Experience with some Principles for Building an Internet-Scale Reliable System

Mike Afergan
Akamai and MIT

Joel Wein
Akamai and Polytechnic University

Amy LaMeyer
Akamai

Abstract

We discuss the design methodology used to achieve commercial-quality reliability in the Akamai content delivery network. The network consists of 15,000+ servers in 1,100+ networks and spans 65+ countries. Despite the scale of the Akamai CDN and the unpredictable nature of the underlying Internet, we seek to build a system of extremely high reliability. We present some simple principles we use to assure high reliability, and illustrate their application. As there is some similarity-in-spirit between our implementation and recent trends in the research literature, we hope that sharing our experiences will be of value to a broad community.

1 Introduction

In this paper we discuss the design decisions made by Akamai Technologies to assure reliability. Akamai [1] provides distributed computing solutions and services, including a Content Delivery Network (CDN), which customers use to distribute their content to enhance the scalability, reliability, and/or performance of their web properties. To accomplish this, Akamai has built a network of 15,000+ servers located in 1,100+ third-party networks. On these servers we run custom software designed to ensure high reliability and performances in spite of the various failure modes of the underlying Internet.

In our context we define reliability (loosely) as the ability to deliver web content successfully to an end-user under all circumstances. The abstract goal of course is one hundred percent reliability. However, given the practical constraints of the Internet, a more practical benchmark is substantial improvement in customers' web properties reliability with Akamai as opposed to without.

Our approach to reliability is to *assume that a significant and constantly changing number of component or other failures occur at all times in the network.* Consequently, we design our software to tolerate such numerous and dynamic failures as part of the operational network. All builders of distributed systems must handle failures.

However, our decision to expect frequent and diverse failures, coupled with our widespread deployment of relatively small *regions* of machines (typically 2-18 machines), leads to a design space that is somewhat different from that of most large distributed systems. For example, in most commercial distributed systems the failure of a datacenter would be a major event, requiring immediate repair. In contrast, multiple datacenter failures in our network are not uncommon events and do not require immediate attention. In this paper, we briefly explain why we make this fundamental assumption, and then illustrate some simple design principles that are consequences of this assumption. We then demonstrate that these principles lead to a number of advantages for our software development and operations processes.

We note here what this short paper is *not.* It is not an explication of the entire Akamai architecture; we assume basic familiarity with CDN principles and Akamai functionality [6]. Nor is it an argument that the Akamai architecture is the best, nor, given the space constraints, a comparison to numerous important related efforts in distributed systems. Neither is this paper a detailed data-driven study of the reliability of our network or of the underlying components to justify our fundamental principle. Rather, this paper is born of both the sufficient maturity of our technology to enable us to identify some principles that we have applied throughout our system, and our realization that our approach resonates in many ways with recent academic activity.

One particularly relevant research effort is Recovery Oriented Computing (ROC) [5, 8]. A fundamental element of ROC's approach is component recovery, in the form of fail-stop and restart, to increase overall system reliability. In particular, "ROC emphasizes recovery from failures rather than failure-avoidance" [3]. While the approaches presented in this paper and the ROC effort developed independently, this paper can be viewed as the summary of a seven-year experiment in implementing a ROC system.

2 Our Philosophy

To begin, we first motivate and then describe our basic design philosophy.

2.1 Challenges of running an Internet-Scale System

Numerous failure modes present challenges to a would-be reliable Internet-scale system:

Path Failure: Various problems can degrade or destroy connectivity between any two endpoints in the Internet.

Machine Failure: Servers fail for a variety of reasons, from hardware failure to cable disconnects.

Region/Rack Failure: The cause here may be within Akamai's control, as when a switch fails. It may also be outside our control, such as a rack losing power, failure of an upstream router, or datacenter maintenance.

Multiple Rack/Data Center Failure: Multiple racks within a datacenter or even an entire datacenter can fail, for reasons ranging from operational error in the host network to physical problems including natural disasters.

Network Failure: ISP-internal failures can affect a subset of datacenters or even the core of a particular network.

Multi-Network/Internet-wide Failures: We continue to see issues including trans-oceanic cable cuts, Internet-scale worms (e.g., SQL-Slammer), peering problems [2], and BGP operator errors. The impact of these incidents can vary from significantly higher latency or packet loss to complete disconnection.

