Akamai

WHITE PAPER

Akamai's Approach to Resilience

Updated August 4, 2022

Table of Contents

Addendum

Power resiliency in Europe

With the news of potential energy shortages across Europe, our customers have been expressing their concerns on how these shortages might affect Akamai service availability and sustainability goals in the region. Since the start of the crisis in Ukraine, Akamai has been monitoring the situation and its cascading effects on life and business, specifically technology infrastructure, in neighboring regions. While Akamai is not immune to the pronounced risk of energy shortages in Europe, one of Akamai's greatest values to its customers is the resilient, distributed architecture on which it is built. We have recently released the "Akamai's Approach to Resilience" white paper. Below is an addendum exploring the topic further, with a special focus on our customers' concerns about the European Energy Crisis: energy availability and sustainability.

Akamai's platform utilizes a highly redundant and distributed collection of data centers it has at its disposal, a very diverse set of network connectivity providers, and power sources across the globe. Through these relationships, Akamai is continually making strategic decisions on where to deploy its servers and connectivity infrastructure to best meet the needs of its customers. A foundational part of this strategy is to deploy adequate redundancy across a diverse set of resources and geographies such that Akamai can continue to provide best-in-class reliability, performance, and scale.

In Europe, Akamai is present in more than 500 data centers connected to over 370 networks. In Germany alone, Akamai is present in 54 data centers across 20 cities. The largest data centers from which Akamai procures services use renewable energy as their primary power source. Akamai's sustainability team conducts regular surveys with our data center vendors to measure alignment with Akamai's overall sustainability goals. In light of recent events in Europe, Akamai has also been working with our data center vendors and partners to understand how they procure energy from utilities and what actions they've taken to protect their customers from market volatility. We are confident in the measures our data center vendors and partners are making to ensure continuity of service for Akamai, and we continue to coordinate with them.

On the software side, to best utilize Akamai's highly distributed infrastructure, Akamai's platform continuously monitors the availability and performance of servers, network links, and data centers at large. Using this information, the Akamai platform automatically routes traffic around problems — including areas that are without power, congested network links, and down servers. A vital design principle of Akamai's platform is the expectation that servers, data centers, and network links will go down. Based on this principle, Akamai deploys servers to numerous data centers in local geographies to allow for redundancy, allowing Akamai's software to automatically route traffic to network links and servers that are best available. All of this activity is continuously monitored by Akamai staff, who then forecast and plan new server deployments to meet the evolving needs of our customers across various geographies and to keep enhancing our platform redundancy and resiliency.

Akamai is closely monitoring the energy crisis in Europe and is actively coordinating with its data center partners throughout the region to ensure that Akamai maintains the reliability and performance our customers expect from us.

Introduction

At Akamai, we recognize the essential role our platform and products play in our customers' ability to deliver exceptional digital experiences. For more than 20 years, Akamai has met the demands of the world's largest sites for digital commerce, streaming media, online banking, and more. Ensuring the resilience of the world's most highly distributed global network of servers for delivering and securing content on the internet, therefore, is a priority at the core of our commitment to Akamai customers and internet users.

In the context of Akamai's products and services, resilience means designing our systems to continue operating despite unforeseen problems in a complex world where anything can go wrong at any time.

Akamai's far-reaching platform is deployed in approximately 4,200 points of presence (PoPs) in 1,400 networks, located in 135 countries around the world. The platform comprises more than 350,000 servers, each running sophisticated algorithms that allow them to act together as one massive, highly reliable system. Akamai uses distributed systems principles and high-availability software techniques to build this reliable service on top of the unreliable building blocks the internet provides.

It's important to note that the internet is a best-effort technology. Information to be transmitted is divided into packets, addressed to a distant machine across the internet, and then sent out on the local wire with the hope it will arrive at its intended destination. There is absolutely no guarantee that it will. Connections may be overloaded, servers and routers may crash, links can be severed, or routing paths may fluctuate. How can anyone both meet the demands for 100% uptime, which is necessary for supporting real-time communications and compute for use cases like retail purchases, financial services transactions, or machine-to-machine communications in the face of this chaos? A down server or a broken network link is not an excuse for an interruption in service — and neither are the inevitable operator errors or bugs that may occur. All of these are conditions Akamai's platform must detect and account for automatically to provide a seamless experience in spite of any underlying problems.

This paper highlights a number of our most important design principles for achieving resilience. While standard techniques like quality assurance best practices, peer review of changes, and vulnerability management are vital elements of a resilient system, they are not covered here. Instead, we focus on the specialized technologies and techniques Akamai employs for resiliency on a highly distributed global platform.

