ACMS: The Akamai Configuration Management System

Alex Sherman[†‡], *Philip A. Lisiecki*[†], *Andy Berkheimer*[†], *and Joel Wein*[†*].
[†]Akamai Technologies, Inc. [‡]Columbia University [*]Polytechnic University.
[†]{*andyb,lisiecki,asherman,jwein*}*@akamai.com*
[‡]*asherman@cs.columbia.edu* [*]*wein@mem.poly.edu.*

Abstract

An important trend in information technology is the use of increasingly large distributed systems to deploy increasingly complex and mission-critical applications. In order for these systems to achieve the ultimate goal of having similar ease-of-use properties as centralized systems they must allow fast, reliable, and lightweight management and synchronization of their configuration state. This goal poses numerous technical challenges in a truly Internet-scale system, including varying degrees of network connectivity, inevitable machine failures, and the need to distribute information globally in a fast and reliable fashion.

In this paper we discuss the design and implementation of a configuration management system for the Akamai Network. It allows reliable yet highly asynchronous delivery of configuration information, is significantly fault-tolerant, and can scale if necessary to hundreds of thousands of servers.

The system is fully functional today providing configuration management to over 15,000 servers deployed in 1200+ different networks in 60+ countries.

1 Introduction

Akamai Technologies operates a system of 15,000+ widely dispersed servers on which its customers deploy their web content and applications in order to increase the performance and reliability of their web sites. When a customer extends their web presence from their own server or server farm to a third party Content Delivery Network (CDN), a major concern is the ability to maintain close control over the manner in which their web content is served. Most customers require a level of control over their distributed presence that rivals that achievable in a centralized environment.

Akamai's customers can configure many options that determine how their content is served by the CDN. These options may include: html cache timeouts, whether to allow cookies, whether to store session data for their web applications among many other settings. Configuration files that capture these settings must be propagated quickly to all of the Akamai servers upon update.

In addition to the configuring customer profiles, Akamai also runs many internal services and processes which require frequent updates or "reconfigurations." One example is the mapping services which assign users to Akamai servers based on network conditions. Subsystems that measure frequently-changing network connectivity and latency must distribute their measurements to the mapping services.

In this paper we describe the Akamai Configuration Management System (ACMS), which was built to support customers' and internal services' configuration propagation requirements. ACMS accepts distributed submissions of configuration information (captured in configuration files) and disseminates this information to the Akamai CDN. ACMS is highly available through significant fault-tolerance, allows reliable yet highly asynchronous and consistent delivery of configuration information, provides persistent storage of configuration updates, and can scale if necessary to hundreds of thousands of servers.

The system is fully functional today providing configuration management to over 15,000 servers deployed in 1200+ different ISP networks in 60+ countries. Further, as a lightweight mechanism for making configuration changes, it has evolved into a critical element of how we administer our network in a flexible fashion.

Elements of ACMS bear resemblance to or draw from numerous previous efforts in distributed systems – from reliable messaging/multicast in wide-area systems, to fault-tolerant data replication techniques, to Microsoft's Windows Update functionality; we present a detailed comparison in Section 8. We believe, however, that our system is designed to work in a relatively unique environment, due to a combination of the following factors.

- The set of end clients – our 15,000+ servers – are very widely dispersed.

- At any point in time a nontrivial fraction of these servers may be down or may have nontrivial connectivity problems to the rest of the system. An individual server may be out of commission for several months before being returned to active duty, and will need to get caught up in a sane fashion.

- Configuration changes are generated from widely dispersed places – for certain applications, any server in the system can generate configuration information that needs to be dispersed via ACMS.

- We have relatively strong consistency requirements. When a server that has been out-of-touch regains contact it needs to become up to date quickly or risk serving customer content in an outdated mode.

Our solution is based on a small set of front-end distributed Storage Points and a back-end process that manages downloads from the front-end. We have designed and implemented a set of protocols that deal with our particular availability and consistency requirements.

The major contributions of this paper are as follows:

- We describe the design of a live working system that meets the requirements of configuration management in a very large distributed network.

- We present performance data and detail some lessons learned from a building and deploying such a system.

- We discuss in detail the distributed synchronization protocols we introduced to manage the front ends Storage Points. While these protocols bear similarity to several previous efforts, they are targeted at a different combination of reliability and availability requirements and thus may be of interest in other settings.

1.1 Assumptions and Requirements

We assume that the configuration files will vary in size from a few hundred bytes up to 100MB. Although very large configuration files are possible and do occur, they in general should be more rare. We assume that most updates must be distributed to every Akamai node, although some configuration files may have a relatively small number of subscribers. Since distinct applications submit configuration files dynamically, there is no particular arrival pattern of submissions, and at times we could expect several submissions per second. We also assume that the Akamai CDN will continue to grow. Such growth should not impede the CDN's responsiveness to configuration changes. We assume that submissions could originate from a number of distinct applications running at distinct locations on the Akamai CDN. We assume that each submission of a configuration file *foo* completely overwrites the earlier submitted version of *foo*. Thus, we do not need to store older versions of *foo*, but the system must correctly synchronize to the latest version. Finally, we assume that for each configuration file there is either a single writer or multiple idempotent (non-competing) writers.

Based on the motivation and assumptions described above we formulate the following requirements for ACMS:

High Fault-Tolerance and Availability. In order to support all applications that dynamically submit configuration updates, the system must operate 24x7 and experience virtually no downtime. The system must be able to tolerate a number of machine failures and network partitions, and still accept and deliver configuration updates. Thus, the system must have multiple "entry points" for accepting and storing configuration updates such that failure of any one of them will not halt the system. Furthermore, these "entry points" must be located in distinct ISP networks so as to guarantee availability even if one of these networks becomes partitioned from the rest of the Internet.

Efficiency and Scalability. The system must deliver updates efficiently to a network of the size of the Akamai CDN, and all parts of the system must scale effectively to any anticipated growth. Since updates, such as a customer's profile, directly effect how each Akamai node serves that customer's content, it is imperative that the servers synchronize relatively quickly with respect to the new updates. The system must guarantee that propagation of updates to all "alive" nodes takes place within a few minutes from submission. (Provided of course, that there is network connectivity to such "alive" or functioning nodes from some of our "entry points.").

Persistent Fault-Tolerant Storage. In a large network some machines will always be experiencing downtime due to power and network outages or process failures. Therefore, it is unlikely that a configuration update can be delivered synchronously to the entire CDN in the time of submission. Instead the system must be able to store the updates permanently and deliver them asynchronously to machines as they become available.

Correctness. Since configuration file updates can be submitted to any of the "entry points," it is possible that two updates for the same file *foo* arrive at different "entry points" simultaneously. We require that ACMS provide a unique ordering of all versions and that the system synchronize to the latest version for each configuration file. Since slight clock skews are possible among our machines, we relax this requirement and show that we allow a very limited, but bounded reordering. (See section 3.4.2).

Acceptance Guarantee. ACMS "accepts" a submis-

sion request only when the system has "agreed" on this version of the update. The agreement in ACMS is based on a "quorum" of "entry points." (The quorum used in ACMS is at the core of our architecture and is discussed in great detail throughout the paper). The agreement is necessary, because if the "entry point" that receives an update submission becomes cut off from the Internet it will not be able to propagate the update to the rest of the system. In essence, the *Acceptance Guarantee* stipulates that if a submission is accepted, a quorum has agreed to propagate the submission to the Akamai CDN.

