An Edge Networking Playbook

by Brian Pane

August 23, 2024

Copyright © 2024 by Brian Pane

This work is licensed under CC BY-ND 4.0.

The diagrams in Chapter 1 incorporate public domain maps from Wikimedia Commons.

Contents

Acknowledgements 4

1. Introduction 5

2. Edge Network Basics 9

3. Roadmap 15

4. Real User Monitoring 18

5. Planning Edge Services 21

6. PoP Site Selection 27

7. Network Connectivity 29

8. Edge Software Services 37

9. Traffic Steering 42

10. Rollout and Operation 48

Acknowledgements

More than a decade ago, I stumbled into the world of edge networking. Coming from a software background, I was looking for new ways to make applications faster. As I began to build edge services, I grew to appreciate how broad the field was, with people from vastly different areas of expertise coming together to solve hard problems. Over the years, I was fortunate to learn a lot from my colleagues across many companies.

In the process of writing this book, I was reminded of the power of the community. I thank all the volunteer reviewers who helped shape the content and shared new learnings with me in the process: Alexey Ivanov, Charles Thayer, Harshiva Matcha, Hossein Sahabi, Ian Holsman, Liuyang Li, and Naveen Achyuta.

Brian Pane
August, 2024

1. Introduction

Audience

This book describes how and why to use networking technologies to make online services faster on a global scale. It is written for people working in the diverse roles that come together to make such endeavors possible: engineering teams who build the software and networks, finance teams who guide the capital planning, legal teams who navigate the risks, executives who drive the strategic tradeoffs, and program managers who coordinate it all.

Background

People increasingly depend on *online applications* for everything from communication to entertainment to commerce. These applications are complex and diverse, but most follow the pattern shown in Figure 1: the user interacts with some client software (web browser, phone app, etc.) that communicates over the Internet with backend services run by the *application provider* organization.

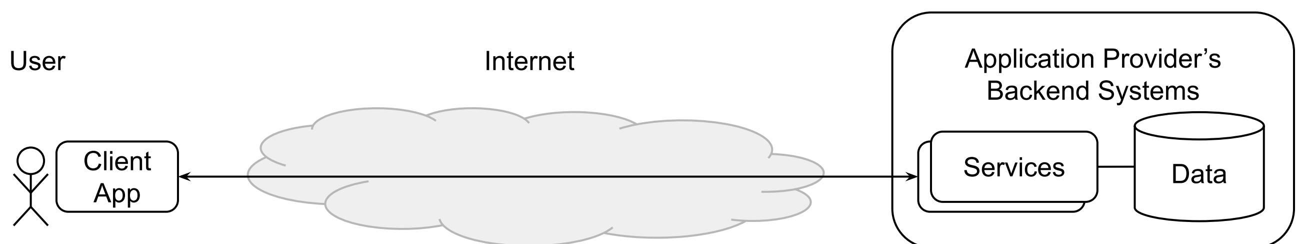


Figure 1: Online application

From the user's perspective, life is better when this communication is fast and reliable. If a ride-hailing app starts up and finds a driver quickly, the passenger can get to the airport on time. If a brokerage app submits trades quickly, the client can react to a rapidly changing market. If an online game has a responsive, low-lag connection, the player can avoid the ignominy of losing to a novice competitor.

Making an online application faster is beneficial for the application provider, too. Many companies have found via A/B testing that their product KPIs improve when they accelerate key user interactions. When a search engine delivers results more quickly, for example, users

conduct more searches.[1] When an e-commerce website's pages load more quickly, more users complete the checkout process; one study found that a 100 millisecond speedup across key use cases produced an 8% increase in conversions.[2]

Often, a popular online application will attract users all around the world. This creates a challenge for the application provider: how to make the application's client-to-server interactions fast for users located far away from the backend servers and data. Figure 2 shows an example where the online application, having started out with its backend systems in a datacenter in North America, has become popular with users in other continents.

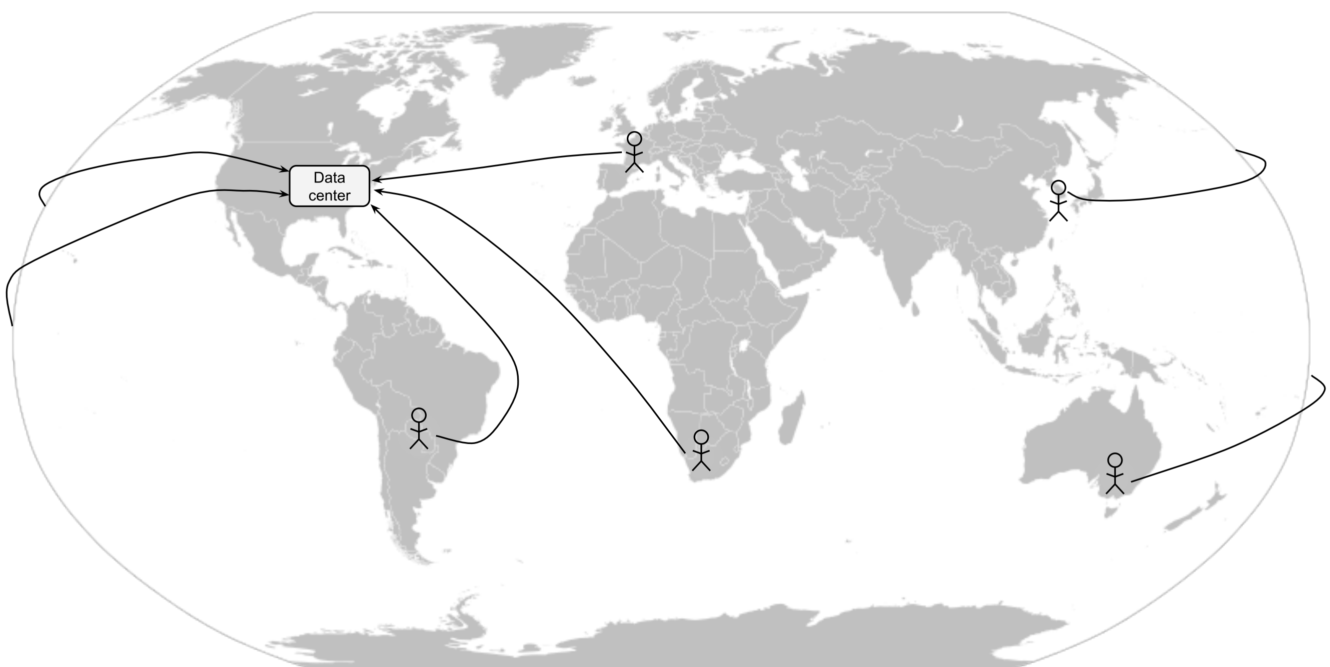


Figure 2: Online application with users distributed around the world

Figure 3 shows typical network Round Trip Time (RTT) measurements between a datacenter in Ashburn, US and each of the user locations from Figure 2.

---

[1] Speed Matters, Google, 2009

[2] Milliseconds make Millions, Deloitte, 2020

Location	Typical RTT in msec to Ashburn, US
Paris, FR	75
Asuncion, PY	180
Seoul, KR	195
Melbourne, AU	210
Cape Town, ZA	220

Figure 3: Network round trip times[3]

The RTT represents the minimum time it will take the online application to process any operation that requires a call from the user's client software to the backend datacenter. If the user gets a higher-bandwidth Internet connection and a faster computer, or the application provider upgrades their datacenter with faster servers and better-optimized software, the RTT still will not get any faster. And some common interactions will take many round trips (Chapter 2 covers the details).

In addition, the network path between any of those client locations and the backend datacenter may run through multiple third-party providers with unpredictable performance and reliability. That results in even worse performance than the RTT numbers suggest.

Approach

Given these challenges, how can the application provider deliver a faster user experience? One approach is to clone the backend systems and deploy them in datacenters throughout the world, so that each user can interact with a closer datacenter. In practice, though, the number of backend datacenter locations is limited by cost, operational complexity, application complexity (especially for workflows that require strong consistency for replicated data), and geopolitical issues. As a result, even organizations with very large global footprints do not have backend datacenters in every country where they have users.[4]

Therefore a common way to improve the application user experience is by using an *edge network*: a combination of networking and software infrastructure located closer to the users that makes the application faster. Figure 4 shows an example: when the user in France interacts

[3] Public measurements from WonderNetwork, May 2024

[4] See, for example, Meta's and Amazon's datacenter locations.

with the online application, the network communication flows through a London *Point of Presence* (PoP) operated by the application provider.

Figure 4: User interacting with a local Point of Presence

An edge network consists of many PoPs distributed throughout the world. Each PoP contains networking equipment and possibly also servers and storage, depending on the nature of the online application(s) it serves. This infrastructure can do several things to improve the user experience: serving some parts of the application locally, caching popular content, accelerating network protocols, and providing a faster, more reliable network path for the interactions that need to go all the way to the backend datacenter.

Edge network capabilities can be obtained either by building one's own infrastructure or by purchasing hosted services from third parties. The right choice — which often will be a hybrid of building and buying — depends in large part on the application provider's size, capabilities, and priorities.

Chapter 2 explains how edge networks function. Chapter 3 lays out a roadmap for application providers who want to develop edge network capabilities, and the subsequent chapters explore each step of the roadmap in more detail.