Design principles for resilience

Assume everything will break

The most important step in the design of a highly resilient platform is to begin with a clear recognition that anything that can go wrong, will go wrong. After more than 20 years of operating our global platform, we've seen it all. Hardware fails; network links get cut; data centers catch fire; data centers don't catch fire, but the fire suppression system triggers anyway. Software will have bugs, both simple and complex. Human operators will make mistakes.

Each failure point for a system may have a variety of different failure modes. If a server has a hardware failure of its disks, CPU, memory, networking card, or other components, will it become unreachable? Will it stay online operating at a reduced capacity? Perhaps it will stay online and emit faulty data — a potential catastrophe for downstream systems that rely on that data in a critical manner. Maybe the server will be unreachable initially but later come back online emitting stale data — another potential catastrophe.

The impact of all these failure modes must be considered within the context of a distributed system. If a local node detects it has become isolated from the rest of the world, it should perhaps take itself out of service so it does not provide stale data. But if the detected isolation is due to a widespread fault in a communications subsystem, every node may be isolated, and it is imperative for all nodes to continue operating as well as possible under the circumstances.

To build our highly reliable systems, Akamai carefully studies all failure modes, and designs resilient technologies that will continue operating seamlessly when those failures occur. We constantly review the resilience posture of our systems in the face of these failure modes. In complex systems, dependencies change over time and the designs to address them will need regular updates.

Fault tolerance

The bread and butter of resilience is designing systems to continue operating when they experience some kind of error condition or fault. This principle, fault tolerance, can be achieved through a variety of techniques, each appropriate to a different situation. What follows are some of the most critical design techniques we employ to create fault-tolerant systems, illustrated with examples from Akamai's platform.

Redundancy

Redundancy is perhaps the most common technique for providing fault tolerance. It is applied pervasively throughout Akamai's systems, although in a variety of ways to address different circumstances and an array of potential failure modes.

For example, we use one type of hot-failover redundancy to address a particular set of failure modes related to individual servers in a local cluster. Within the cluster, servers monitor one another to check if their neighbors are alive and correctly providing service. If a server fails, another server in the same cluster will immediately take over the IP address of the failed server to continue providing service at that address. Akamai's deployments are built as racks of machines coordinated by software algorithms to act together as one large, highly reliable content delivery, security, and compute node. If any of the individual servers fail, the system adapts automatically.

We use another kind of redundancy designed to ensure the reliability of our more centralized systems. Note that at Akamai, when we refer to a system as being "centralized" we are often still talking about being deployed in a dozen or more locations around the globe; it is centralized only in comparison with the very highly distributed edge nodes. Some of these centralized systems must operate in a mode where a single node is the decision-maker, or "leader." It is critical that the leader be chosen:

- With care to be the node with the best possible operating posture, having the best set of input information available, the best connections to peer systems, and the least likelihood of encountering a failure mode
- Automatically and rapidly, so that if the leader fails, another node will take over service quickly

- In a stable fashion, so leadership doesn't bounce from node to node unnecessarily

- With the understanding that network disruptions may create a situation in which that a set of potential leaders don't know if they are isolated from another set of potential leaders (and are thus unaware if they may independently elect two leaders operating simultaneously)

Akamai uses specialized distributed systems algorithms designed to ensure redundant versions of critical systems can be deployed globally, yet act immediately to take over service in the event of faults.

Distributed algorithms and local decisioning

An effective resilience technique we employ at Akamai is performing work that is relevant to a distributed node at, or close to, that node. This increases the likelihood of any node's ability to continue providing service in the event of degraded connectivity or capacity.

As an example of this, Akamai's traffic load balancing system operates at two levels:

- The global load balancing system determines which traffic (and how much of it) to assign to each local cluster

- The local load balancing system determines how to spread the traffic among the machines in the cluster to which it is assigned

While it would be possible to perform both functions centrally, we choose to run the local load balancing systems within the local clusters themselves. This allows the cluster to act more as an independent unit that can manage itself in the face of various failure modes, while offloading centralized systems by distributing the workload. Note that while the load balancing software in the local cluster does communicate with other parts of the global platform, it is designed to continue operating in the event these communications fail.

Component-level resilience

The techniques for fault tolerance discussed thus far are system-level techniques, but fault tolerance is equally, if not more, critical at the component level. When a piece of software fails, what steps can it take to minimize the impact of that failure?

Software can fail in a variety of modes, but let's take the example of a process crash, which could be either unexpected or part of a component's strategy to halt when detecting an internal inconsistency. One effective tactic we employ is to ensure that when a process does crash, it can restart as quickly as possible. This may seem straightforward, but a number of factors can slow down a component's restart. For example, if the system is configured to allow core dumps and the process is using a very significant amount of memory, it may take considerable time to write the core dump to disk, which could block a new version of the process from starting. Completely disabling core dumps carries its own risks for debugging unexpected problems, so another approach must be taken.