To better understand the nature of failures, we examine manual suspensions in our network. A manually suspended machine can communicate with the rest of the system but has been flagged to not serve traffic. The set of machines does not, for example, include servers in regions with transient packet loss—our mapping system handles those. Instead, suspended machines are those with long-term and/or habitual problems. We observe that this set has a daily churn rate of approximately 4%, supporting our assumption that components continually fail. This also implies the mean-time-to-recovery is approximately 25 days, though this distribution is heavy-tailed. For some insight into the nature of the failures, we note that approximately 40% of the suspended machines are in regions where the whole region is suspended. This is likely from a region or datacenter error.

2.2 Our Philosophy

Given the significant possibilities for failure throughout the network, we were led to the following assumption.

Our Assumption: *We assume that a significant and constantly changing number of component or other failures occur at all times in the network.*

This leads naturally to the following:

Our Development Philosophy: *Our software is designed to seamlessly work despite numerous failures as part of the operational network.*

In other words, we choose to embrace, not confront, these component failure modes. In particular, *we expect a fraction of our network to be down at all times and we design for a significant fraction of our network to be down for particular moments in time*. Losing multiple datacenters or numerous servers in a day is not unusual. While every designer of robust distributed systems expects failures and aims to provide reliability in spite of them, our philosophy, combined with our wide and diverse deployment makes our approach relatively unique among commercial distributed systems.

This philosophy pervasively informs our design. For example, we buy commodity hardware, not more reliable and expensive servers. Because the servers in a region share several potential points of failure, we build more smaller regions instead of fewer larger ones. We spread our regions among ISPs and among the datacenters within an ISP. We rely on software intelligence to leverage the heterogeneity and redundancy of our network. Finally, we communicate over the public Internet, even for internal communication. An alternative would be to purchase dedicated links; instead we augment the reliability of the public Internet with logic and software.

3 Three Design Principles

In this section we present and illustrate three practical principles that follow from our philosophy.

3.1 Principle #1: Ensure significant redundancy in all systems to facilitate failover

Realizing the principle in practice is challenging because of aspects of the Internet architecture, interactions with 3rd-party software, and cost. Because DNS plays a fundamental role in our system (it is used to direct end-users to Akamai servers), we highlight two challenges in achieving significant redundancy in DNS.

One problem is the *size constraints for DNS responses*. The Generic Top Level Domain (gTLD) servers are critical as they supply the answer to queries for `akamai.net`. However, the size of DNS responses limits the number of servers we can return to 13 [9]. This is a relatively small number and not something we can address directly. We take a number of steps to increase this redundancy, including using IP anycast.

Another problem is that *DNS TTLs* challenge reliability by fixing a resolution for a period of time. If the server fails, a user could fail to receive service until the TTL expired. We address this in two ways.

The first is a two-level DNS system to balance responsiveness with DNS latency. The top level directs the user's DNS resolver to a region with a TTL of 30 to 60 minutes for a particular domain (e.g., `g.akamai.net`).

In the typical case, this region then resolves the lower-level queries (e.g., `a1.g.akamai.net`). To prevent this single region from inducing user-appreciable failures, the top level returns (low-level) nameservers in multiple regions. This requires determining an appropriate secondary choice such that a) performance does not suffer but b) the chance of all nameservers failing simultaneously is low.

The second aspect of our approach is that, within each level, we have an appropriate failover mechanism. Because the top-level resolution is long, much of the actual mapping occurs in the low-level resolution, which has a TTL of only 20 seconds. This provides us with significant flexibility to facilitate intra-region reliability. To address failures during the 20 seconds, a live machine within the region will automatically ARP over the IP address of a down machine.

3.2 Principle #2: Use software logic to provide message reliability

The operation of Akamai's system entails the distribution of numerous messages and data within the network, including control messages, customer configurations, monitoring information, load reporting, and customer content.

One approach for these communications channels would be to build dedicated links—or contract for virtually dedicated links—between our datacenters. We did not choose this architecture because this would not scale to a large number of datacenters. Further, since even these architectures are not immune to failures, we still would have had to invest effort in building resiliency.

We have thus built two underlying systems—a real-time (UDP) network and a reliable (HTTP/TCP) transport network. Our real-time transport network which was first built in 1999 uses multi-path routing, and at times limited retransmission, to achieve a high level of reliability without sacrificing latency. (This system is discussed in detail in [7].) We have since leveraged this system for a variety of functions including most internal network status messages.