2. Edge Network Basics

Reference Architecture

Figure 5 presents a high level reference architecture for an edge network. Later chapters will zoom into various parts of this model.

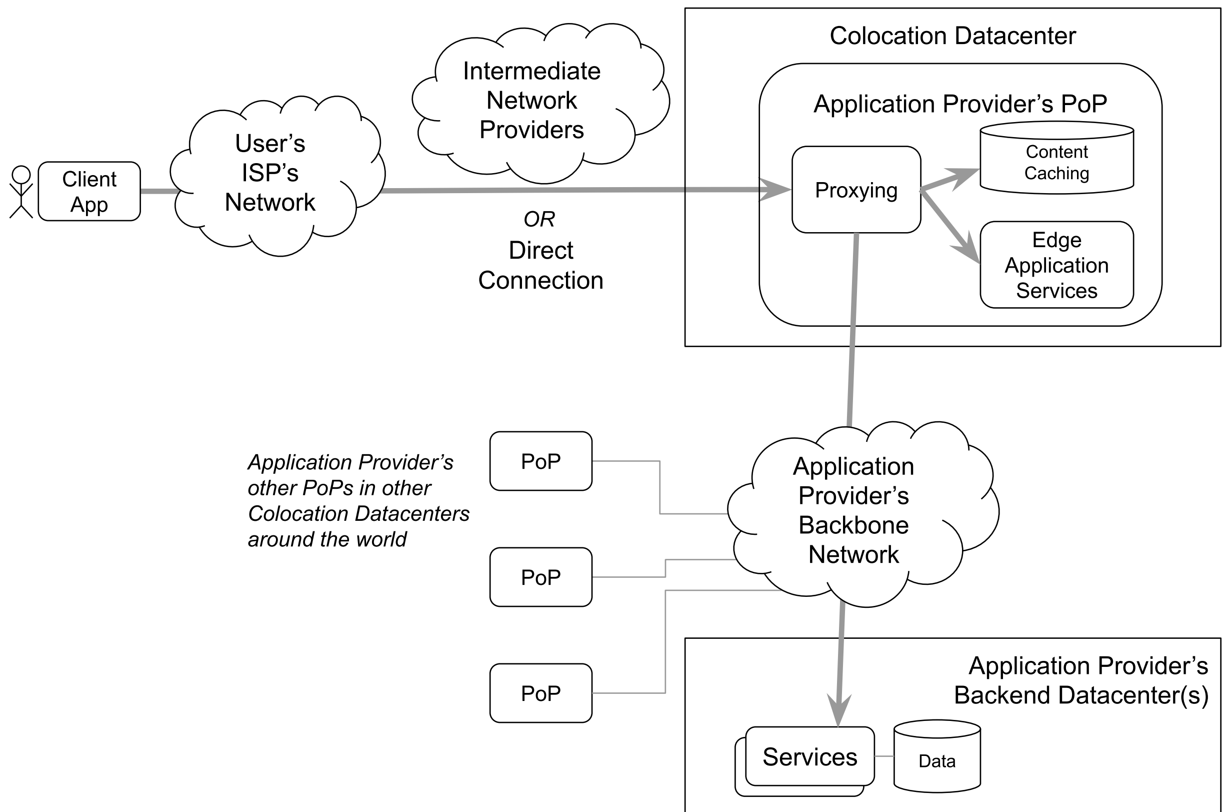


Figure 5: Edge network reference architecture

Physical Facilities

Application providers usually build their PoPs in rented space inside *colocation datacenters* ("colos"). There are multiple reasons for hosting in someone else's existing facility, rather than building one's own edge datacenter from the ground up. First, the lead time is much shorter. In addition, the amount of floor space needed for a PoP is often too small to justify building a new datacenter. Finally, one of the most important functions of a PoP is to connect the application provider to external networks, and this interconnection becomes easier if the PoP is located in a multi-tenant facility where many other Internet providers already have a presence. Chapter 6 covers site selection in more detail.

As shown in Figure 5, the client sends requests (the leftmost grey arrow) to the application provider's PoP. If needed, these requests can flow over the open Internet, through third party networks. But the

application provider can achieve better performance and reliability by establishing a direct connection to the client's ISP in the colo facility. Typically, the PoP also will connect to a *backbone network* that links all of the application provider's sites together. By bypassing the open Internet for most or all of the distance between the client's ISP and the application's backend servers, this technique can improve reliability and performance: even though the best-case latency is still dominated by the speed of light, the common case may be substantially improved. Chapter 7 discusses the connectivity options.

For the application provider, an alternative to building an edge network is to purchase services from a third party. Content Delivery Networks (CDNs) and general-purpose cloud providers offer various edge services with pricing based on usage. In general, building and operating one's own edge network is cheaper at large scale but has nontrivial fixed costs in the form of both the physical infrastructure and the people's time needed to manage it. In contrast, buying edge services from a third party often results in a higher unit cost of capacity, but with the flexibility of little or no fixed cost. The right choice will vary based on the size and goals of the application provider organization.

Edge Software Services

Depending on the specifics of the online application, it may also make sense to run software services in the PoPs (aka "at the edge").

Proxying

Chapter 1 introduced the notion that network RTT is a lower bound on the time needed for any synchronous, client-to-server interaction in an online application. In practice, necessary protocol overhead often adds additional round trips.

For example, the transport protocols commonly used by online applications provide secure and reliable delivery of data, but they require some initial back-and-forth communication before the application can start sending the first message. Figure 6 shows this startup overhead for various popular protocols.

Transport Protocol	Round Trips for Protocol Setup
TCP with TLS 1.2	3
TCP with TLS 1.3	2
QUIC	1

Figure 6: Overhead of secure session setup[5]

Client application developers know that new connections are slow, so they use techniques like connection pooling to reuse existing connections for subsequent messages. However, during application startup and in other scenarios where the client app needs a new connection, the additional round trips for protocol setup are inevitable.[6]

Even after a connection is established, the RTT still acts as a limiting factor. The two most commonly used transport protocols, TCP and QUIC, both use a *slow start* strategy: on a new connection, they send only a small amount of data, because they do not yet know how unreliable or overloaded the network path might be. When the other end of the connection acknowledges that it has received this data, the sender grows more confident and sends a bigger block. This process continues for multiple iterations, each taking one RTT, until the amount of data being sent (the *congestion window*, in protocol lingo) is large enough to make good use of the available bandwidth. The bigger the RTT is, the longer it takes the connection to ramp up to full speed. Figure 7 shows an example of slow start for a data transfer over a new TCP connection with an RTT of approximately 50 milliseconds; it takes over a second for the connection to ramp up to full speed.

[5] HTTP versions up through and including HTTP/2 use TCP + TLS, while HTTP/3 uses QUIC.

[6] TLS 1.3 and QUIC offer a “zero-RTT” mechanism that makes it cheaper to establish additional connections after the first. Such protocol optimizations, where available, are complementary to the edge proxying speedups described in this playbook.

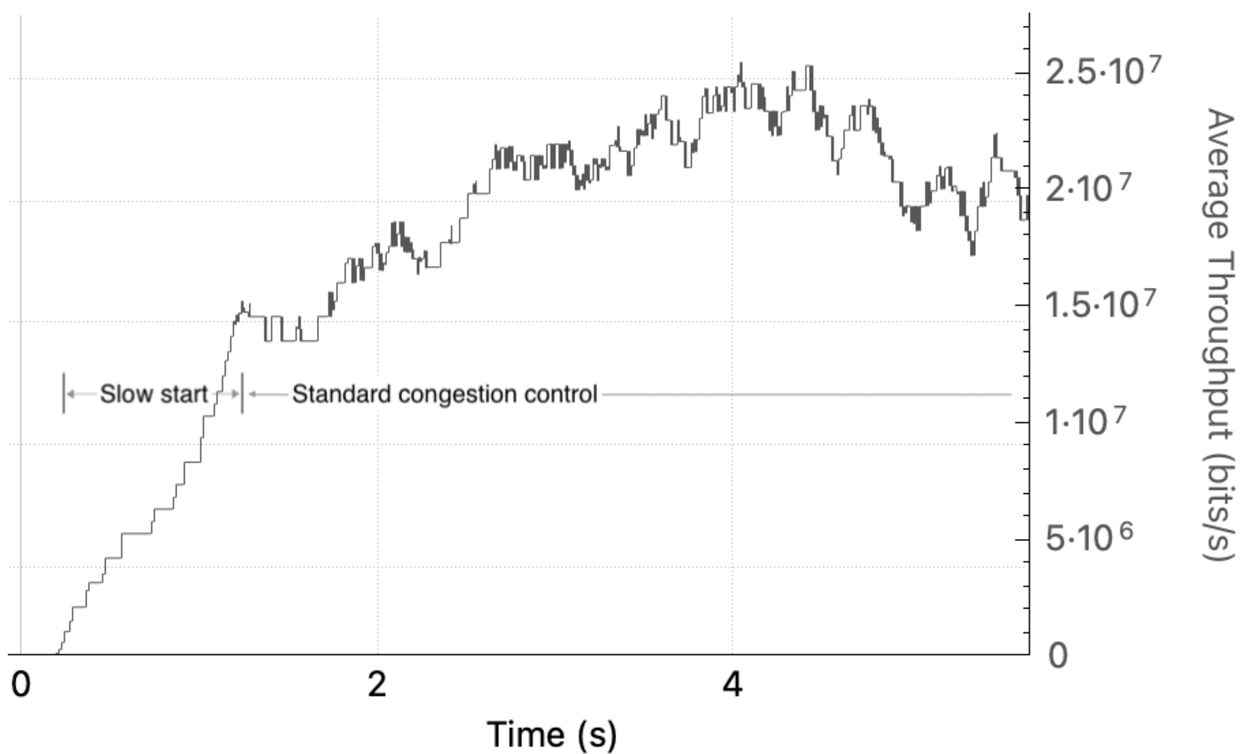


Figure 7: TCP slow start example[7]

Given these impacts of RTT on application performance, a simple but effective service to add to PoPs is a *proxy*, which intermediates the protocol communication between the client and backend apps. Figure 8 shows the basic operation. Instead of talking directly to the backend services, the client connects to the proxy in a nearby PoP. The proxy maintains a pool of reusable, secure connections to the backend app. Whenever the proxy receives a message from the client, it forwards the message to the backend app over one of the pooled connections.