More commonly, it is possible for a component to take an unusually long time to restart if it has accumulated a large amount of configuration it must process before offering its service. Measuring and managing restart time to ensure it remains fast is one of many effective strategies for the component-level resilience that Akamai practices.

Another common but highly impactful failure mode is when a component produces a faulty output that is sent to another component. This may be the result of a logic bug, a hardware failure, or some

other unanticipated fault. One technique we use to mitigate this problem is deploying an output checker, which reads the component's proposed output and runs a set of sanity checks before allowing it to be published. This includes looking for internal incoherence or inconsistency in the meaningfulness of the output, such as data that doesn't make sense within the context of other systems components (e.g., nonexistent ID numbers), or unexpectedly large values or magnitudes of change in the output. In some cases, we use similar techniques implemented as input checkers on downstream receiving components.

Finally, a component that crashes can employ clever resilience strategies if it believes it has crashed due to input from some other part of the system. For example, let's say a component receives a new configuration, crashes, then restarts and crashes again within a short time span. Given the recent arrival of a new configuration and the two crashes shortly thereafter, there's good reason to suspect the configuration is the cause of the crash. If the system dynamics of the component allow for it, that component can automatically revert to using the previous version of the configuration. We design the component's behavior to exit this mode when a new, good configuration is available.

Fault isolation

When all else fails, a critical backstop for achieving fault tolerance is fault isolation. In the event of some unaccounted-for failures, it's important to locally contain the impact of the problem. Akamai uses a sophisticated version of fault isolation on our platform to meet the dual requirements of scaling automatically to serve high-demand, high-traffic events while also protecting the broader platform from unexpected problems with that traffic.

As an example, imagine that a site on our platform with moderate traffic trips some type of error condition. Perhaps something specific to that site's configuration combined with a bug in our edge server software only exhibits itself when the site publishes a new piece of content. Despite the techniques we use for component-level resilience, for the purposes of this example, let's assume all other safeties have failed, and the edge server crashes and does not recover automatically. In our normal mode of operation, another server in the local cluster would immediately take over for the failed server, as described earlier. However, in this circumstance, that may be problematic. The second server may also crash because of the same bug. The traffic would continue to shift to one new server after another, a cascading failure eventually crashing all servers in the local cluster.

One simple solution to this problem would be to enforce a hard limit that no single site can use more than a small fraction of a given cluster. But that would severely limit the ability to automatically scale to use all the resources necessary to deliver content to end users from the best possible node. Instead, Akamai employs a more sophisticated resilience design that detects if unexpected server crashes are attributed to a particular site's traffic. Under normal conditions, we allow a site to scale to use all necessary resources, but if error conditions are detected, scaling is halted and the damage is limited in scope.

Automation and self-healing

A key principle that comes into play in Akamai's resilience strategy is a design philosophy of automation and self-healing. With a network of over 350,000 servers, it will never be possible for operations staff to react to problems manually fast enough. It is imperative, therefore, that problems are detected and mitigated automatically. Automated mitigation is the only way to scale, and it is the only way to react quickly enough.

Automation and self-healing principles are pervasive at all layers of Akamai's technology stack. As previously mentioned, machines in a cluster automatically take over for other machines in the event of a failure. In fact, a "down edge server" alert is the lowest priority alert in our system. We know it will happen often, but it's not a high priority because the system will accommodate it seamlessly — repairing the machine can happen on a longer time scale.

At the same time, if an entire cluster of servers or even an entire data center goes offline, our system reroutes traffic quickly and automatically to the next best data center, without any operator intervention. This is discussed in greater detail in the following section.

Graceful degradation, traffic control, and scalability

Because anything that can go wrong, will go wrong, it is imperative to design for failure modes that allow for a graceful degradation in service. It's often the case that, with extra forethought and design, various failure modes can be made to allow for "good enough" service to continue instead of complete unavailability.

This is a design principle Akamai employs in many different contexts. One example is our approach to traffic mapping, by which we mean the assignment of end users to clusters around the world. Our traffic mapping system also clearly illustrates resilience principles related to control mechanisms and scalability, so all three will be discussed here.

Akamai's traffic mapping system is designed to direct each new end-user request for content to one of our global clusters, based on real-time performance measurements of internet traffic conditions among users and clusters. This is designed to ensure that users get the best possible performance. The cluster physically closest to a user is not always the one with the best performance when the internet is suffering from certain kinds of connectivity problems.