Security. Configuration updates must be authenticated and encrypted so that ACMS cannot be spoofed nor updates read by any third parties. The techniques that we use to accomplish this are standard, and we do not discuss them further in this document.

1.2 Our Approach

We observe that the ACMS requirements fall into two sets. The first set of requirements deals with update management: highly available, fault-tolerant storage and correct ordering of accepted updates. The second set of requirements deals with delivery: efficient and secure propagation of updates. Instinctively we split the architecture of the system into two subsystems – the "front-end" and the "back-end" – that correspond to the two sets of requirements. The front-end consists of a small set (typically 5 machines) of Storage Points (or SPs). The SPs are deployed in distinct Tier-1 networks inside well-connected data centers. The SPs are responsible for accepting and storing configuration updates. The back-end is the entire Akamai CDN that subscribes to the updates and aids in the update delivery.

High availability and fault-tolerance come from the fact that the SPs constitute a fully decentralized subsystem. ACMS does not depend on any particular SP to coordinate the updates, such as a database master in a persistent MOM (message-oriented middleware) storage. ACMS can tolerate a number of failures or partitions among the Storage Points. Instead of relying on a coordinator, we use a set of distributed algorithms that help the SPs synchronize configuration submissions. These algorithms that will be discussed later are quorum-based and require only a majority of the SPs to stay alive and connected to one another in order for the system to continue operation. Any majority of the SPs can reconstruct the full state of the configuration submissions and continue to accept and deliver submissions.

To propagate updates, we considered a push-based vs. a pull-based approach. In a push-based approach the SPs would need to monitor and maintain state of all Akamai hosts that require updates. In a pull-based approach all Akamai machines check for new updates and request them. We observed that the Akamai CDN itself is fully optimized for HTTP download, making the pull-based approach over HTTP download a natural choice. Since many configuration updates must be delivered to virtually every Akamai server, this allows us to use Akamai caches effectively for common downloads and thus reduce network bandwidth requirements. This natural choice helps ACMS scale with the growing size of the Akamai network.

As an optimization we add an additional set of machines (the Download Points) to the front-end. Download Points offer additional sites for HTTP download and thus alleviate the bandwidth demand placed on the Storage Points.

To further improve the efficiency of the HTTP download we create an index hierarchy that concisely describes all configuration files available on the SPs. A downloading agent can start with downloading the root of the hierarchical index tree and work its way down to detect changes in any particular configuration files it is interested in.

The rest of this paper is organized as follows. We give an architecture overview in section 2. We discuss our distributed techniques of quorum-based replication and recovery in sections 3 and 4. Section 5 describes the delivery mechanism. We share our operational experience and evaluation in sections 6 and 7. Section 8 discusses related work. We conclude in section 9.

2 Architecture Overview

The architecture of ACMS is depicted in Figure 1.

First an application submitting an update (also known as a *publisher*) contacts an ACMS Storage Point. The publisher transmits a new version of a given configuration file. The SP that receives an update submission is also known as the *Accepting SP* for that submission. Before replying to the client the Accepting SP makes sure to replicate the message on at least a quorum (a majority) of Servers (i.e., Storage Points). Servers store the message persistently on disk as a file. In addition to copying the data, ACMS runs an algorithm called Vector Exchange that allows a quorum of SPs to *agree* on a submission. Only after the agreement is reached does the Accepting SP acknowledge the publisher's request, by replying with "Accept."

Once the agreement among the SPs is reached, the data can also be offered for download. The Storage Points *upload* the data to their local HTTP servers (i.e., HTTP servers runs on the same machines as the SPs).

Since only a quorum of SPs is required to reach an agreement on a submission, some SPs may miss an occasional update due to downtime. To account for replication messages missed due to downtime, the SPs run

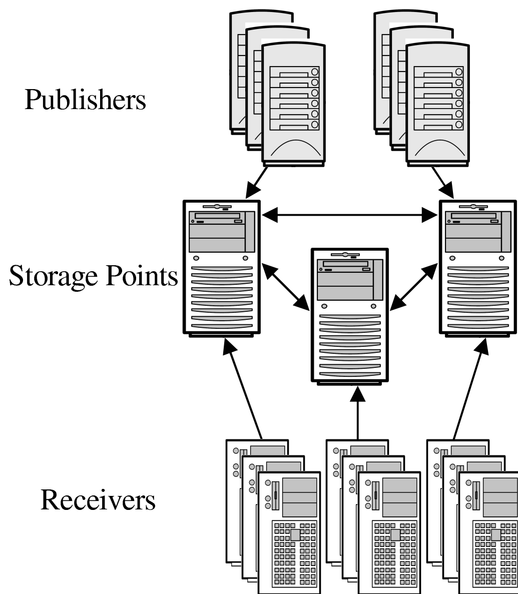

Figure 1: ACMS: Publishers, Storage Points, and Receivers (Subscribers)

a recovery scheme called *Index Merging*. Index Merging helps the Storage Points recover any missed updates from their peers.

To subscribe for configuration updates, each server (also known as a *node*) on the Akamai CDN runs a process called *Receiver* that coordinates subscriptions for that node. Services on each node subscribe with their local Receiver process to receive configuration updates. Receivers periodically make HTTP IMS (If-Modified-Since) requests for these files from the SPs. Receivers send these requests via the Akamai CDN, and most of the requests are served from nearby Akamai caches reducing network traffic requirements.

We add an additional set of a few well-positioned machines to the front-end, called the Download Points (DPs). DPs never participate in initial replication of updates and rely entirely on Index Merging to obtain the latest configuration files. DPs alleviate some of the download bandwidth requirements from the SPs. In this way data replication between the SPs does not need to compete as much for bandwidth with the download requests from subscribers.

3 Quorum-based Replication

The fault-tolerance of ACMS is based on the use of a simple quorum. In order for an Accepting SP to accept an update submission we require that the update be both replicated to and agreed upon by a quorum of the ACMS SPs. We define *quorum* as a majority. As long as a majority of the SPs remain functional and not partitioned from one another, this majority subset will intersect with the initial quorum that accepted a submission. Therefore, this latter subset will collectively contain the knowledge of all previously accepted updates.

This approach is deeply rooted in our assumption that ACMS can maintain a majority of operational and *connected* SPs. If there is no quorum of SPs that are functional and can communicate with one another ACMS will halt and refuse to accept new updates until a *connected quorum* of SPs is re-established.

Each SP maintains connectivity by exchanging liveness messages with its peers. Liveness messages also indicate whether the SPs are fully functional or *healthy*. Each SP reports whether it has pairwise connectivity to a quorum (including itself) of healthy SPs. The reports arrive at the Akamai NOCC (Network Operations Command Center) [2]. If a majority of ACMS SPs fails to report pairwise connectivity to a quorum, a red alert is generated in the NOCC and operation engineers perform immediate connectivity diagnosis and attempt to fix the network or server problem(s).