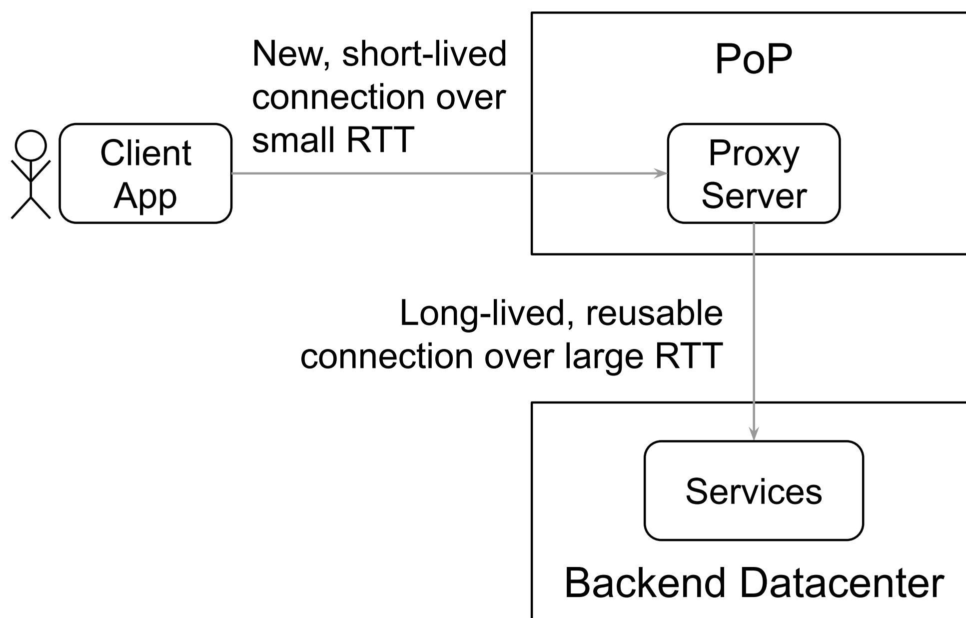


Figure 8: Edge proxy

[7] Data captured and plotted using Wireshark

The communication between the client app and the proxy still has to make all the round trips required by the transport protocol. However, these round trips happen over a smaller RTT, because the PoP is close to the client. And when the proxy forwards messages to the backend application far away, the use of a preexisting connection lets it avoid the extra round trips for connection establishment and slow start.

With a secure protocol like TLS or QUIC, the proxy in the PoP must decrypt and re-encrypt the messages passing between the client and the backend services. This adds a nontrivial engineering challenge: the proxy implementation needs to be especially secure. However, once a secure proxy is in place, its ability to see messages flowing between the client and the backend servers can help solve other problems. For example, if different users' data is homed in different backend datacenters, the proxy may be able to look at cookies in the incoming requests to choose the right destination for each user. Or, if the proxy sees a suspiciously large volume of incoming requests for a registration form, it can implement rate-limiting at the edge. Chapter 8 discusses the challenges and opportunities in more detail.

Caching

Proxying through a PoP can accelerate the client app's communication with the backend services. An even bigger performance win is possible in cases where the PoP can satisfy a request from the client without having to call the backend at all. In online applications, it is common to have data that many or all of the clients need to fetch: graphics and stylesheets, JavaScript libraries, configuration updates, news feeds, recommendations, and so on. The proxy software in the PoP can *cache* this data for fast delivery to the client.

Edge Application Services

Depending on the application, it may be possible to run some or all of the backend services in the PoPs. This provides an additional performance benefit, but with a significant caveat: any service deployed into PoPs becomes a globally distributed system, with a high latency between its components. Some services work well in that environment, but many do not. In addition, hosting servers in PoPs tends to be more expensive than in backend datacenters. Some examples of services that

can work well at the edge are autocompletion (but usually not full-scale search engines), ML inference (but usually not ML training), game engines, chat servers, and logging (and some streaming analytics processing). Chapter 5 provides some recommendations on choosing which services to run at the edge.

3. Roadmap

Figure 9 shows the sequence of major steps needed to plan and build a new edge network.

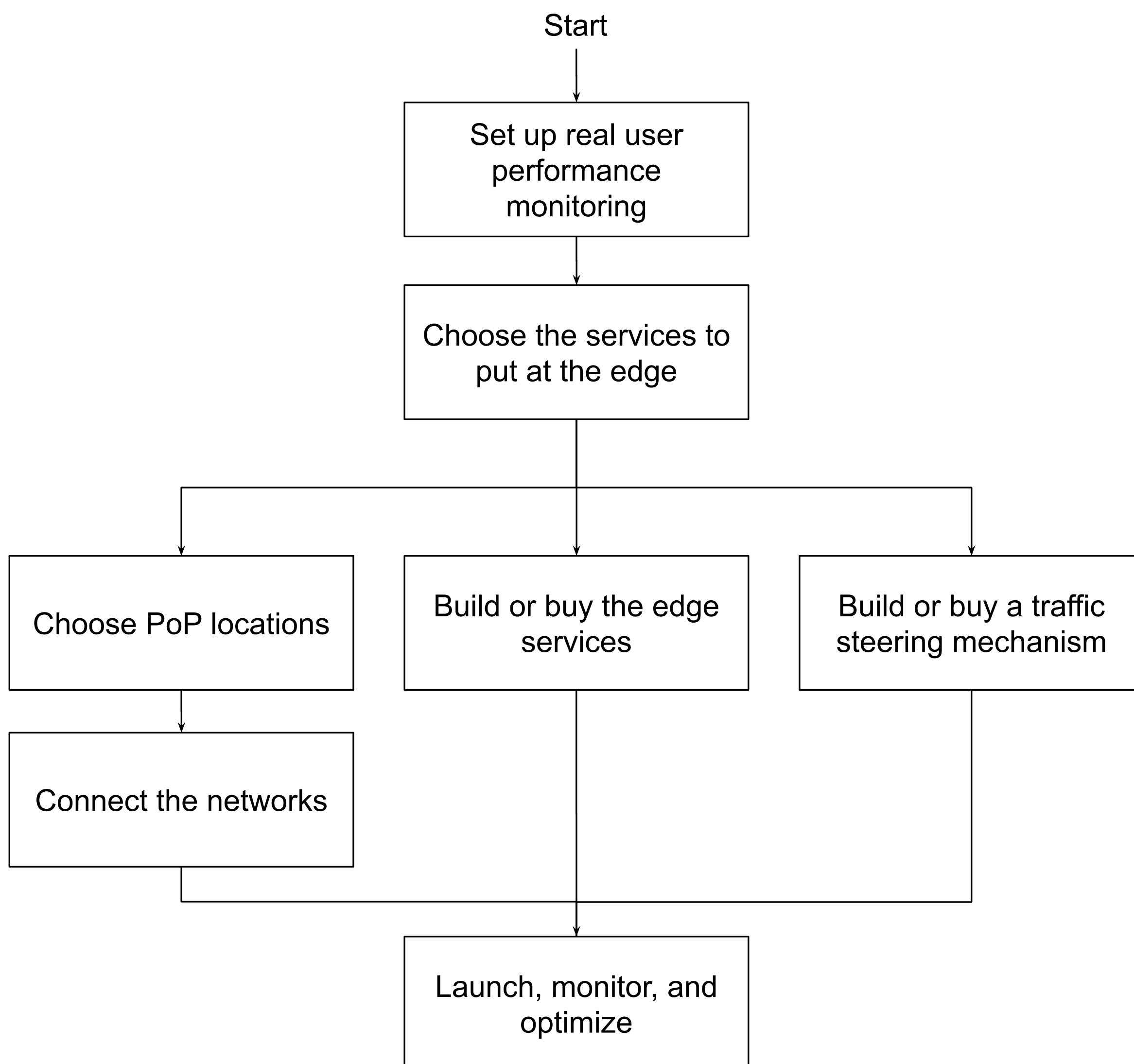


Figure 9: Edge planning and execution roadmap

Set up real user performance monitoring (Chapter 4)

> The first step in developing an edge network strategy is to measure and understand the network performance as experienced by the users of one's online application. This will help guide and justify the subsequent investments, and the data often will be useful in ongoing operations.

Choose the services to put at the edge (Chapter 5)

> The next step is to decide which services to run in the PoPs: just routing, for example, or proxying and caching, or even part or all

of the backend application. The selected services will determine what type of equipment the PoPs need, and with how much capacity, and this information will feed into site selection.