But this performance-based mapping mechanism also plays an important role in the resilience principle of graceful degradation. If a cluster goes offline or is overloaded, we can direct end users to the next-best-performing cluster, likely in the very same city as the first one. If even more capacity is required — such as during a very high-demand, live sports event — we can continue step-by-step, employing additional nearby resources and continuing to give very good performance when optimal is not possible. Our traffic mapping system's DNS responses have 20-second time to lives (TTLs are the settings that tell the client's DNS resolver how long to cache an answer before requesting a new one). This gives us the ability to react quickly, whether when redirecting users in the event of an increased demand in traffic, or in response to a data center that unexpectedly goes offline.[1]

There are two additional benefits to the fine-grained control and load feedback that Akamai's traffic mapping provides. First, it allows us to make very efficient use of resources. Servers, clusters, and network links can be driven to use nearly their full capacity because we have the capability to direct each next request to another resource if one becomes fully loaded. Second, by addressing clusters directly, we have a great deal of control over the direction of traffic — far more than through the use of technologies like Anycast, which have occasionally been the cause of severe disruptions to services.

[1] Note that because extra DNS lookups can impact performance, an additional component of our system is designed to ensure that users are making their low-TTL DNS lookups from nameserver software that is also mapped to be very close to the end users, with a selection of servers made available for both performance and resilience.

Change safety

As a matter of practice, many significant failure modes exhibit themselves as the result of a change that carries unintended consequences. A classic example is pushing a configuration update to a software component that triggers a bug in the software or causes the software to crash. Given the pace of innovation on the internet, it's not feasible to ensure all software will be free of bugs. So, how do we achieve resilience beyond the component-level resiliencies previously discussed?

One of the most effective methods Akamai employs is performing staged rollouts of new configurations, with automated testing between stages. An automated configuration rollout and safety system can help manage this. The parameters of staged rollouts will vary based on the risk profile of the configuration channel in question, but let's talk through one example:

- When a new configuration is ready to deploy, it is first sent to a nonproduction testing system. If the software on that system crashes or experiences an error, the automated deployment system halts rollout and alerts an operator.

- If the nonproduction testing passes, the configuration may then be sent to a small number of production servers. The selected servers should be relevant to the configuration changes being made, but small enough in scope so that a fault will be of limited impact to the overall correct functioning of the service. If automated testing detects the target servers exhibiting faults or not operating correctly, the automated deployment system halts the deployment of this configuration and alerts an operator to the situation.

- If the target servers in the previous stages continue to look healthy, rollout can continue in a staged manner until the configuration is fully deployed with a pace and staging appropriate for the needs of the configuration and components in question.

For some manual processes, such as actions taken by operations staff, we use a "what if" tool that determines if an action will have unsafe consequences, alert the operator to this fact, and potentially require an override to continue

Layering of safeties

While the principles of resilience described throughout this document are all valuable independently, their effectiveness increases exponentially when they are combined. The power of layered safety is so significant that it is itself a resilience principle.

Talking through the simple example of deploying a configuration that would trigger a bad bug in software illustrates this clearly:

- When the configuration is pushed that would trigger the bug, it is first automatically tested on a nonproduction machine. If the bug exhibits there, rollout halts and the failure is averted.

- If the live testing does not exhibit the problem, the configuration is rolled out in stages to the production network, where additional testing will halt rollout if the bug presents itself.

- If the bug still has not triggered by the time the configuration has reached global deployment (perhaps because the bug is dependent on a transient condition), the defensive posture of the component's resilience kicks in. The software may crash, but if it restarts quickly it still has the chance to provide good service.

- If the bug trips a second time not long after the first, and the component's system dynamics can support it, the component decides two crashes in a row may be the result of a bad configuration and automatically reverts to using a previous version of the configuration.

- At the same time, if the transient condition necessary to trip the bug was based on a request to a particular site, the fault isolation system will prevent the traffic mapping system from spreading this site's traffic to too many servers.

- That's important for keeping the "good" traffic working correctly, but also gives the traffic mapping system the opportunity to send some of the good traffic to additional resources if local capacity is degraded due to the failures.

Visibility

Having good visibility into systems status would seem an obvious operational requirement for resilient systems. Good telemetry allows operators to detect problematic conditions and repair them before they cause an interruption in service. However, there are subtleties in how to provide the right kind of visibility for when things go wrong. Standard techniques for logging warnings or errors may alert an operator to a problem, but they are often woefully insufficient for answering the next question about the scope or nature of the problem, or how to diagnose it.