By placing SPs inside distinct ISP networks we reduce the probability of an outage that would disrupt a quorum of these machines. (See some statistics in section 6.) Since we require only a majority of SPs to be connected, it means we can tolerate a number of failures due to partitioning, hardware, or software malfunctions. For example, with an initial set containing five SPs, we can tolerate two SP failures or partitions and still maintain a viable majority of three SPs. When any single SP malfunctions, a lesser priority alert also triggers corrective action from the NOCC engineers. ACMS operational experience with maintaining a connected quorum and various failure cases are discussed in detail in section 6.

The rest of the section describes the quorum-based Acceptance Algorithm in detail. We also explain how ACMS replication and agreement methods satisfy *Correctness* and *Acceptance* requirements outlined in section 1.1 and discuss maintenance of the ACMS SPs.

3.1 Acceptance Algorithm

The ACMS Acceptance Algorithm consists of two phases: replication and agreement. In the replication phase, the Accepting SP copies the update to at least a quorum of the SPs.

The Accepting SP first creates a temporary file with a unique filename (UID). For a configuration file *foo* the UID may look like this: “foo.A.1234”, where *A* is the name of the Accepting SP and “1234” is the timestamp of the request in UTC (shortened to 4 digits for this example). This UID is unique, because each SP allows only

one request per file per second.

The Accepting SP then sends this file that contains the update along with its MD5 hash to a number of SPs over a secure TCP connection. Each SP that receives the file stores it persistently on disk (under the UID name), verifies the hash, and acknowledges that it has stored the file.

If the Accepting SP fails to replicate the data to a quorum after a timeout, it replies with an error to the publishing application. The timeout is based on the size of the update, and a very low estimate of available bandwidth between this SP and its peers. (If the Accepting SP does not have connectivity to a quorum it replies much sooner and does not wait for a timeout to expire).

Otherwise, once at least a quorum of SPs (including the Accepting SP) has stored the temporary file, the Accepting SP initiates the second phase to obtain an agreement from the Storage Points on the submitted update.

3.2 Vector Exchange

Vector Exchange (also called "VE") is a light-weight protocol that forms the second phase of the acceptance algorithm – the *agreement* phase. As the name suggests, VE involves Storage Points exchanging a state vector. The VE vector is just a bit vector with a bit corresponding to each Storage Point. A 1-bit indicates that the corresponding Storage Point *knows* of a given update. When a majority of bits are set to 1, we say that an *agreement* occurs and it is safe for any SP (that sees the majority of the bits set) to upload this latest update.

In the beginning of the agreement phase, the Accepting SP initializes a bit vector by setting its own bit to 1 and the rest to 0, and broadcasts the vector along with the UID of the update to the other SPs. Any SP that sees the vector sets its corresponding bit to 1, stores the vector persistently on disk and re-broadcasts the modified vector to the rest of the SPs. Persistent storage guarantees that the SP will not lose its vector state on process restart or machine reboot. It is safe for each SP to set the bit even if it did not receive the temporary file during the replication phase. Since at least a quorum of the SPs have stored this temporary file, it can always locate this file at a later stage.

Each SP learns of the agreement independently when it sees a quorum of bits set. Two actions can take place when a SP learns of the agreement for the first time. When the Accepting SP that initiated the VE instance learns of the agreement it *accepts* the submission of the publishing application. When any SP (including the Accepting SP) learns of the agreement it *uploads* the file. Uploading means that the SP copies the temporary file to a permanent location on its local HTTP server where it is now available for download by the Receivers. If it does not have the temporary file then it downloads it from one of the other SPs via the recovery routine (section 4).

Note, that it is possible for the Accepting SP to become "cut-off" from the quorum after it initiates the VE phase. In this case it does not know whether its broadcasts were received and whether the agreement took place. It is then forced to reply only with "Possible Accept" rather than "Accept" to the publishing application. We recommend that the publisher that gets cut off from the Accepting SP or receives a "Possible Accept" should try to re-submit its update to another SP. (From a publisher's perspective the reply of "Possible Accept" is equivalent to "Reject." The distinction was made initially purely for the purpose of monitoring this condition.)

As in many *agreement* schemes, the purpose of the VE protocol is to deal with some Byzantine network or machine failures [18]. In particular, VE prevents an individual SP (or a minority subset of SPs) from uploading new data and then becoming "disconnected" from the rest of the SPs. A quorum of SPs could then continue to operate successfully without the knowledge that the minority is advertising a new update. This new update would become available only to a small subset of the Akamai nodes that can reach the minority subset, possibly causing a discord in the Akamai network viz. the latest updates.

VE is based on earlier ideas of vector clocks introduced by by Fidge [10] and Mattern [24]. Section 8 compares Acceptance Algorithm with Two-Phase Commit and other agreement schemes used in common distributed systems.

3.3 An Example

We give an example to demonstrate both phases of the Acceptance Algorithm. Imagine that our system contains five Storage Points named *A*, *B*, *C*, *D*, and *E* with SP *D* down temporarily for a software upgrade. With five SPs the quorum required for the Acceptance algorithm is three SPs.

SP *A* receives a submission update from publisher *P* for configuration file "foo". To use the example from section 3.1 SP *A* stores the file under a temporary UID: *foo.A.1234*.

SP *A* initiates the replication phase by sending the file in parallel to as many SPs as it can reach. SPs *B*, *C*, and *E* store the temporary update under the UID name. (SP D is down and does not respond). SPs *B* and *C* happen to be the first SPs to acknowledge the reception of the file and the MD5 hash check. Now *A* knows that the majority (*A*, *B*, and *C*) have stored the file and *A* is ready to initiate the agreement phase.

SP *A* broadcasts the following VE message to the other SPs:

```
foo.A.1234 A:1 B:0 C:0 D:0 E:0
```

This message contains the UID of the pending update and the vector that has only *A*'s bit set. (*A* stores this vector state persistently on disk prior to sending it out).

When SP *B* receives this message it adds its bit to the vector, stores the vector, and broadcasts it:

```
foo.A.1234 A:1 B:1 C:0 D:0 E:0
```

After a couple of rounds all four live SPs store the following message with all bits set except for *D*'s:

```
foo.A.1234 A:1 B:1 C:1 D:0 E:1
```

At this point, as each SP sees that the majority of bits is set, *A*, *B*, *C*, and *E* upload the temporary file in place of the permanent configuration file foo, and store in their local database the UID of the latest agreed upon version of file foo: *foo.A.1234*. All older records of foo can be discarded.

3.4 Guarantees

We now show that our Acceptance Algorithm satisfies the acceptance and correctness requirements, provided that our quorum assumption holds.

3.4.1 Acceptance Guarantee

Having introduced the quorum-based scheme we now restate the acceptance guarantee more precisely than in section 1.1. The acceptance guarantee states that if the Accepting SP has accepted a submission, it will be uploaded by a quorum of SPs.

Proof: The Accepting SP accepts only when the update has been replicated to a quorum AND when the Accepting SP can see a majority of bits set in the VE vector. Now if the Accepting SP can see a majority of bits set in the VE vector it means that at least a majority of the SPs have stored a partially filled VE vector during the agreement phase. Therefore, any future quorum will include at least one SP that stores the VE vector for this update. Once such a SP is part of a quorum, after a few re-broadcast rounds, all of the SPs in this future quorum will have their bits set. Therefore, all the SPs in the latter quorum will decide to upload.