Sometimes this analysis will indicate point toward different priorities. For example, if a critical application workflow consists of one second of network communication and ten minutes of computation, the network is not the first thing that needs engineering attention. Or, if the client is using inefficient network protocols to talk to backend services, fixing that first (e.g., by replacing HTTP/1.1 + TLS 1.2 with HTTP/2 + TLS 1.3 or QUIC) will provide an interim win that will also work well with a future edge network.

Choose PoP locations (Chapter 6)

Edge site selection is the process of finding PoP locations with the right balance of proximity to users, connectivity to the rest of the Internet, available capacity, and cost. In some cases, the easiest solution may be to purchase edge services from a third party CDN instead of building one's own PoPs. Country-specific regulations and taxes also are also a factor, so the site selection process requires a multidisciplinary team to analyze the financial, legal, and engineering details.

Connect the networks (Chapter 7)

After choosing PoP locations, the next step is to arrange for the needed network connectivity at those sites. This often will be an ongoing project, starting with a couple of connections to the Internet and then incrementally adding private connections to different ISPs to further improve speed and reduce cost.

Build or buy the edge services (Chapter 8)

> If the edge plan includes services such as proxying and caching, developing or acquiring the needed software will take time. Chapter 8 provides high-level software guidance for implementors.

Build or buy a traffic steering mechanism (Chapter 9)

> The performance benefits of edge networking all depend on the idea that users will somehow talk to the nearest PoP. The technologies that make this actually happen are collectively known as *traffic steering*. There are many design options, each with distinct tradeoffs to evaluate.

Launch, monitor, and optimize (Chapter 10)

> Launching an edge network is a significant investment of time and money in pursuit of a quantitative goal: improving application speed and dependent product metrics. Therefore it is important to use a metrics-driven approach to ensure that the investment is yielding the expected results.
>
> In addition, most edge networks start out small and then grow incrementally over time: more countries, more PoPs, more network connectivity, more software features. The same type of quantitative approach that has informed the initial build can be used to guide future expansion.

4. Real User Monitoring

Before building an edge network to improve network performance, it is important to have monitoring that accurately measures the performance — from the perspective of online application's users.

There are third-party monitoring services that repeatedly send requests to one's web endpoints from hundreds of agents in datacenters around the world to track response speed, but these services generally do not have enough coverage to fully represent last-mile performance as seen by end users. A better approach is to implement *Real User Monitoring (RUM)*: network performance measurement from the application's actual clients. Figure 10 shows the major components in a RUM system.

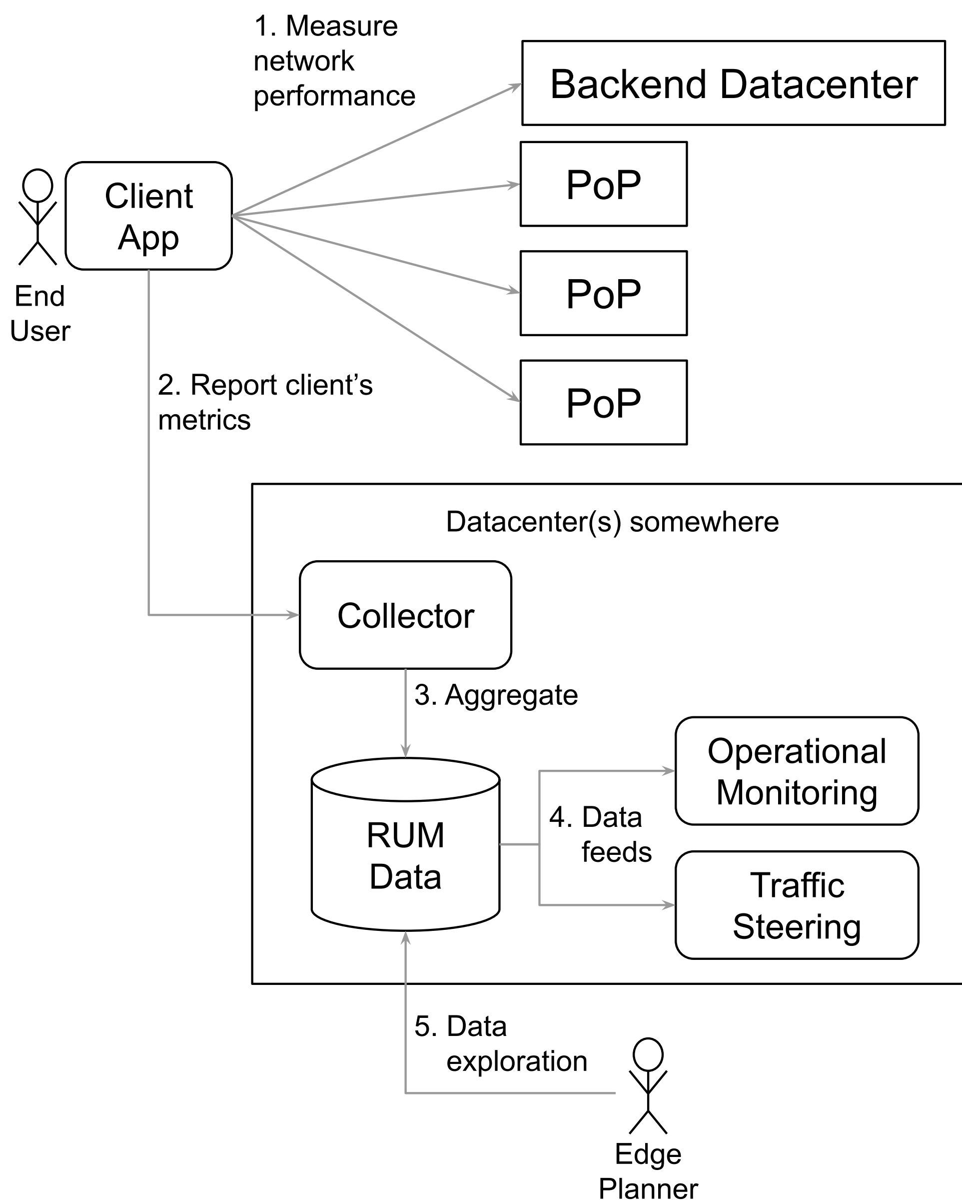


Figure 10: Real User Monitoring

The client application periodically measures the RTT from its location to the PoPs and backend datacenters. If the application performance depends heavily on other aspects of network performance, such as throughput, the client can measure those too. The client app reports its measurements to a central collector service. (If the client uses a web browser instead of a custom app as its user interface, this same measurement and reporting can be done in JavaScript in a web page.)

The collector service aggregates the data from many clients, grouping it by various dimensions to support both long-term planning and day-to-day operations. The data collection and aggregation run continuously, because performance will change over the course of a day (e.g., worse during peak hours due to network congestion) as well as over longer stretches of time (e.g., worse as user growth creates congestion, or better when the network capacity is upgraded).

For edge network planning, aggregating the RUM data by geography will show high-level patterns: "our users in country X have a median RTT of 200 milliseconds to our backend datacenter, so we should consider building a PoP in that region." In addition, aggregating based on a hierarchical geo-encoding such as Geohash makes it easy to display performance patterns on a map.

A secondary grouping by client *Autonomous System* (AS) will show additional insights. An Autonomous System is a group of subnetworks managed under a single administrative domain. In practice, this usually means an ISP or other company or organization, although some organizations have their networks divided into multiple Autonomous Systems.[8] Aggregating the RUM data by AS will reveal things like, "the median RTT in city Y is 50 milliseconds for users on this ISP but 200 for this other ISP, so we should diagnose whether the second one is due to a routing problem."

Another common aggregation of RUM data is by client IP prefix: "users coming from 203.0.113.0/24[9] have a median RTT of 150 milliseconds."

---

[8] RFC 1930 provides an official but still vague definition: "An AS is a connected group of one or more IP prefixes run by one or more network operators which has a SINGLE and CLEARLY DEFINED routing policy." For RUM data analytics, it is almost always sufficient to treat AS as a shorthand for "the user's ISP."

[9] For readers who are not network engineers, that notation 203.0.113.0/24 means "the range of IPv4 addresses that start with 203.0.113." For readers who are network engineers, the example addresses in this playbook are from the documentation ranges reserved in RFC 3849 and RFC 5737.

This view of the data can be used programmatically to drive some forms of traffic steering (Chapter 9). Also, because Internet routing operates on IP prefixes, grouping the performance data this way can help with troubleshooting.

The RUM data stream can be a good source of fine-grained availability signals, too. Observations like “users on ISP X cannot reach our PoP in Singapore at all” can help network engineers detect and triangulate problems. And traffic steering systems can use this availability information to avoid sending clients to PoPs they cannot reach.

5. Planning Edge Services

Chapter 2 introduced several types of services that can be implemented in PoPs: routing, proxying, caching, and hosting miscellaneous application features.

The next steps in edge network planning are to select which of these services are appropriate for one's online application and to conduct initial capacity planning for those services. This will inform the hardware design (Chapter 7), site selection requirements (Chapter 6), and software development plan (Chapter 8).

Routing

Things to determine in this phase of the planning:

- What are the external networks with which the online application exchanges the most data in each part of the world?
- How much external network bandwidth will be needed between each PoP and the rest of the internet?
- How much backbone network bandwidth will be needed between each PoP and the application provider's other sites?