Motivated by our customers' increasing use of dynamic or completely uncacheable content, we also built an HTTP/TCP equivalent network in 2000 to distribute content from the origin to the edge. In contrast to the UDP network and RON [4], each file is transmitted as a separate HTTP request—though often in the same connection. This HTTP subsystem serves as the basis for our SureRoute product and is an important internal communication tool. The system explores a variety of potential paths and provides an HTTP-based tunnel for requests through intermediate Akamai regions.

3.3 Principle #3: Use distributed control for coordination

This is not a surprising principle but one which we include for completeness and because there are often interesting practical subtleties. At a high level, we employ two forms of decision making. The first and most simple is failover, for cases where the logic is simple and we must immediately replace the down component. A previously discussed example of this is the case where a machine will ARP over the IP address of a down machine. The second technique is leader election, where leadership evaluation can depend on many factors including machine status, connectivity to other machines in the network, or even a preference to run the leader in a region with additional monitoring capabilities.

4 Example: How Akamai Works Under Extreme Component Failure

In this section, we examine how Principles 1-3 are applied by considering a hypothetical failure scenario.

4.1 Basic Operation

The basic HTTP request to the Akamai network consists of two parts – a DNS lookup and a HTTP request. In this section we separate the two. The DNS lookup for `a123.g.akamai.net` consists of 3 steps:

S.1) The client nameserver looks up `akamai.net` at the gTLD servers. The authorities returned are Akamai *Top Level Name Servers* (TLNSes).

S.2) The client's nameserver queries a TLNS for `g.akamai.net` and obtains a list of IP addresses of *Low Level Name Servers* (LLNSes). The LLNSes returned should be close to the requesting system. As discussed in Section 3.1, the TLNSes ordinarily return eight LLNS IPs in three different regions.

S.3) The client requests `a123.g.akamai.net` from a LLNS and obtains two server IPs.

In steps (S.1) and (S.2), the responding nameserver consults a *map* to select appropriate IPs. In the case of the TLNSes, this map is produced by a special type of region called a *Mapping Center*. Each Mapping Center runs software components that analyze system load, traffic patterns, and performance information for the Akamai network and the Internet. With this information, a top-level map is produced and distributed to the TLNSes.

When the server receives the HTTP request, it serves the file as quickly as possible subject to the configuration for the particular file. In this example we assume the file

is not on the server and the server requests the file from the origin server.

4.2 Operation under Failure

Let us now consider this operation under a case of extreme failure. Consider a model where a user selects a TLNS in network X, which also contains the current lead Mapping Center. In normal mode, the user would select Akamai region A, which would in this case need to fetch content from an origin in network C (for Customer).

We now assume three *independent and simultaneous* failures:

1. Network X has significant packet loss across its backbone.
2. Akamai region A fails.
3. Another network, B, has a problem with a peering point that is on the default (BGP) route between region $\hat{A}$ and network C.

We now examine how these problems might be handled. The particulars of the response of course depend on other circumstances and failures that may be occurring.

1. Network X degrades

- One of the metrics for leader election of Mapping Centers is connectivity to the rest of the Akamai network. Therefore, a second Mapping Center (MC) $\hat{X}$ assumes leadership from MC X and begin publishing the top-level maps. (Principles 1 and 3)
- The user's nameserver may fail over to using another one of the TLNSes. (Principle 1)
- Since some users may continue to query TLNS X, it still obtains the top level maps from MC $\hat{X}$ via the real time multi-path system. (Principle 2)

2. Region A fails

- Upon seeing that LLNS A does not respond, the client's nameserver will start using one of the other LLNSes. Let us call that LLNS $\hat{A}$. (Principle 1)
- Since region A is no longer optimal for the client, LLNS $\hat{A}$ directs the user to region $\hat{A}$. (Principle 1)
- Mapping Center $\hat{X}$ considers this failure and updates the top level maps. This includes removing region A as a choice for users and perhaps some additional load balancing. (Principle 1)

3. Network B has a problem at a peering point.

Those edge regions that use this peering point will attempt to route around this problem using SureRoute (Principle 2) by finding appropriate intermediate regions (Principle 1). This decision is made at the Edge Server in network $\hat{A}$ (Principle 3).

5 Realizing the Benefits of our Approach in our Software and Operations

In this section, we present a second group of principles relating to how Akamai designs reliable software and insulates the system from software faults. It is impossible in any system for any piece of software to function correctly always. While 100 percent correctness is our goal, we invest significant effort to ensure that if something goes wrong, the system can recover appropriately. *Most importantly, we argue that this set of principles is informed and facilitated by the first set of principles* presented in the previous two sections.