So based on our assumption that a quorum of connected SPs can be reasonably maintained, acceptance by ACMS implies a future decision by at least a quorum to upload the update.

The converse of the acceptance guarantee does not necessarily hold. If the quorum decides to upload, it does not mean that the Accepting SP will accept. As stated earlier the Accepting SP may be "cut off" from the quorum after VE phase is initiated, but before it completes. In that case the Accepting SP replies with "Possible Accept," because it's likely but not definite. The publishing application treats this reply as "Reject" and tries to resubmit to another SP.

The probability of a "Possible Accept" is very small, and we have never seen it occur in the real system. The reason for that is that in order for the VE phase to be initiated the replication phase must succeed. If the replication is successful it most likely means that the lighter VE phase that also requires connectivity to a quorum (but less bandwidth) will also succeed. If the replication phase fails, ACMS replies with a definite "Reject."

3.4.2 Correctness

The Correctness requirements state that ACMS provides a unique ordering of all update versions for a given configuration file AND that the system synchronizes to the latest submitted update. We later relaxed that guarantee to state that ACMS allows *limited* re-ordering in deciding which update is the latest, due to clock skews. More precisely, accepted updates for the same file submitted at least $2T + 1$ seconds apart will be ordered correctly. T is the maximum allowed clock skew between any two communicating SPs.

The unique ordering of submitted updates is guaranteed by the UID assigned to a submission as soon as it is received by ACMS (regardless of whether it will be accepted). The UID contains both a UTC timestamp from the SP's clock and the SP's name. The submissions for the same configuration file are first ordered by time and then by the Accepting SP name. So "foo.B.1234" is considered to be more recent than "foo.A.1234", and it is kept as the later version. A Storage Point accepts only one update per second for a given configuration file.

Since we do not use logical synchronized clocks, slight clock skews and reordering of updates are possible. We now explain how we bound such reordering, and why any small reordering is acceptable in ACMS.

We bound the possible skew between any two communicating SPs by T seconds (where T is usually set to 20 seconds). Our communication protocols enforce this bound by rejecting liveness messages from SPs that are at least T seconds apart. (I.e., such pairs of servers appear virtually dead to each other). As a result it follows that no two SPs that accept updates for the same file can have a clock skew more than $2T$ seconds.

Proof: Imagine SPs *A* and *B* that are both able to accept updates. This means both *A* and *B* are able to replicate these update to a majority of SPs. These majorities must overlap by at least one SP. Moreover, neither *A* nor *B* can have more than a T second clock skew from that SP. So *A* and *B* cannot be more than $2T$ seconds apart.

Developers of the Akamai subsystems that submit configuration files to Akamai nodes via ACMS are advised to avoid mis-ordering by submitting updates to the

same configuration file at intervals of at least $2T + 1$. In addition, we use NTP [3] to synchronize our server clocks, and in practice we find very rare instances when our servers are more than one second apart.

Finally with ACMS, it is actually acceptable to reorder updates within a small bound such as $2T$. We are not dealing with competing editors of a distributed filesystem. Subsystems that are involved in configuring a large CDN such as Akamai must and do cooperate with each other. In fact, we considered two cases of such subsystems that update the same configuration file. Either there is only one process that submits updates for file "foo", or there are redundant processes that submit the same or idempotent updates for file "foo". In the case of a single publishing process, it can easily abide by the $2T$ rule and therefore avoid reordering. In the case of redundant writers – that exist for fault-tolerance – we do not care whose update within the $2T$ period is submitted first as these updates are idempotent. Any more complex distributed systems that publish to ACMS use leader election to select a publishing process, effectively reducing these systems to one-publisher systems.

3.5 Termination and Message Complexity

In almost all cases VE terminates after the last SP in a quorum broadcasts its addition to the VE vector. However, in an unlikely event where a SP becomes partitioned off during a VE phase it attempts to broadcast its vector state once every few seconds. This way, once it reconnects to a quorum it can notify the other SPs of its partial state.

VE is not expensive and the number of messages exchanged is quite small. We make a small change to the protocol as it was originally described by adding a small random delay (under 1 second) before a re-broadcast of the changed vector by a SP. This way, instead of all SPs re-broadcasting in parallel, only one SP broadcasts at a time. With the random delay, on average each SP will only broadcast once after setting its bit. This results in $O(n^2)$ unicast messages.

We use the gossip model, because the numbers of participants and the size of the messages are both small. The protocol can easily be amended to have only the Accepting SP do a re-broadcast after it collects the replies. Only when an SP does not hear the re-broadcast does it switch to a gossip mode. When the Accepting SP stays connected until termination the number of messages exchanged is just $O(n)$.

3.6 Maintenance

Software or OS upgrades performed on individual Storage Points must be coordinated to prevent an outage of a quorum. Such upgrades are scheduled independently on individual Storage Points so that the remaining system still contains a connected quorum.

Adding and removing machines with quorum-based systems is a theoretically tricky problem. Rambo [19] is an example of a quorum-based system that solves dynamic set configuration changes by having an old quorum agree on a new configuration.

Since adding or removing SPs is extremely rare we chose not to complicate the system to allow dynamic configuration changes. Instead, we halt the system temporarily by disallowing accepts of new updates, change the set configuration on all machines, wait for a new quorum to sync up on all state (via the Recovery algorithm), and allow all SPs to resume operation. Replacing a dead SP is a simpler procedure where we bring up a new SP with the same SP ID as the old one and clean state.

3.7 Flexibility of the VE Quorum

ACMS' quorum is configured as *majority*. Just like in the Paxos [16] algorithm this choice guarantees that any future quorum will necessarily intersect with an earlier one and all previously accepted submissions can be recovered. However, this definition is quite flexible in VE and allows for consistency vs. availability trade-offs. For example, one could define a quorum to be just a couple of SPs which would offer loose consistency, but much higher availability. Since there is a new VE instance for each submission, one could potentially configure a different quorum for each file. If desired, this property can be used to add or remove SPs by reconfiguring each SP independently, resulting in a very slight and temporary shift toward consistency over availability.

4 Recovery via Index Merging

Recovery is an important mechanism that allows all Storage Points that experience down time or a network outage to "sync up" all latest configuration updates.

Our Acceptance Algorithm guarantees that at least a quorum of SPs stores each update. Some Akamai nodes may only be able to reach a subset of the SPs that were not part of the quorum that stored the update. Even if that subset intersects with the quorum, that Akamai node may need to retry multiple downloads before reaching a SP that stores the update. To increase the number of Akamai nodes that can get their updates and improve the efficiency of download, preferably all SPs should store all state.

In order to "sync up" any missed updates Storage Points continuously run a background recovery protocol with one another. The downloadable configuration files are represented on the SPs in the form of an index tree.

The recovery protocol is called Index Merging. The SPs "merge" their index trees to pick up any missed updates from one another.

The Download Points also need to "sync up" state. These machines do not participate in the Acceptance Algorithm and instead rely entirely on the recovery protocol on Storage Points to pick up all state.