To answer these questions, the application provider can analyze either network flow data (if available) or application logs.

Proxying

Things to determine in this phase of the planning:

- Is proxying useful for the online application?
- If so, how much server capacity does each PoP need for proxying?

If, as described in Chapter 2, the client application spends a lot of time making new connections to backend applications (after fixing any low-hanging fruit in the implementation) and it is infeasible to run those applications at the edge, proxy servers in the PoPs can help.

To forecast how much server capacity is needed to run the proxies in a PoP, a reasonable starting point is:

$$\text{CPU cores needed} = (\text{rps} / \text{rps-per-core} + (100\% - \text{conn-reuse}) / \text{cps-per-core}) * (100\% + \text{redundancy-amount}) * (100\% + \text{dos-overprovisioning-amount})$$

Where:

- rps is the peak number of HTTP (or whatever other application protocol the system uses) requests that the PoP is expected to handle per second, based on the online application's usage and expected growth.
- rps-per-core is the number of HTTP (or whatever other application protocol the system uses) requests the software can process per second on one CPU core without adding significant queuing delays. This can be determined by running benchmark tests with the candidate proxy software.
- conn-reuse is the percentage of requests from clients to the proxy that reuse an existing network connection. This can be determined by instrumenting the client or backend servers.
- cps-per-core is the number of new connections per second the software can accept from clients on one CPU core without adding significant queuing delays. This number usually is orders of magnitude smaller than rps-per-core because of the cryptographic math required to set up a secure connection. This can be determined via benchmark tests.
- redundancy-amount is the fraction of the proxy servers that can be out of service at the same time (for planned or unplanned downtime) without affecting the PoP's ability to handle traffic. This is determined by the application provider's operational policies and practices. Note that redundancy planning for an edge network also should account for full-PoP outages. This may mean, for example, having enough spare capacity to survive an extended outage of any one PoP in a region.
- dos-overprovisioning-amount is the fraction of extra capacity provisioned to help survive denial of service (DoS) attacks.

It also is important to model memory usage as part of proxy capacity planning. The basic model is:

RAM needed = concurrent-conns * mem-per-conn + baseline-mem

Where:

- concurrent-conns is the maximum number of concurrent connections each proxy server is expected to handle from clients. This should be large enough to accommodate unusual events such as DoS attacks, client retry floods, and connections accumulating when the backend service becomes overloaded and slow.[10]
- mem-per-conn is the amount of memory used by each connection. This can be measured during benchmark testing. Note that it includes both kernel and userspace memory usage.
- baseline-mem is the amount of memory needed to run the server and the proxy software at idle with no connections.

Caching

Things to determine in this phase of the planning:

- Is edge caching useful for the online application?
- If so, how much server capacity does each PoP need for its cache?

If large numbers of clients fetch the exact same content from the backend systems, it probably will help to add a caching service at the edge. The complicated part is determining how big the cache needs to be: can it fit in RAM on the same servers where the proxies run, for example, or does it need dedicated servers and/or flash storage?

A good way to forecast the necessary cache size is to write a cache simulator program. The simulator implements an index of cached objects' keys in memory, keeps track of the total size of the cached objects' values (without actually storing those values)[11], and evicts keys from the index based on an algorithm such as Least Recently Used

[10] In addition, the proxy should use defensive measures such as circuit breakers and rate-limiting to limit its exposure to these events. Chapter 8 discusses proxy implementation guidelines.

[11] A real cache implementation often will use additional space per cached object to achieve other optimizations. For example, many RAM-based caches pre-divide the available memory into chunks of various fixed sizes and store each object in the available smallest chunk whose size is greater than or equal to that of the object. A cache simulator program can model this space overhead as needed.

(LRU)[12] when the simulated storage space is exhausted. The input to the simulator should be a log of the online application's requests from real clients, filtered by geography to approximate the request stream that the cache will see in a specific PoP. The output of the simulator is the cache hit ratio for that combination of request pattern, eviction algorithm, and (simulated) storage size. Repeating the process for different simulated storage sizes will produce a set of data points similar to Figure 10.

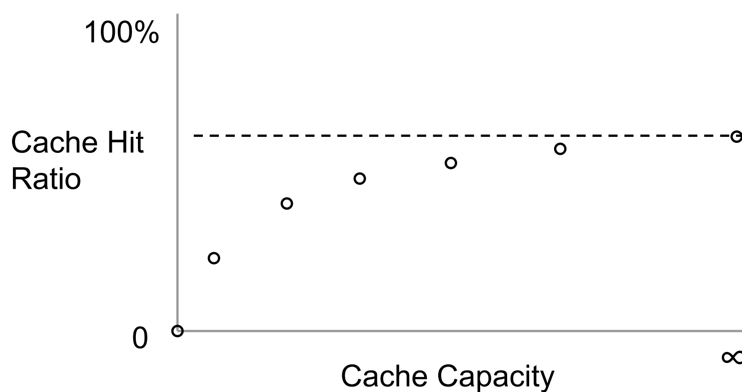


Figure 10: Example of cache size simulation results

The horizontal dotted line in this chart represents the maximum possible cache hit ratio for the logs being replayed. This often is well below 100%, especially for social applications where users constantly post new content. The cache hit ratio asymptotically approaches this line as the capacity increases.

Based on the simulation results, the team planning the edge network can choose a cache size. This is a subjective business decision, as there is a cost-vs-effectiveness tradeoff.

After choosing a total cache size, the next step is to map it to a server hardware configuration. The storage devices used for the cache must have enough space, of course, but they must also provide enough throughput ("random IOPS," in storage engineers' lingo) to keep up with the expected request workload. Flash drives often end up being a good fit, as RAM is very fast but has a high cost per Gigabyte, and rotating disks have a low cost per GB but a very low performance on random read and write workloads. Some designs use a tiered cache, with the most frequently-requested content in RAM and the long tail in flash.

---

[12] There also are newer algorithms that offer better cache hit ratios than LRU, such as ARC and SIEVE. A cache simulator can help choose the best algorithm for one's workload.

With flash storage, *write endurance* is a concern: each flash drive has a lifetime limit on how many bytes can be written to it, and this can be a problem for caches that are constantly overwriting old, stale content with new, popular content. It is important to choose flash drives whose lifetime write capacity can accommodate the projected usage. The cache software can help by reducing *write amplification*; Chapter 8 describes the technical details.

It is a good practice to shard the cache across many servers based on a consistent hash[13] of the cache key, both for scalability and to avoid a single point of failure. If possible, the cache should be sized so that if any one server rack fails or is offline for maintenance, the remaining servers have sufficient compute and network capacity to to serve the full cache workload.

Miscellaneous Application Features

This analysis is naturally application-specific, but there are a few criteria that can help determine whether a backend service is a candidate to host at the edge:

1. First, can the service operate without having to make many synchronous calls to backend datacenters while the client is waiting? By corollary, this means that if the service operates on a writable copy of any database that is shared with other locations, the application should tolerate those writes being asynchronous and eventually consistent.

2. Next, will the service fit in the PoPs? Space for servers and storage tends to be less plentiful and more expensive in edge colo datacenters than in backend datacenters.

3. Will a PoP satisfy the security requirements for the service? The application provider organization may find, for example, that the locations where they are comfortable putting network equipment are not necessarily locations where they are comfortable putting a copy of their user database. Chapter 6 discusses some of the security considerations for site selection.

[13] There are several commonly used algorithms, including the traditional ring-based consistent hash, rendezvous hash, and maglev hash. See, for example, Consistent Hashing: Algorithmic Tradeoffs for a survey of different algorithms' pros and cons.

For many online applications, a PoP provisioned to handle peak user demand will have substantial idle server capacity during off-peak hours. In some cases, it is possible to use this spare capacity to run batch workloads such as analytics, depending on the cost and feasibility of copying the needed data to the PoPs each day.

6. PoP Site Selection

One of the biggest decisions when creating or expanding an edge network is where to put the PoPs.

Usage metrics, along with the RUM data discussed in Chapter 4, can provide a coarse-grained starting point: a list of geographies where the online application has a large concentration of users experiencing poor network performance.

The critical next step is for the application provider organization to filter this list:

- Legal review: Determine whether the candidate locations are in countries where the organization is allowed to operate, and where the laws adequately protect the provider's equipment and the users' data.
- Financial review: Determine the cost implications of the candidate locations, including tariffs on importing equipment and any applicable taxes on revenue.
- Engineering review: Determine whether the candidate locations have colo datacenters with diverse Internet connectivity.

Sometimes, based on this review, the easiest way to support a large user community in a country will be to put a PoP in another country nearby.

Once candidate countries are vetted, the search for a colo datacenter can proceed based on a technical and business evaluation, including:

- Connectivity: Of the top ISPs used by the online application's users in the region, how many can be reached via network peering at each candidate colo site?[14] Do multiple, competing vendors of IP transit (Chapter 7) offer their services at the site?
- Capacity: Does each candidate colo site have enough space, power, and cooling for the application provider's PoP? This question is usually phrased as, "can this facility provide N racks of space for our equipment that uses M kilowatts of power per rack?" Also, is there

[14] PeeringDB is a useful resource for this.

space for the application provider to keep spare parts on hand: network optics, replacement drives drives for servers, etc?

