a command line utility

or the Cassandra web dashboard. The node giving up the data streams the data over to the new node using kernel-kernel copy techniques. Operational experience has shown that data can be transferred at the rate of 40 MB/sec from a single node. We are working on improving this by having multiple replicas take part in the bootstrap transfer thereby parallelizing the effort, similar to Bittorrent.

5.6 Local Persistence

The Cassandra system relies on the local file system for data persistence. The data is represented on disk using a format that lends itself to efficient data retrieval. Typical write operation involves a write into a commit log for durability and recoverability and an update into an in-memory data structure. The write into the in-memory data structure is performed only after a successful write into the commit log. We have a dedicated disk on each machine for the commit log since all writes into the commit log are sequential and so we can maximize disk throughput. When the in-memory data structure crosses a certain threshold, calculated based on data size and number of objects, it dumps itself to disk. This write is performed on one of many commodity disks that machines are equipped with. All writes are sequential to disk and also generate an index for efficient lookup based on row key. These indices are also persisted along with the data file. Over time many such files could exist on disk and a merge process runs in the background to collate the different files into one file. This process is very similar to the compaction process that happens in the Bigtable system.

A typical read operation first queries the in-memory data structure before looking into the files on disk. The files are looked at in the order of newest to oldest. When a disk lookup occurs we could be looking up a key in multiple files on disk. In order to prevent lookups into files that do not contain the key, a bloom filter, summarizing the keys in the file, is also stored in each data file and also kept in memory. This bloom filter is first consulted to check if the key being looked up does indeed exist in the given file. A key in a column family could have many columns. Some special indexing is required to retrieve columns which are further away from the key. In order to prevent scanning of every column on disk we maintain column indices which allow us to jump to the right chunk on disk for column retrieval. As the columns for a given key are being serialized and written out to disk we generate indices at every 256K chunk boundary. This boundary is configurable, but we have found 256K to work well for us in our production workloads.

5.7 Implementation Details

The Cassandra process on a single machine is primarily consists of the following abstractions: partitioning module, the cluster membership and failure detection module and the storage engine module. Each of these modules rely on an event driven substrate where the message processing pipeline and the task pipeline are split into multiple stages along the line of the SEDA[20] architecture. Each of these modules has been implemented from the ground up using Java. The cluster membership and failure detection module, is built on top of a network layer which uses non-blocking I/O. All system control messages rely on UDP based messaging while the application related messages for replication and request routing relies on TCP. The request routing modules are implemented using a certain state machine. When a read/write request arrives at any node in the cluster the state machine morphs through the following states (i) identify the node(s) that own the data for the key (ii) route the requests to the nodes and wait on the responses to arrive (iii) if the replies do not arrive within a configured timeout value fail the request and return to the client (iv) figure out the latest response based on timestamp (v) schedule a repair of the data at any replica if they do not have the latest piece of data. For sake of exposition we do not talk about failure scenarios here. The system can be configured to perform either synchronous or asynchronous writes. For certain systems that require high throughput we rely on asynchronous replication. Here the writes far exceed the reads that come into the system. During the synchronous case we wait for a quorum of responses before we return a result to the client.

In any journaled system there needs to exist a mechanism for purging commit log entries. In Cassandra we use a rolling a commit log where a new commit log is rolled out after an older one exceeds a particular, configurable, size. We have found that rolling commit logs after 128MB size seems to work very well in our production workloads. Every commit log has a header which is basically a bit vector whose size is fixed and typically more than the number of column families that a particular system will ever handle. In our implementation we have an in-memory data structure and a data file that is generated per column family. Every time the in-memory data structure for a particular column family is dumped to disk we set its bit in the commit log stating that this column family has been successfully persisted to disk. This is an indication that this piece of information is already committed. These bit vectors are per commit log and also maintained in memory. Every time a commit log is rolled its bit vector and all the bit vectors of commit logs rolled prior to it are checked. If it is deemed that all the data has been successfully persisted to disk then these commit logs are deleted. The write operation into the commit log can either be in normal mode or in *fast sync* mode. In the fast sync mode the writes to the commit log are buffered. This implies that there is a potential of data loss on machine crash. In this mode we also dump the in-memory data structure to disk in a buffered fashion. Traditional databases are not designed to handle particularly high write throughput. Cassandra morphs all writes to disk into sequential writes thus maximizing disk write throughput. Since the files dumped to disk are never mutated no locks need to be taken while reading them. The server instance of Cassandra is practically lockless for read/write operations. Hence we do not need to deal with or handle the concurrency issues that exist in B-Tree based database implementations.

The Cassandra system indexes all data based on primary key. The data file on disk is broken down into a sequence of blocks. Each block contains at most 128 keys and is demarcated by a block index. The block index captures the relative offset of a key within the block and the size of its data. When an in-memory data structure is dumped to disk a block index is generated and their offsets written out to disk as indices. This index is also maintained in memory for fast access. A typical read operation always looks up data first in the in-memory data structure. If found the data is returned to the application since the in-memory data struc-

ture contains the latest data for any key. If not found then we perform disk I/O against all the data files on disk in reverse time order. Since we are always looking for the latest data we look into the latest file first and return if we find the data. Over time the number of data files will increase on disk. We perform a compaction process, very much like the Bigtable system, which merges multiple files into one; essentially merge sort on a bunch of sorted data files. The system will always compact files that are close to each other with respect to size i.e there will never be a situation where a 100GB file is compacted with a file which is less than 50GB. Periodically a major compaction process is run to compact all related data files into one big file. This compaction process is a disk I/O intensive operation. Many optimizations can be put in place to not affect in coming read requests.