4.1 The Index Tree

For a concise representation of the configuration files, we organize the files into a tree. The configuration files are split into groups. A Group Index file lists the UIDs of the latest agreed upon updates for each file in the group. The Root Index file lists all Group Index files together with the latest modification timestamps of those indexes. The top two layers (i.e. the Root and the Group indexes) completely describe the latest UIDs of all configuration files and together are known as the *snapshot* of the SP.

Each SP can modify its *snapshot* when it learns of a quorum agreement through the Acceptance Algorithm or by seeing a more recent UID in a snapshot of another SP.

Since a quorum of SPs should together have a complete state, for full recovery each SP needs only to merge in a snapshot from $Q-1$ other SPs (where $Q = majority$). (Download Points need to merge in state from Q SPs).

The configuration files are assigned to groups statically when the new configuration file is provisioned on ACMS. A group usually contains a logical set of files subscribed to by a set of related receiving applications.

4.2 The Index Merging Algorithm

At each round of the Index Merging Algorithm a SP A picks a random set of $Q-1$ other SPs and downloads and parses the index files from those SPs. If it detects a more recent UID of a configuration file, SP A updates its own snapshot, and attempts to download the missing file from one of its peers. Note that it is safe for A to update its snapshot before obtaining the file. Since the UID is present in another SP's snapshot it means that the file has already been agreed upon and stored by a quorum.

To avoid frequent parsing of one another's index files, the SPs remember the timestamps of one another's index trees and make HTTP IMS (if-modified-since) requests. If an index file has not been changed, HTTP 304 (not-modified) is returned on the download attempt.

Index Merging rounds run continuously.

4.3 Snapshots for Receivers

As a side-effect the snapshots also provide an efficient way for Receivers to learn of latest configuration file versions. Typically receivers are only interested in a subset of the index tree that describes their subscriptions. Receivers also download index files from the SPs via HTTP IMS requests.

Using HTTP IMS is efficient but is also problematic because each SP generates its own snapshot and assigns its own timestamps to the index files that it uploads. Thus it is possible for a SP A to generate an index file with more recent timestamp than SP B, but less recent information. If a Receiver is unlucky and downloads the index file from A first, it will not download an index with a lower timestamp from B, until the timestamp increases. It may take a while for it to get all the necessary changes.

There are two solutions to this problem. In one solution we could require a Receiver to download an index tree independently from each SP, or at least a quorum of the SPs. Having each Receiver download multiple index trees is an unnecessary waste of bandwidth. Furthermore, requiring each Receiver to be able to reach a a quorum of SPs reduces system availability. Ideally, we only require that a Receiver be able to reach one SP that itself is part of a quorum.

We implemented an alternative solution, where the SPs merge their index timestamps, not just the data listed in the those indexes.

4.4 Index Time-stamping Rules

With just a couple of simple rules that constrain how Storage Points assign timestamps to their index files, we can present a coherent snapshot view to the Receivers:

1. If a Storage Point A has an index file bar.index with a timestamp T, and then A learns of new information inside bar.index (either through Vector Exchange agreement or Index Merging from a peer), then on the next iteration A must upload a new bar.index with a timestamp at least $T+1$.

2. If Storage Points A and B have an index file bar.index that contains identical information and have timestamps T_a and T_b respectively with $T_a > T_b$, then on the next iteration B must upload bar.index with a timestamp at least as great as T_a.

Simply put, rule 1 says that when a Storage Point includes new information it must increase the timestamp. This is really a redundant rule — a new timestamp would be assigned anyway when a Storage Point writes a new file. Rule 2 says that a Storage Point should always set its index's timestamp to the highest timestamp for that index among its peers (even if it includes no new information).

Once a Storage Point modifies a group index it must modify the Root Index as well following the same rules. (The same would apply to a hierarchy with more layers).

We now show the correctness of this timestamping algorithm.

4.5 Timestamping Correctness

Guarantee: If a Receiver downloads bar.index (index file for group bar) with a timestamp T_1 from any Storage Point, then when new information in group bar becomes available all Storage Points will publish bar.index with a timestamp at least as big as $T_1 + 1$, so that the Receiver will quickly pick up the change.

Proof: Assume in steady state a set of k Storage Points $1...k$ each has a bar.index with timestamps $T_1, T_2, ..., T_k$ sorted in non-decreasing order. (i.e., T_k is the highest timestamp). When new information becomes available, then following rule 1 above, Storage Point k will incorporate new information and increase its timestamp to at least $T_k + 1$. On the next iteration, following rule 2, SPs $1...k-1$ will make their timestamps at least $T_k + 1$ as well. Before the change, the highest timestamp for bar.index known to a Receiver was T_k. A couple of iterations after the new information becomes incorporated, the lowest timestamp available on any Storage Point is $T_k + 1$. Thus, a Receiver will be able to detect an increase in the timestamp and pick up a new index quickly.

5 Data Delivery

In addition to providing high fault-tolerance and availability the system must scale to support download by thousands of Akamai servers. We naturally use the Akamai CDN (Content Distribution Network) which is optimized for file download. In this section we describe the Receiver process, its use of the hierarchical index data, and the use of the Akamai CDN itself.

5.1 Receiver Process

Receivers run on each of over 15,000 Akamai nodes and check for message updates on behalf of the local subscribers.

A Subscription for a configuration file specifies the location of that file in the index tree: the root index, the group index that includes that file, and the file name itself. Receivers combine all local subscriptions into a subscriptions tree. (This is a subtree of the whole tree stored by the SPs.)

A Receiver checks for updates to the subscription tree by making HTTP IMS requests recursively beginning at the Root Index. If the Root Index has changed, Receiver parses the file, and checks whether any intermediate indexes that are also in the Receiver's subscription tree have been updated (i.e., if they are listed with a higher timestamp than previously downloaded by that Receiver). If so, it stores the timestamp listed for that index as the "target timestamp," and keeps making IMS requests until it downloads the index that is at least as recent as the target timestamp. Finally it parses that index and checks whether any files in its subscription tree (that belong to this index) have been updated. If so the Receiver then tries to download a changed file until it gets one at least as recent as the target timestamp.

There are a few reasons why a Receiver may need to attempt multiple IMS requests before it gets a file with a target timestamp. First some Storage Points may be a bit behind with Index Merging and not contain the latest files. Second, an old file may be cached by the Akamai network for a short while. The Receiver retries its downloads frequently until it gets the required file. Once the Receiver downloads the latest update for a subscription, it places the data in a file on local disk and points a local subscriber to it.

The Receiver must know how to find the SPs. The Domain Name Service provides a natural mechanism to distribute the list of SPs' and DPs' addresses.

5.2 Optimized Download

The Akamai network's support for HTTP download is a natural fit to be leveraged by ACMS for message propagation. Since the indexes and the configuration files are requested by many machines on the network, these files benefit greatly from the caching capabilities of the Akamai network.

First, Receivers running on colocated nodes are likely to request the same files, which makes it likely that the request is served from a neighboring cache in the local Akamai cluster. Furthermore, if the request leaves the cluster it will be directed to other nearby Akamai clusters which are also likely to have a response cached. Finally, if the file is not cached in another nearby Akamai cluster, the request goes through to one of the Storage Points. These cascading Akamai caches greatly reduce the network bandwidth required for message distribution and make pull-down propagation the ideal choice.