- Physical plant: Does each candidate site have redundant power and cooling? Is the physical security sufficient? Will it be convenient to ship equipment to the facility?
- Support: Does each candidate site offer a “remote hands” service for any maintenance or troubleshooting work that requires physical access to the service provider’s equipment? Is onsite support available 24x7?
- Commercial considerations: How much does each candidate colo site cost? Are the contract terms suitable? Does the colo provider have experience and a good track record? Will a relationship with the colo provider be beneficial when expanding into additional regions in the future?

If the PoP needs a large amount of server capacity, the application provider may face a dilemma: colo facilities that are great for network peering are in high demand, so space there is scarce and expensive. When this happens, one solution is to *disaggregate* the PoP design: put the network routers in a colo site that is good for connectivity, put the servers in another datacenter nearby that has more and cheaper capacity, and connect the two with a metro optical network over dark fiber. Chapter 7 discusses this option in more detail.

7. Network Connectivity

Figure 11 shows the network equipment and connections typically found in a PoP.

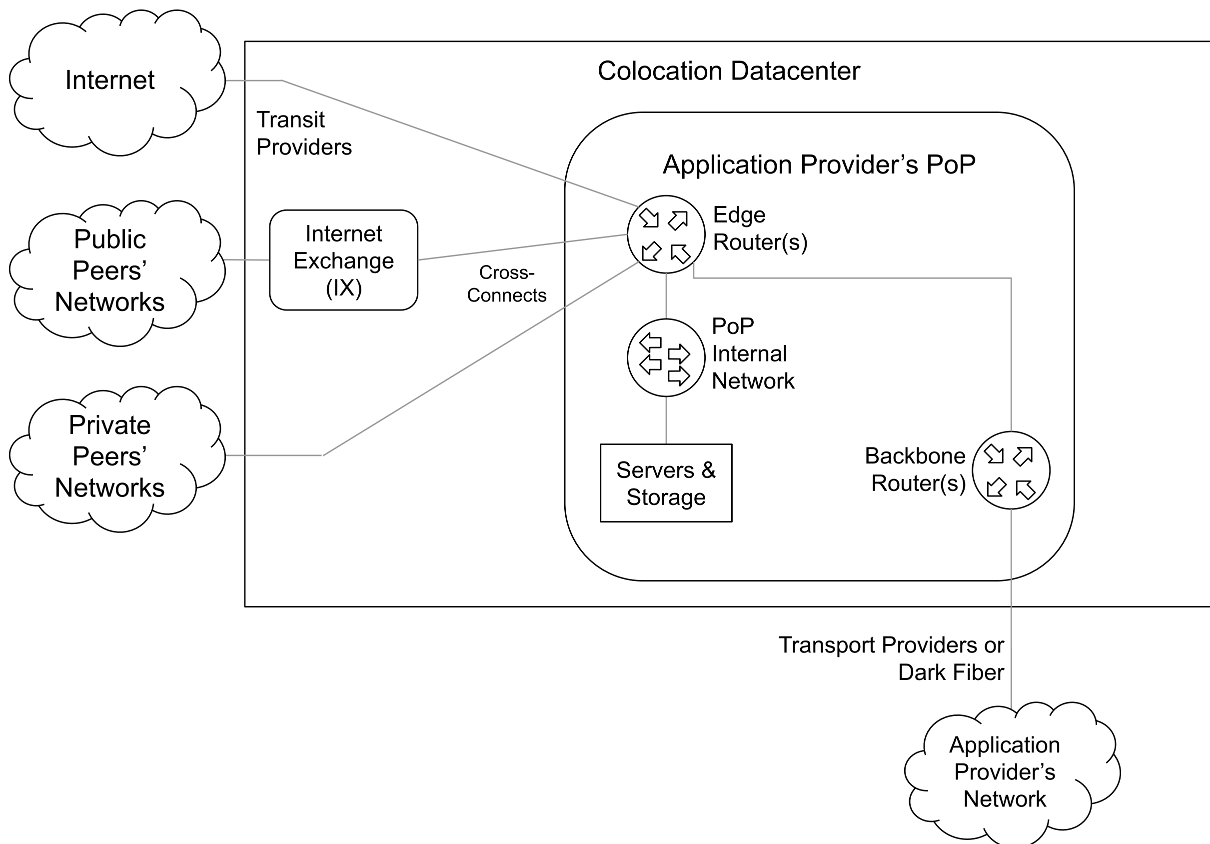


Figure 11: PoP networking overview

Connecting to the Internet

The fundamental piece of network equipment that connects a PoP to the rest of the Internet is the *edge router*. There may be multiple of these in a PoP, for redundancy. The fundamental characteristics of an edge router are that it speaks the BGP protocol to exchange routing information with other systems, it can keep track of millions of routes to the rest of the Internet[15] and select the right route for each packet at wire speed, and it has a large (but finite) number of network ports.

An online application’s clients typically are scattered among thousands of different external networks. In addition to talking to all those users’ ISPs, the application’s backend systems may also need to exchange a lot of data with external services like payment providers. The challenge when building a PoP is to obtain connectivity to all of those networks with high performance and reliability while controlling cost. The two

[15] Measuring BGP in 2023 - Have We Reached Peak IPv4?, APNIC, 2024

major cost components are the price of connectivity itself (monthly recurring OpEx) and the ports on the edge routers (a scarce capital resource).

There are a few different types of connectivity available, and it is common for an application provider to use a mix of all of them: *transit*, *private peering*, and *public peering*.

Transit

IP Transit is a service sold by third parties who operate large global or regional networks. The application provider leases one or more links from a transit supplier and connects these links to their edge routers. The transit links normally provide connectivity to and from the entire public Internet.

The normal pricing model for these links is based on peak utilization in both directions: the transit provider measures the peak bandwidth used during every 5-minute interval of the month, takes the 95th percentile largest measurement, and bills based on that amount.

Transit is good for coverage: with a transit link plugged into a single router port, the application provider can talk to the entire public Internet. The drawbacks of transit are the cost and the lack of speed guarantees.

Private Peering

Private peering connections are links from the online application provider directly to some other network (called the *peer*), which might be consumer ISP whose customers use the online application, or a SaaS provider used by the application's backend systems.

Private peering is often *settlement free*, meaning that neither side bills the other for using the link. Each party benefits financially by not having to pay a transit provider in the middle, and technologically by gaining a more direct network path to the peer. Some major consumer ISPs, however, offer only paid peering; i.e., they will let application providers connect directly to them in exchange for a utilization fee. The value proposition for the application provider in such cases is that the paid peering, while not necessarily cheaper than transit, offers a faster and more reliable way to reach the ISP's users.

Each connection to a private peer uses at least one router port, and possible multiple ports for capacity or redundancy. Therefore, private peering makes sense for for peers with whom the application provider exchanges enough network traffic to justify the dedicated port(s).

If the application uses third-party cloud providers, the PoP can be a good place to connect to those clouds' networks. Most cloud operators offer some type of direct-connection service that is similar to private peering, but with the added feature of being able to route between the private, internal address spaces of the application provider's on-premises and cloud-hosted networks, for internal server-to-server communication.

Public Peering

With private peering as a good solution for talking to high-traffic peers, and transit to cover the long tail of low-traffic networks and networks who do not have a presence at the same colo as the application provider's PoP, there is a space in the middle that is served by *Internet Exchange* (*IX*) providers.

An IX is a connectivity service operated by a third party with a presence in the colo, or sometimes provided as a service by the colo datacenter provider. Networks that have PoPs in the colo can connect links from their edge routers to a router provided by the IX. Any of these networks can then establish peering with any of the others.[16]

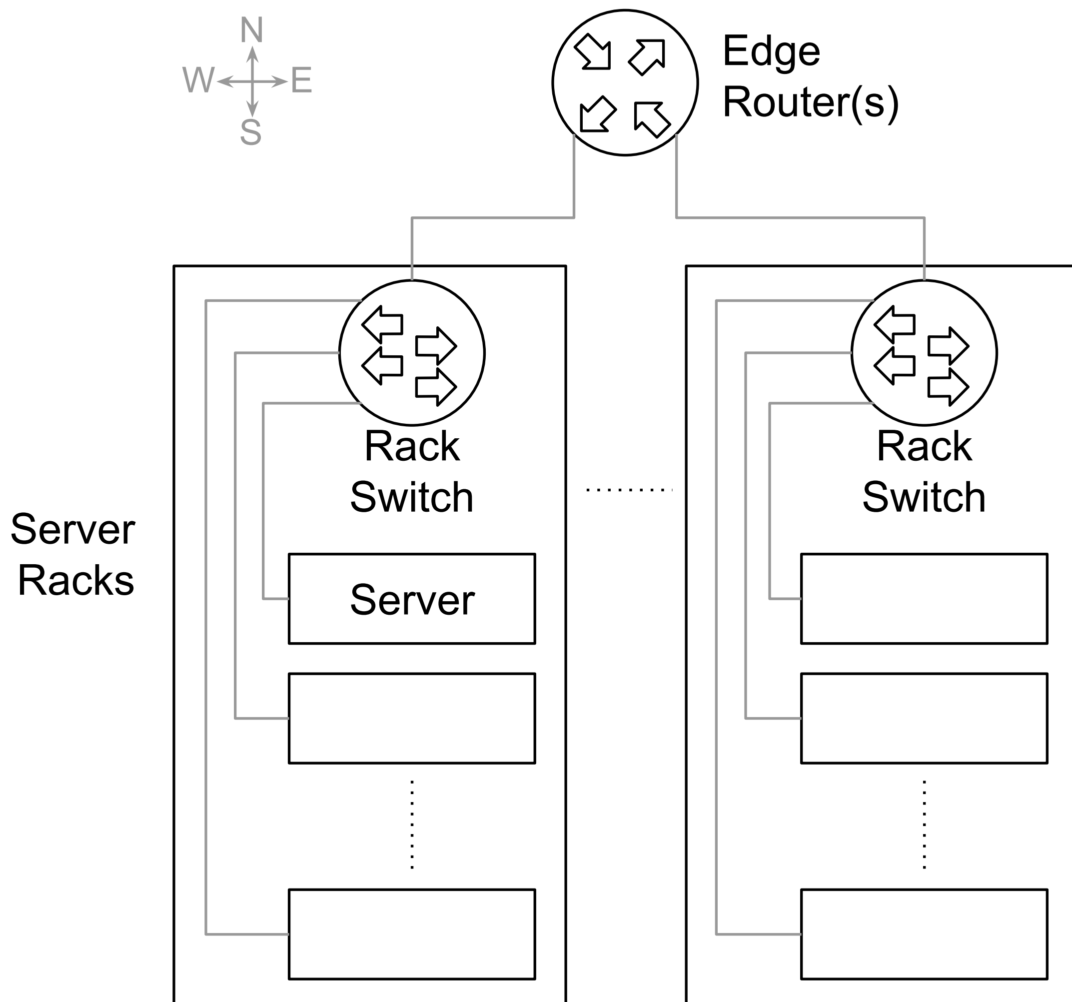
IX pricing typically is a monthly fixed fee per connected port.