6. PRACTICAL EXPERIENCES

In the process of designing, implementing and maintaining Cassandra we gained a lot of useful experience and learned numerous lessons. One very fundamental lesson learned was not to add any new feature without understanding the effects of its usage by applications. Most problematic scenarios do not stem from just node crashes and network partitions. We share just a few interesting scenarios here.

- Before launching the Inbox Search application we had to index 7TB of inbox data for over 100M users, then stored in our MySQL[1] infrastructure, and load it into the Cassandra system. The whole process involved running Map/Reduce[7] jobs against the MySQL data files, indexing them and then storing the reverse-index in Cassandra. The M/R process actually behaves as the client of Cassandra. We exposed some background channels for the M/R process to aggregate the reverse index per user and send over the serialized data over to the Cassandra instance, to avoid the serialization/deserialization overhead. This way the Cassandra instance is only bottlenecked by network bandwidth.

- Most applications only require atomic operation per key per replica. However there have been some applications that have asked for transactional mainly for the purpose of maintaining secondary indices. Most developers with years of development experience working with RDBMS's find this a very useful feature to have. We are working on a mechanism to expose such atomic operations.

- We experimented with various implementations of Failure Detectors such as the ones described in [15] and [5]. Our experience had been that the time to detect failures increased beyond an acceptable limit as the size of the cluster grew. In one particular experiment in a cluster of 100 nodes time to taken to detect a failed node was in the order of two minutes. This is practically unworkable in our environments. With the accrual failure detector with a slightly conservative value of PHI, set to 5, the average time to detect failures in the above experiment was about 15 seconds.

- Monitoring is not to be taken for granted. The Cassandra system is well integrated with Ganglia[12], a distributed performance monitoring tool. We expose various system level metrics to Ganglia and this has helped us understand the behavior of the system when subject to our production workload. Disks fail for no apparent reasons. The bootstrap algorithm has some hooks to repair nodes when disk fail. This is however an administrative operation.

- Although Cassandra is a completely decentralized system we have learned that having some amount of coordination is essential to making the implementation of some distributed features tractable. For example Cassandra is integrated with Zookeeper, which can be used for various coordination tasks in large scale distributed systems. We intend to use the Zookeeper abstraction for some key features which actually do not come in the way of applications that use Cassandra as the storage engine.

6.1 Facebook Inbox Search

For Inbox Search we maintain a per user index of all messages that have been exchanged between the sender and the recipients of the message. There are two kinds of search features that are enabled today (a) term search (b) interactions - given the name of a person return all messages that the user might have ever sent or received from that person. The schema consists of two column families. For query (a) the user id is the key and the words that make up the message become the super column. Individual message identifiers of the messages that contain the word become the columns within the super column. For query (b) again the user id is the key and the recipients id's are the super columns. For each of these super columns the individual message identifiers are the columns. In order to make the searches fast Cassandra provides certain hooks for intelligent caching of data. For instance when a user clicks into the search bar an asynchronous message is sent to the Cassandra cluster to prime the buffer cache with that user's index. This way when the actual search query is executed the search results are likely to already be in memory. The system currently stores about 50+TB of data on a 150 node cluster, which is spread out between east and west coast data centers. We show some production measured numbers for read performance.

Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

7. CONCLUSION

We have built, implemented, and operated a storage system providing scalability, high performance, and wide applicability. We have empirically demonstrated that Cassandra can support a very high update throughput while delivering low latency. Future works involves adding compression, ability to support atomicity across keys and secondary index support.

8. ACKNOWLEDGEMENTS

Cassandra system has benefitted greatly from feedback from many individuals within Facebook. In addition we thank Karthik Ranganathan who indexed all the existing data in

MySQL and moved it into Cassandra for our first production deployment. We would also like to thank Dan Dumitriu from EPFL for his valuable suggestions about [19] and [8].

9. REFERENCES

[1] MySQL AB. Mysql.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI*, pages 1–14, 2002.

[3] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205–218, 2006.

[5] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazonÕs highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM, 2007.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. The ϕ accrual failure detector. In *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology*, pages 66–78, 2004.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[10] Jim Gray and Pat Helland. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[11] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[12] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30:2004, 2004.

[13] Benjamin Reed and Flavio Junquieira. Zookeeper.

[14] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.

[15] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Service,Ť Proc. Conf. Middleware*, pages 55–70, 1996.

[16] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990.

[17] Ion Stoica, Robert Morris, David Liben-nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, 2003.

[18] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[19] Robbert van Renesse, Dan Mihai Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Large Scale Distributed Systems and Middleware Workshop (LADIS '08)*, New York, NY, USA, 2008. ACM.

[20] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.