The trade-off of having great cacheability is the increased propagation delay of the messages. The longer the file is served out of cache, the longer it takes for the Akamai system to refresh cached copies. Since we are more concerned here with efficient rather than very fast delivery, we set a long cache TTL on the ACMS files, for example, 30 seconds.

As mentioned in section 2 we augment the list of SPs with a set of a few Download Points. Download Points provide an elegant way to alleviate bandwidth requirements from the SPs. As a result replication and recovery algorithms on the SPs experience less competition with the download bandwidth.

6 Operational Experience

The design of ACMS has been an iterative process between implementation and field experience where our assumptions of persistent storage, network connectivity, and OS/software fault-tolerance were tested.

6.1 Earlier Front-End Versions

Our prototype version of ACMS consisted of a single primary Accepting Storage Point replicating submissions to a few secondary Storage Points. Whenever the Accepting SP would lose connectivity to some of the Storage Points or experience a software or hardware malfunction the entire system would halt. It quickly became imperative to design a system that did not rely entirely on any single machine. We also considered a solution of using a set of auto-replicating databases. We encountered two problems. First, commercial databases would prove unnecessarily expensive as we would have to acquire licenses to match the number of customers using ACMS. More importantly, we required consistency. At the time we did not find database software that would deal with various Byzantine network failures. Although some academic systems were emerging that in theory did promise the right level of wide-area fault-tolerance we required a professional, field-tested system that we could easily tune to our needs. Based on our study of Paxos [16] and BFS [17] we designed a simpler version of decentralized quorum-based techniques. Similar to Paxos and BFS our algorithm requires a quorum. However, there is no leader to enforce strict ordering in VE as bounded re-ordering is permitted with non-competing configuration applications.

6.2 Persistent Storage Assumption

Storage Points rely on persistent disk storage to store configuration files, snapshots, and temporary VE vectors. Most hard disks are highly reliable, but guarantees are not absolute. Data may get corrupted, especially on systems with high levels of I/O. Moreover, if the operating system crashes before an OS buffer is flushed to disk, the result of the write may be lost.

After experiencing a few file corruptions we adopted the technique of writing out MD5 hash together with the file's contents before declaring a successful write. The hash is checked on opening the file. A Storage Point which detects a corrupted file will refuse to communicate with its peers and require an engineer's attention. Over the period of six months ending in February 2005, the NOCC [2] monitoring system has recorded 3 instances of such file corruption on ACMS.

Since ACMS runs automatic recovery routines replacing damaged or old hardware on ACMS is trivial. The SP process running on a clean disk quickly recovers all of the ACMS state from other SPs via Index Merging.

6.3 Connected Quorum Assumption

The assumption of a connected quorum turned out to be a very good one. Nonetheless, network partitions do occur, and the quorum requirement of our system does play its role. For the first 9 months of 2004 the NOCC monitoring system recorded 36 instances where a Storage Point did not have connectivity to a quorum due to network outages that lasted for more than 10 minutes. However, in all of those instances there was an operating quorum of other SPs that continued to accept submissions.

Brief network outages on the Internet are also common although they would generally not result in a SP losing connectivity to a quorum. For example, a closer analysis of ACMS logs over a 6 day period revealed two short outages within the same hour between a pair of SPs located in different Tier-1 networks. They lasted for 8 and 2 minutes respectively. Such outages emphasize the necessity for an ACMS-like design to provide uninterrupted service.

6.4 Lessons Learned

As we anticipated, redundancy has been important in all aspects of our system. Placing the SPs in distinct networks has protected ACMS from individual network failures. Redundancy of multiple replicas helped ACMS cope with disk corruption and data loss on individual SPs.

Even the protocols used by ACMS are in some sense redundant. The continuous recovery scheme (i.e., Index Merging) helps the Storage Points recover updates that they may miss during the initial replication and agreement phases of the Acceptance Algorithm. In fact, in some initial deployments Index Merging helped ACMS overcome some communication software glitches of the Acceptance Algorithm.

The back-end of ACMS also benefited from redundancy. Receivers begin their download attempt from nearby Akamai nodes, but can fail over to higher layers of the Akamai network if needed. This approach allows Receivers to cope with downed servers on their download path.

Despite the redundant and self-healing design sometimes human intervention is required. We rely heavily on the Akamai error reporting infrastructure and the operations of the NOCC to prevent critical failures of ACMS. Detection of and response to secondary failures such as

individual SP corruption or downtime helps decrease the probability of full quorum failures.

7 Evaluation

To evaluate the effectiveness of the system we gathered data from the live ACMS system accepting and delivering configuration updates on the actual Akamai network.

7.1 Submission and Propagation

First we looked at the workload of the ACMS front-end over a 48 hour period in the middle of a work week. There were 14,276 total file submissions on the system with five operating Storage Points. The table below lists the distribution of the file sizes. Submission of smaller files (under 100KB) were dominant, but files on the order of 50MB also appear about 3% of the time.

size range	avg file sz	distribution	avg.time (s)
0K-1K	290	40%	0.61
1K-10K	3K	26%	0.63
10K-100K	22K	23%	0.72
100K-1M	167K	7%	2.23
1M-10M	1.8M	1%	13.63
10M-100M	51M	3%	199.87

The last column of the table shows the average submission time for various file sizes. We evaluated the "submission" time by measuring the period from the time an Accepting SP is first contacted by a publishing application, until it replies with "Accept." The submission time includes replication and agreement phases of the Acceptance Algorithm. The agreement phase for all files takes 50 milliseconds on average. For files under 100KB, all "submission" times are under one second. However, with larger files, replication begins to dominate. For example, for 50MB files, the time is around 200 seconds. Even though our SPs are located in Tier 1 networks they all share replication bandwidth with the download bandwidth from the Receivers. In addition, replication for multiple submissions and multiple peers is performed in parallel.

We also measured the total update propagation time from when many configuration updates were first made available for download through receipt on the live Akamai network for a random sampling of 250 Akamai nodes. Figure 2 shows the distribution of update propagation times. The average propagation time is approximately 55 seconds. Most of the delay comes from Receiver polling intervals and caching.

Figure 3 examines the effect of file size on propagation time. We have analyzed the mean and 95th-percentile delivery time for each submission in the test period. 99.95% of updates arrived within three minutes. The remaining 0.05% were delayed due to temporary network connectivity issues; the files were delivered promptly after connectivity was restored. These delivery times meet our objectives of distributing files within several minutes. The figure shows a high propagation time for especially small files. Although one would expect that the propagation time increases monotonically with the file size, CDN caching slows down files submitted more frequently. We believe that many smaller files are updated frequently on ACMS. As a result the caching TTL of the CDN is more heavily reflected in propagation delay.

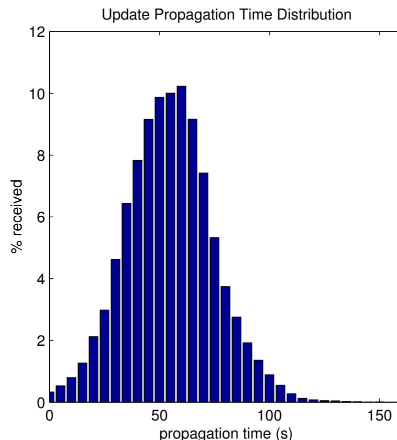

Figure 2: Propagation time distribution for a large number of configuration updates delivered to a sampling of thousands of machines.