Other Connectivity Considerations

The physical connection from the application provider's routers to a transit provider, IX, or private peer located in the same colo datacenter is usually a fiber *cross-connect* that the colo provider sets up and maintains for a monthly fee.

It is useful to evaluate the total cost of ownership when comparing connectivity options. For example, settlement-free peering is free in the sense that there is no charge for transferring data, but the cross-connects add a recurring cost, and the router ports have depreciation and a support contract. These fixed costs mean that the links become

[16] In network engineering terms, this can be either traditional, bilateral peering where the two networks establish a BGP session, or multilateral peering via a route server.

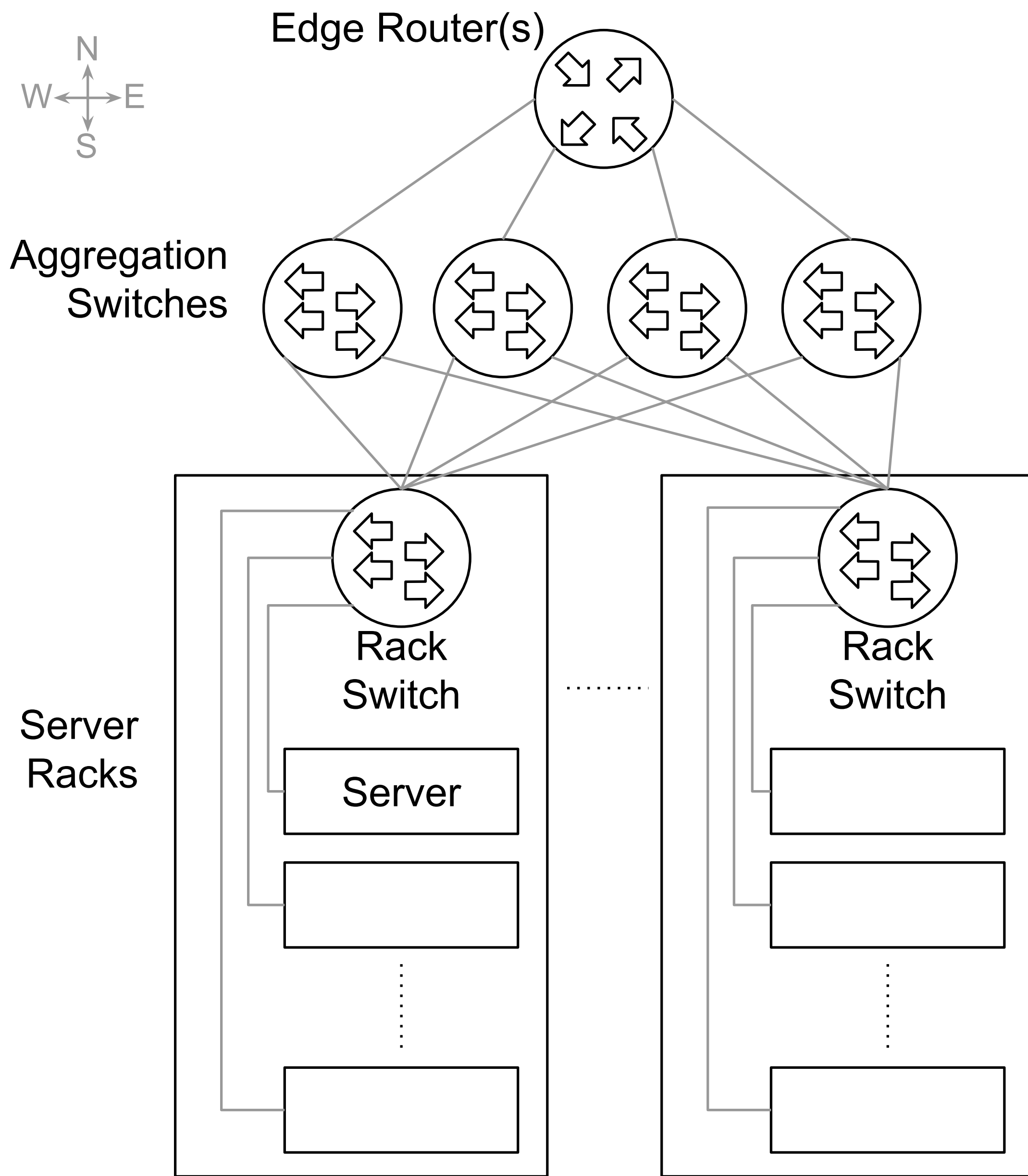


more cost-effective as their utilization increases. Dividing the monthly peak bandwidth usage for each link by the monthly cost will provide an effective unit cost of bandwidth that can be compared across transit, private peering, and IX links.

After establishing a private peering link, it is important for the application provider to tune its traffic steering (Chapter 9) to maintain a healthy peak utilization on the link. The peer has committed to devote their scarce resources — router ports, people's time — to the link and will be disappointed if the bulk of their communication with the application provider still flows through transit or shifts to some other PoP.

Intra-PoP Connectivity

If the PoP contains servers, it needs an internal network to connect the server racks to each other and to the edge routers. The requirements for this network depend in large part on the software services selected to run in the PoP. Proxying uses almost exclusively *north-south* bandwidth: the proxy servers talk mostly to things outside the PoP (clients on one side of the proxy, backend datacenters on the other) but exchange little or no data with other servers in the same PoP. However, if the proxy servers also talk to a distributed cache located in the PoP,



they need significant bandwidth *east-west*, to the other server racks in the same facility.

If the servers' do not have much east-west communication, it is possible to use a minimalist topology by just connecting the rack switches' uplinks directly to the edge router (Figure 12). This has the advantages of simplicity and low cost, but it consumes a quantity of router ports proportional to the number of server racks.

Figure 12: Simplest topology for PoPs with mostly north-south traffic

If the services running in the PoP need more east-west bandwidth, or if the traffic pattern is mostly north-south but with a large number of server racks, a better approach is to add a switching layer in between the rack switches and the edge router. This intermediate layer can be as simple as two or four aggregation switches to which the rack switches and edge routers connect (Figure 13). If the PoP is very large, this layer can be a Clos fabric.

Figure 13: 4-post aggregation switch tier to handle east-west traffic

Figure 14 shows a variant where the PoP is disaggregated into a peering site and a compute site, with metro optical links connecting the two.