5.1 System Principle #4: Fail Cleanly and Restart

We aggressively fail and restart from a last known good state (a process we call "rolling") in response to errors that are not clearly addressable. There are two reasons that we made this decision. The first is that we knew that our network was already architected to handle server failure quickly and seamlessly. As such, the loss of one server does not pose a problem. The second is that while it may be possible to recover from these errors, the risk of operating in a mode where we are possibly behaving incorrectly can be quite high (e.g., serving the wrong homepage). Thus, comparing the low cost of rolling and the high risk of operating in a potentially corrupted state, we choose to roll.

The naive implementation as described above is however not sufficient. The problem we first encountered is that a *particular server may continually roll*. For example, a hardware problem may prevent a machine from starting up. While losing the server is not a problem, the state oscillations create problems (or at least complexity) for the overall system. Part of our current solution is to enter a "long-sleep" mode after a certain number of rolls. This is a very simple and conservative strategy, enabled by the overall system's robustness to individual machine failures.

The second problem with failing-and-recovering is *network-wide rolling*. While even a set of machines can restart at any time, it is not acceptable for the whole network (or all the machines in a particular subnetwork) to roll at once. This could be a problem for events that affect the whole network at once—such as configuration updates or a new type of request—that could suddenly trigger a latent bug. It is also a complex problem to solve. In our current solution, we enter a different mode of operation when we observe a significant fraction of the network rolling. In particular, instead of rolling we will attempt to be more aggressive in recovering from errors—and perhaps even shut down problematic software modules.

Table 1: Minimum Phase Durations

Release Type	Phase One	Phase Two
Customer Configuration	15 mins	20 mins
System Configuration	30 min	2 hours
Standard Software Release	24 hours	24 hours

5.2 System Principle #5: Zoning

In addition to standard quality practices, our software development process uses a phased rollout approach. Internally, we refer to each phase as a zone and thus call the approach *zoning*. The steps of zoning for both software and for configuration releases are:

- Phase One: After testing and QA, the release is deployed to a single machine.
- Phase Two: After monitoring this machine and running checks, the release is deployed to a single region (not including the machine used in Phase One.)
- Phase Three: Only after allowing the release to run in Phase Two and performing appropriate checks do we allow it to be deployed to the world.

The minimum duration of each phase per release type is summarized in Table 1. The actual time can be longer based on the nature of the release and other concurrent events. On the extreme end, operating system releases are broken into many more sub-phases and we will often wait days or weeks between each phase.

While the concept of a phased rollout is not unique, our system is interesting in that the process is explicitly supported by the underlying system. This facilitates a more robust and clean release platform which benefits the business. The properties of redundancy (P1) and distributed control (P3) enable us to lose even sets of regions with minimal user-appreciable impact. Consequently, this obviates any concern about taking down a machine for an install—or even a full region in Phase Two. This is in stark contrast even to some well-distributed networks. For example, a system with 5 datacenters is more distributed that most, but even in this case, taking even one datacenter down for an install reduces capacity by a 20%—significant fraction.

We believe this is an example where our design principles present us with a significant yet unexpected benefit. Principles 1 (significant redundancy), 3 (distributed control), and 4 (aggressively fail-stop) were not chosen to facilitate software roll-outs. However, over time we have seen that these principles have enabled a much more reliable and aggressive release process, both of which have been a huge benefit to our business. We present some metrics substantiate this claim in Section 6.1.

5.3 System Principle #6: The network should notice and quarantine faults

While many faults are localized, some faults are able to move or spread within the system. This is particularly true in a recovery-oriented system—and this behavior must be limited. One hypothetical example is a request for certain customer content served with a rare set of configuration parameters that triggers a latent bug. Simply rolling the servers experiencing problems would not solve the problem since the problematic request would be directed to another server and thus spread. It is important that such an event not be allowed to affect other customers to the extent possible.

We address this problem through our our low-level load balancing system. Low level load balancing decides, for each region, to which edge servers to assign which customer content. To achieve fault isolation, we can constrain the assignment of content to servers to limit the spread of particular content. In the unlikely case that some customer content instantly crashes all the servers to which it is assigned, this approach allows us to mostly serve all other content.

To effectively respond when seeing problems, we must involve both localized and globally distributed logic. In a purely local solution, we could constrain all regions equally. This has the downside that, for example, a customer with geographically localized traffic may produce more load relative to the number of servers than we'd be able to load balance effectively, even though the customer posed little risk to the global network. As a result, we've designed a two-level solution. The mapping system, via the messaging infrastructure, gathers data from all regions and tells the low-level load balancing systems in each region what constraints they need to impose in order to keep the system in a globally safe state.