The use of caching reduces bandwidth on the Storage Points anywhere from 90% to 99%, increasing in general with system activity and with the file size being pushed, allowing large updates to be propagated to tens of thousands of machines without significant impact on Storage Point traffic.

Finally to analyze general connectivity and the tail of the propagation distribution we looked at a propagation of short files (under 20KB) to another random sample of 300 machines over a 4 day period. We found that 99.8% of the time a file was received within 2 minutes from becoming available and 99.96% of the time it was received within 4 minutes.

7.2 Scalability

We analyzed the overhead of the Acceptance Algorithm and its effect on the scalability of the front-end. Over a recent 6 day period we recorded 43,504 successful file submissions with an average file size of 121KB. In a system with 5 SPs, the Accepting SP needs to replicate data to 4 other SPs requiring 484 KBytes per file on average. The size of a VE message is roughly 100 bytes. With $n(n-1)$ VE messages exchanged per submission, VE uses 2 KB per file or 0.4% of the replication bandwidth.

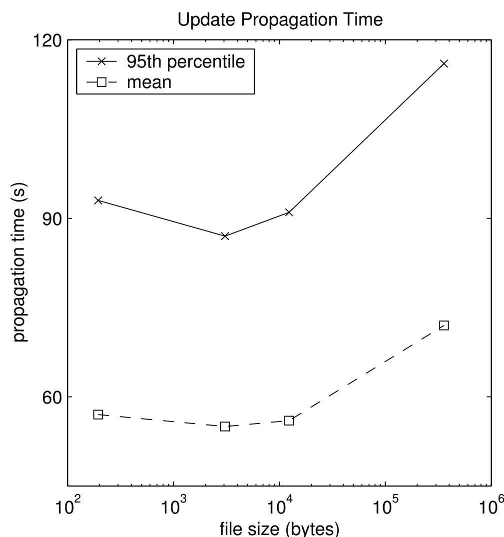

Figure 3: Propagation times for various size files. The dashed line shows the average time for each file to propagate to 95% of its recipients. The solid line shows the average propagation time.

For our purposes we chose 5 SPs, so that during a software upgrade of one machine the system cold tolerate one failure and still maintain a majority quorum of 3. Extending the calculation to 15 SPs, for example, with an average file size of 121 KB the system would require 1.7 MB for replication and 21KB for VE. The VE overhead becomes 1.2%, which is higher, but not significant.

Such a system is conceivable if one chooses not to rely on a CDN for efficient propagation, but instead offer more download sites (SPs). The VE overhead can be further reduced as described in section 3.5. However, the minimum bandwidth required to replicate the data to all 15 machines may grow to be prohibitive. In such a system one could still allow each Server to maintain all indexes, but split the actual storage into subsets based on some hashing function such as Consistent Hashing [4].

For ACMS choosing the Akamai CDN itself for propagation is the natural choice. The cacheability of the system grows as the CDN penetrates more ISP networks, and the system scales naturally with its own growth. Also, as the CDN grows the reachability of receivers inside more remote ISPs improves.

8 Related Work

8.1 Fault Tolerant Replication

Many distributed filesystems such as Coda [20], Pangea [21], and Bayou [22] store files across multiple replicas similar to the Storage Points of ACMS. Similar to ACMS' Index Merging these filesystems run recovery algorithms that synchronize the data among replicas, such as Bayou's anti-entropy algorithm. However, all of these systems attempt to improve the availability of data at the expense of consistency. The aim is to allow file operations to clients on a set of disconnected machines. ACMS, on the other hand must provide a very high level of consistency across the Akamai network and cannot allow a single SP to accept and upload a new update independently.

The two-phase Acceptance Algorithm used by ACMS is similar in nature to the Two Phase-Commit [12]. Two-phase commit also separates a transaction phase from a commit phase, but its failure modes make it more suitable to a local environment.

The Vector Exchange (the agreement phase of our algorithm) was inspired by the concept of vector clocks introduced by Fidge [10] and Mattern [24] which are used to determine causality of events in a distributed system. Bayou also uses vectors to represent latest known commit sequence numbers for each server. In our algorithm, the vectors' contents are simply bits since each message only has two interesting states, known to a server or not. Each subsequent agreement is a separate "instance" of the protocol.

VE uses a quorum-based scheme similar to Paxos [16] and BFS [17]. Paxos defines quorum as strict majority while BFS defines it as "more than $2/3$." VE allows "quorum" to be configurable as long as it is at least a majority. All these algorithms consider Byzantine failures and rely on persistent storage by a quorum to enable a later quorum to recover state. This strong property precludes scenarios allowed by a simpler two phase commit protocol for a minority of partitioned replicas to commit a transaction. Other quorum systems include weighted voting [11] and hierarchical quorum consensus [15].

At the same time VE is simpler than Paxos and BFS and does not implement a full Byzantine Fault-Tolerance. It does not require an auxiliary protocol to determine a leader or a *primary* as in Paxos or BFS respectively. This relaxation stems from the nature of ACMS applications where only a single or redundant writers exist for each file and thus, some bounded reordering is permissible as explained in section 3.4.2. No leader is enforcing ordering.

OceanStore [31] is an example of a storage system that implements Byzantine Fault-Tolerance to have replicas agree on the order of updates that originate from different sources. ACMS, on the other hand must complete "agreement" at the time of an update submission. This is primarily due to the important aspect of the Akamai network where an application that publishes a new configuration file must know that the system has agreed to upload and propagate the new update. (Otherwise it will

keep retrying.)

8.2 Data Propagation

Similar to multicast [9], ACMS is designed to deliver data to many widely dispersed nodes in a way that conserves bandwidth. While ACMS takes advantage of the Akamai Network optimizations for hierarchical file caching, multicast uses proximity of network IP addresses to send fewer IP packets. However, due to the lack of more intelligent routing infrastructure between major networks on the Internet, it is virtually impossible to multicast data across these networks.

To bypass the Internet routing shortcomings many application-level multicast schemes based on overlay networks were proposed: CAN-Multicast [27], Bayeux [34], and Scribe [29] among others [14] [7]. These systems leverage communication topologies of P2P overlays such as CAN [26], Chord [30], Pastry [28], Tapestry [33]. Unlike ACMS, these systems create a propagation tree for each new source of the multicast, incurring an overhead. As shown in [5], using these systems for multicast is not always efficient. In our system on the other hand, once the data is injected into ACMS, it is available for download from any Storage or Download Point, and propagates down the tree from these distinct well-connected sources. The effect of the overlay networks used in reliable multicasting networks [23], [6] is replaced by cooperating caches in our system.

ACMS is similar to Messaging Oriented Middleware (MOM) in that it provides persistent storage and asynchronous delivery of updates to subscribers that may be temporarily unavailable. Common MOMs include Sun's JMS [32], IBM's MQSeries [13], Microsoft's MSMQ [25], and the like. These system usually contain a server that persists the messaging "queue" which helps deal with crash recovery, but does create a single point of failure. The distributed model of ACMS storage, on the other hand, helps it tolerate multiple failures or partitions.