Take, for example, a server that sends a warning message about approaching overload when it is at 85% capacity utilization of some resource. If half the machines on the network log that message, it suggests there's a looming problem, but then what? The operator only has more questions the log messages can't answer. Is the rest of the network nearly at 85% load or far below it? Does the majority of the load on those servers have something in common, like a single customer, traffic type, software version, or configuration? Is there anything else unusual happening on those servers, but not others? Unfortunately, the rigidity of logging can't help us answer these questions.

For this reason, Akamai built a telemetry system that gives far greater flexibility, especially in the face of complex and unexpected problems. With this system, the software developer of a component can make arbitrary underlying data about the component available via an API. Operators can query for any of this information remotely as needed. What's more, the interface to this system is SQL, with the telemetry system itself aggregating together data from servers and components across the platform, on demand. The result is that an operator can write an ad-hoc SQL statement as if against an enormous database of systems information, but that's actually the live status of the network.

It's hard to overstate the power of such a system. Imagine what would happen in the scenario described above in which a component reports high load. Instead of being hard-coded to log at 85%, the component simply makes its current load percentage available via telemetry. The alerting system queries the entire network for the component's current load and triggers an alert that too many servers are at 85% load. But now the operator can easily answer all the other questions above. By issuing new queries, they can determine the load on the other components, understand the distribution of load among the ones that did alert, and combine tables with other information such as customer traffic levels, configuration settings, software version, and more.

Security model

The focus of this document has largely been on the design of techniques for resilience in the event of faults rather than on the software design itself.

However, the security design of software and systems carries an outsized importance in overall systems resilience and so deserves attention.

A number of relatively common security practices are highly detrimental to systems resilience. One example is overreliance on IP access control lists (ACLs) as a primary control for access or systems communications. While an IP ACL is a worthwhile addition to other security measures, it is imperative that it not be the only control. Strong cryptographic methods, employed both for encryption and authentication, must be the first line of defense across all communications channels.

Another common security practice that carries unnecessary risk is overly broad access privileges. Take, for example, an operator who requires access to a global network of servers to perform maintenance tasks or to debug problems. At the beginning of the worker's shift, there's no way to know in advance which servers the operator will need access to, so in many environments the operator is simply given access to all systems. This, unfortunately, presents risks to resilience and safety. If a single operator's key can access the entire network simultaneously, it's an attack vector for taking the service offline. It's even a risk from a nonmalicious perspective; A bug in a tool used by an operator may now accidentally perform a damaging action on the entire network of servers.

To help mitigate this risk, Akamai uses an access broker system for maintenance and debugging access to servers. While an operator may need access to any server from a given collection, there is no reason during their shift that they will require access to all servers. Instead of giving direct access to servers, the operator authenticates to the brokerage system, which then mediates access to the server. In this way, an operator may be given access to a limited fraction of the servers on the network, and other controls may be imposed as well.

Operational resilience

Although we strive for as much automation and self-healing as possible, it is necessary to have human operators managing the problems that software and systems can't. The goal of the resilience design principles discussed is to allow the system to continue operating seamlessly during the time a human operator needs to take corrective action on an underlying problem.

Akamai has a number of staffed operations centers located around the globe. Operators receive alerts (usually written as SQL against the telemetry system described previously), with a configured severity and linked procedure as an entry point into a workflow to resolve the issue. Systems have a clear list of contacts for escalations when standard procedures can't resolve a problem.

Operations staff are also responsible for installing new software on the network. This, as with configuration deployment, is performed in stages, although striped differently across the network. Prior to each install phase, we employ techniques to divert traffic away from the machines to be installed. Post-install monitoring assesses the health and performance of parts of the network running the new version of software and compares it with the previous version.

Finally, while the resilience design principles and operational practices described in this document are highly effective at providing a reliable service, Akamai also has a robust incident management framework in place for when things go wrong. Solid incident management is itself a critical resilience technique, as it can truly make the difference between an outage that lasts minutes and one that lasts an hour.

Conclusion

Building a highly reliable service on top of the internet's "best effort" infrastructure is a complex challenge — one that Akamai has embraced for more than 20 years. In this paper, we've explained a number of the most important design principles for providing a highly resilient service on Akamai's globally distributed platform. Although the principles are described here at a high level, Akamai's team of experts in systems architecture, distributed algorithms, software design, security technologies, and operational practices focus on the details to ensure the robustness of our technologies and their application throughout our platform.

Through ongoing innovation and investment, Akamai is continually improving and refining our resilience strategy and tactics to keep pace with changes in technology, content and traffic dynamics, and our customers' evolving needs. We are committed to providing the highly reliable platform and products that empower our customers to deliver digital experiences that delight their customers and drive their business growth.