6 Evaluation

Evaluating our decisions in a completely scientific fashion is difficult. Building alternative solutions is often infeasible and the metric space is multi-dimensional or ill-defined. While it is not a rigorous data-driven evaluation, this section presents two relevant operational metrics.

6.1 Benefits to Software Development

Zoning, as presented in Section 5.2, allows us to be more aggressive in releasing software and other changes by

Table 2: Software Release Abort Metrics

Phase	Number Aborted	Percent of Total Releases
Phase One	36	6.49%
Phase Two	17	3.06
Add'l Phase	3	0.54
World	23	4.14

exploiting our resilience to failure. Over the past year, we have averaged 22 software and network configuration releases and approximately 1000 customer configuration changes (approximately 350 change bundles) per month. Here we examine the number of aborts in each phase of the release process as a metric for the number of errors found. This metric is somewhat subjective. It depends on our ability to catch the errors and our willingness to allow a release with a known (presumably small) problem to continue to a subsequent phase. Further, the optimal value for this metric is not clear. Zero aborts is an obvious goal; however, seeing zero aborts would likely imply little trust in the network's resiliency—and likely reflect a longer time-to-market for features. On the other hand, too many aborts would suggest poor software engineering practices. An ideal network would likely have some, but not too many, aborted releases.

Table 2 presents data taken from several hundred software and network configuration releases that occurred between 7/1/02 and 8/9/04 (25 months). Despite the aforementioned limitations, several observations can be made. First note that the overall level of aborts is roughly as we desire, relatively low but not zero, at 14.23%. Second, when we do abort a release it is most often in Phase One, where the impact to the network is extremely minor. We also see, surprisingly so, that the number of aborts in the World Deploy phase is greater than the number of aborts in Phase Two. This is due in part to the complexity and difficulty in testing a large-scale real-world system. (This is an area of ongoing research at Akamai and we believe an emerging research area of broader interest.) Finally, we note that all the all of the problems found in zoning made it through substantial developer testing and QA.

6.2 Benefits to Operations

A third interesting metric is the number of employees and the amount of effort required to maintain the network. There are many factors that could make our network difficult to maintain—including the size, the large number of network partners, the heterogeneity of environments, and less technical factors such as long geographical distances, different timezones, and different languages.

However, these challenges are mitigated by our fundamental assumption. In particular:

- We assume in our design that components will fail. Therefore, the NOCC does not need to be concerned by most failures.
- Since a component (e.g. machine, rack, datacenter, network) failure does not significantly impact the network, the system and the NOCC can be aggressive in suspending and not using components. Even if the NOCC is mildly concerned, it can suspend a component.
- We assume that a fraction of components will be down at any time. Therefore the NOCC does not need to scramble to get the components back online.

As a result, our minute-to-minute operations team can be quite small relative to the size of our network. In particular, our NOCC has seven or eight people on hand during the day and only three people overnight. For the sake of comparison, that means a ratio of over 1800 servers per NOCC worker during the day and over 5000 servers per NOCC worker at night.

Acknowledgments

The systems discussed in this paper were designed and developed by numerous Akamai employees past and present. We are grateful to Armando Fox for a very careful reading of an early version of the paper and many useful comments and the help of our shepard, Brad Karp. We also gratefully acknowledge conversations with Steve Bauer, Rob Beverly, Leonidas Kontothonasis, Nate Kushman, Dan Stodolsky, and John Wroclawski.

References

[1] Akamai technologies homepage. http://www.akmai.com/.

[2] Net Blackout Marks Web's Achilles Heel. http://news.com.com/2100-1033-267943.html?legacy=cnet.

[3] Recovery-Oriented Computing: Overview. http://roc.cs.berkeley.edu/roc_overview.html.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[5] A. Fox. Toward Recovery-Oriented Computing. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, 2002.

[6] J. Dilley et al. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[7] L Kontothanassis et al. A Transport Layer for Live Streaming in a Content Delivery Network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[8] D. A. Patterson. Recovery Oriented Computing: A New Research Agenda for a New Century. In *HPCA*, page 247, 2002.

[9] I. E. Paul Vixie and W. Akira Kato. Dns response size issues. Dnsop working group ietf internet draft, July 2004. http://www.ietf.org/internet-drafts/draft-ietf-dnsop-respsize-01.txt.