8.3 Software Updates

Finally, we compare a complete ACMS with existing software update systems. LCFG [35] and Novadigm [36] create systems to manage desktops and PDAs across an enterprise. While these systems scale to thousands of servers they usually span a single or a few enterprise networks. ACMS, on the other hand delivers updates across multiple networks for critical customer-facing applications. As a result ACMS focuses on a highly fault-tolerant storage and efficient propagation.

Systems that deliver software, like Windows Updates [37] target a much larger set of machines than found in the Akamai network. However, polling intervals for such updates are not as critical. Some Windows users take days to activate their updates while each Akamai node is responsible for serving requests to tens of thousands of users and thus must synchronize to the latest updates very efficiently. Moreover, systems such as Windows Updates use a rigorous, centralized process to push out new updates. ACMS accepts submissions from dynamic publishers dispersed throughout the Akamai network. Thus, highly fault-tolerant, available, and consistent storage of updates is required.

9 Conclusion

In this paper we have presented the Akamai Configuration Management System that successfully manages configuration updates for the Akamai network of 15,000+ nodes. Through the use of simple quorum-based algorithms (Vector Exchange and Index Merging), ACMS provides highly available, distributed, and fault-tolerant management of configuration updates. Although these algorithms are based on earlier ideas, they were particularly adapted to suit a configuration publishing environment and provide high level of consistency and easy recovery for the ACMS' Storage Points. These schemes offer much flexibility and may be useful in other distributed systems.

Just like ACMS, any other management system could benefit from using a CDN such as Akamai's to propagate updates. First, a CDN managed by a third party offers a convenient overlay that can span thousands of networks effectively. A solution such as multicast requires much management and simply does not scale across different ISPs. Second, a CDN's caching and reach will allow the system to scale to hundreds of thousands of nodes and beyond.

Most importantly we have presented valuable lessons learned from our operational experience. Redundancy of machines, networks, and even algorithms helps a distributed system such as ACMS cope with network and machine failures, and even human errors. Despite 36 network failures that we recorded in the last 9 months, that affected some ACMS Storage Points, the system continued to operate successfully. Finally, active monitoring of any critical distributed system is invaluable. We relied heavily on the NOCC infrastructure to maintain a high level of fault-tolerance.

Acknowledgements

We would like to thank William Weihl, Chris Joerg, and John Dilley among many other Akamai engineers for their advice and suggestions during the design. We want to thank Gong Ke Shen for her role as a developer on this

project. We would like to thank Professor Jason Nieh for his motivation and advice with the paper. Finally, we want to thank all of the reviewers and especially our NSDI shepherd Jeff Mogul for their valuable comments.

References

[1] Akamai Technologies, Inc., *http://www.akamai.com/*.

[2] Network Operations Command Center, *http://www.akamai.com/en/html/technology/nocc.html*.

[3] *http://www.ntp.org/*.

[4] D.Karger, E.Lehman, T.Leighton, M.Levine, D.Lewin and R.Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.* In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663 , 1997.

[5] M. Castro, M. B. Jones, A-M Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, *An Evaluation of Scalable Application-level Multicast Built Using Peer-to-Peer Overlays*, in Proc. INFOCOM, 2003.

[6] Y. Chawathe, S. McCanne, and E. Brewer, *RMX: Reliable Multicast for Heterogeneous Networks*, Proc. of INFOCOM, March 2000, pp. 795–804.

[7] Y. H. Chu, S. G. Rao, and H. Zhang, *A case for end system multicast*, Proc. of ACM Sigmetrics, June 2000, pp. 1–12.

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen, *Consistency in partitioned networks*, ACM Comput. Surveys, 1985.

[9] S. E. Deering and D. R. Cheriton, *Host extensions for IP multicasting*, Technical Report RFC 1112, Network Working Group, August 1989.

[10] C. J. Fidge, *Timestamp in Message Passing Systems that Preserves Partial Ordering*, Proc. 11th Australian Computing Conf., 1988, pp. 56–66.

[11] D. K. Gifford, *Weighted Voting for Replicated Data*, Proceedings 7th ACM Symposium on Operating Systems, 1979.

[12] J. Gray, *Notes on database operating systems*, Operating Systems: An Advanced Course. pp. 394–481, 1978.

[13] IBM Corporation, *WebSphere MQ family*, *http://www-306.ibm.com/software/integration/mqfamily/*.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, F. Kaashoek, and J. W. O'Toole, *Overcast: Reliable Multicasting with an Overlay Network*, Proc. of OSDI, October 2000, pp. 197–212.

[15] A. Kumar, *Hierarchical quorum consensus: A new algorithm for managing replicated data*, IEEE Trans. Computers, 1991.

[16] L. Lamport, *The Part-time Parliament*, ACM Transactions in Computer Systems, May, 1998.

[17] M. Castro, B. Liskov, *Practical Byzantine Fault-Tolerance*, OSDI 1999.

[18] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, July 1982.

[19] N. Lynch and A. Shvartsman, *RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks*, DISC, October 2002.

[20] M. Satyanarayanan, *Scalable, Secure, and Highly Available Distributed File Access*, IEEE Computer, May 1990.

[21] S. Saito, C. Karamanolis, M. Karlsson, M. Mahalingam, *Taming Aggressive Replication in the Pangaea Wide-area File System*, OSDI 2002.

[22] K. Peterson, M. Spreitzer, D. Terry, *Flexible Update Propagation for Weakly Consistent Replication*, SOSP, 1997.

[23] S. Paul, K. Sabnani, J. C. Lin, and S. Bhattacharyya, *Reliable Multicast Transport Protocol (RMTP)*, IEEE Journal on Selected Areas in Communications, April 1997.

[24] F. Mattern, *Virtual Time and Global States of Distributed Systems*, Proc. Parallel and Distributed Algorithms Conf., Elsevier Science, 1988.

[25] Microsoft Corporation, *Microsoft Message Queuing (MSMQ) Center*, *http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp*.

[26] S. Rantmasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network*, Proc. of ACM SIGCOMM, August 2001.

[27] S. Ratnasamy, M. Handley, R. Karp and S. Shenker, *Application-level multicast using content-addressable networks*, Proc. of NGC, November 2001.

[28] A. Rowstron and P. Drischel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, Proc of Middleware, November 2001.

[29] A. Rowstron, A. M. Kermarrec, M. Castro and P. Druschel, *Scribe: The design of a large-scale event notification infra, structure* in Proc of NGC, Nov 2001.

[30] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proc of ACM SIGCOMM, August 2001.

[31] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, *OceanStore: an architecture for global-scale persistent storage*, ASPLOS 2000, November 2000.

[32] Sun Microsystems, *Java Message Service*, *http://java.sun.com/products/jms*.

[33] B. Zhao, J. Kubiatowicz and A. Joseph, *Tapestry: An infrastructure for fault-resilient wide-area location and routing*, U. C. Berkeley, Tech. Rep. April 2001.

[34] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, *Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination*, Proc. of NOSSDAV, June 2001.

[35] *http://www.lcfg.org/*.

[36] *http://www2.novadigm.com/hpworld/*.

[37] Microsoft Windows Update *http://windowsupdate.microsoft.com*.