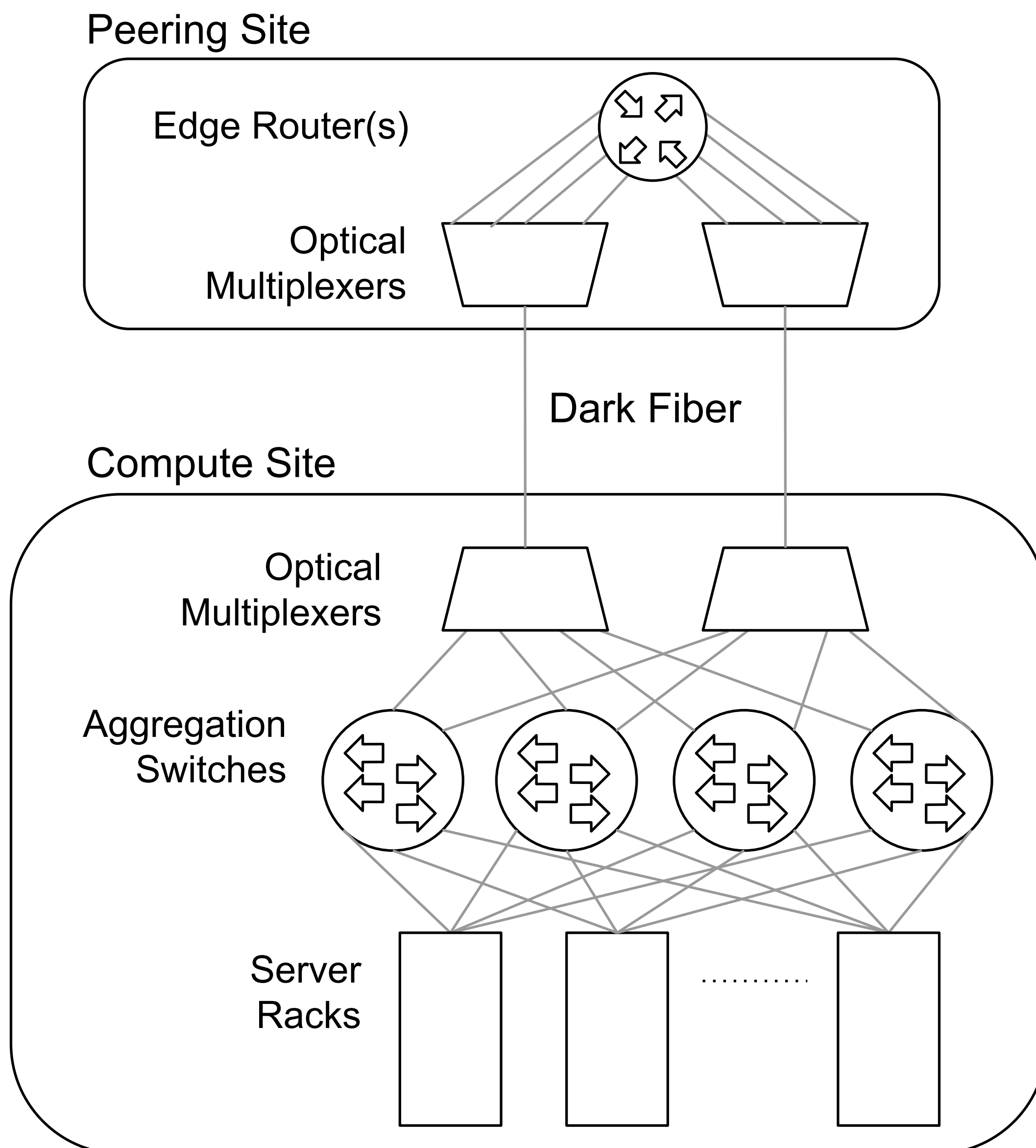


Figure 14: Disaggregated PoP networking

Connecting to the Rest of the Application Provider's Infrastructure

There are several options for connecting the PoP to the application provider's backend datacenters (and, if needed, other PoPs[17]) :

- If the traffic volume between the edge and the backend datacenter is very small, an IPsec tunnel over IP transit may suffice.

[17] One scenario where it is useful for the application provider's backbone network to provide fast PoP-to-PoP connectivity is when the traffic steering (Chapter 9) sends the client to talk to an edge service in one PoP but the best connection to the client's ISP is through another PoP. This can happen frequently if the service is something stateful, like a chat server.

- A more common solution is to rent an *IP transport* circuit from a third party provider. Such circuits provide a contracted amount of network bandwidth between two points for a fixed monthly fee.
- At larger traffic volumes, it is feasible for the application provider to lease dark fiber rather than transport links. This requires an initial purchase of optical transceiver equipment for both ends of the link, but the recurring cost of the dark fiber is lower than a transport circuit.

Note that Figure 11 shows a dedicated *backbone router* handling the communication with the application provider's other sites. It also is possible for the edge router to handle the backbone routing, combining both functions in one box.

Additional Optimizations

Between the client and the PoP, packet sizes are limited in practice to something smaller than 1500 bytes. Most clients announce that they can handle packets up to that size, but sometimes there are tunnels in the middle (for a VPN, for example, or to carry IPv4 traffic through an IPv6 network) that further limit the usable packet size. It is a good practice for Internet-facing servers to be conservative about packet sizes.

For communication between servers in the PoP, however, it is advantageous to use larger packets. Most servers and network switches can be configured to support "jumbo frames," providing a maximum packet size of approximately 9000 bytes. On the servers, the CPU cost of processing incoming or outgoing network data is mostly a factor of the number of packets sent, not the number of bytes. Thus, for large data transfers between servers, being able to split the data into fewer packets frees up CPU capacity.

If the application provider controls the backbone network between the PoP and backend datacenters, the servers in the PoP also can use large packets to talk to backend services. In addition, the backbone operator can use technologies such as MPLS to reserve bandwidth for critical applications and near-instantly reroute around failed links.

8. Edge Software Services

The development of edge software services is a broad topic, and the details depend heavily on the application provider's specific requirements, capabilities, and choice of technology stack. This chapter offers design recommendations that have proven useful for many application providers.

Prerequisites

The drawback of running software at the edge is that the deployment complexity grows proportionately to the number of PoPs. It is essential for the application provider organization to invest in automation to keep the edge software manageable. In particular, deployment, configuration management, and monitoring processes all need to scale painlessly as the PoP count grows.

Proxying

There are multiple open source HTTPS reverse proxies that work well as edge proxies, and some of them also are available in "enterprise" versions with support contracts from commercial vendors. It also is possible to write one's own edge proxy software if needed (for example, if the application uses a custom network protocol).

Load Balancing

The servers running the proxy software in a PoP should be load-balanced by a layer 4 load balancer. While it is technically possible to eliminate the load balancer and let the PoP's network switches distribute the load directly to the proxy servers by hashing the incoming packets' source addresses[18], there are two major disadvantages to that approach:

- If different racks in the PoP contain different numbers of proxy instances, but the aggregation switch layer (as described in Chapter 7) hashes an equal number of incoming traffic flows to each rack, the proxy servers will end up unevenly loaded.

[18] The common way to do this is to configure the servers to establish BGP sessions with their rack switches and advertise themselves as the next hop for the proxy VIP, configure the rack switches to share an aggregated advertisement upward to the switches or routers above them, and enable ECMP based on a 5-tuple hash at each layer.

- When proxy instances are added or removed, the network will not necessarily ensure that new incoming packets on existing connections go to the right instance.

Layer 4 load balancers solve both those problems.

It is possible for the edge proxies themselves to do load balancing among the servers in the backend datacenters to which they proxy requests. However, to reduce configuration complexity, it is better to have the proxies talk to a VIP in each backend datacenter, with a layer 4 or 7 load balancer listening on the VIP and distributing the workload among the servers there.

Security

Edge proxies are, by definition, reachable from the Internet. This makes them prominent targets for attackers. A thorough coverage of network and server security is outside the scope of this playbook, but the following practices are a good starting point for Internet-facing proxy servers.

- Harden the proxy servers, as they contain sensitive configuration data including the private keys for TLS certificates. If possible, do not store any static secrets such as private keys unencrypted in the filesystem. Remove any software packages that are not needed for the operation and management of the proxies. Strictly limit the set of user and role accounts that can log into the proxy servers. Stay up-to-date with security patches for all the layers of software on the servers.

- Consider disabling the generation of core files for the proxy processes, because they contain the transiently decrypted contents of messages passing between clients and backend services. (Doing this will complicate crash debugging, of course.)

- Always encrypt communications between physical sites, no matter how “dedicated” or “private” the links between them are. In addition, ensure that the software does proper certificate verification when establishing connections to remote sites, to prevent man-in-the-middle attacks. Ideally, use mutual TLS (mTLS) — i.e., have the proxy and backend server both use certificates to prove their identity to each other.

- Configure strict firewall rules in both directions. Not only should incoming connections be restricted to the expected protocols and ports on the proxy servers, but processes running on the proxy hosts should only be able to establish outbound connections to a short list of expected destinations.
- Automate the configuration and monitoring of servers and networks to avoid inconsistencies.
- Employ a defense-in-depth strategy, with multiple layers of security in case any one layer is compromised. For example, set up host-level firewall rules on the proxy servers and firewall rules on the edge routers, but also configure firewall rules in the backend datacenters to keep the proxies from talking to unintended internal destinations.
- To defend against DDoS attacks, deploy abundant proxy server capacity. This sometimes means significantly overprovisioning, compared to the peak legitimate request load. Implement rate-limiting per client and, for HTTP-based protocols, per requested URL. For this rate-limiting to be effective, it usually needs to share state asynchronously between proxy servers, lest a clever attacker sneak "under the radar" by spreading requests across a large number of servers and staying just under each one's individual throttling threshold.

Performance

To provide a speedup, proxy servers need to reuse existing connections to the backend datacenter. Proxy implementations usually maintain pools of idle connections for this purpose. The application provider running edge proxies should track the fraction of requests from clients that have to wait for the establishment of a new backend connection; this fraction should be as close to zero as possible.

There are many server settings that must be tuned to ensure optimal proxy performance. For example, on some operating systems, a TCP connection that has been idle for a short time will fall back into slow start mode. This default can be overridden to enable the proxy to send and receive data more aggressively on reused connections. The operating system's default buffer sizes and choice of congestion control algorithm also may need to be overridden for a high-throughput proxy.

Teams managing proxy servers should create a set of OS performance tunings as part of their automated server configuration management.[19]

Caching

There are a few open source web cache implementations currently available. Some application providers develop their own cache software.

Recommendations for developers of caches:

- Use a consistent hash function on some key (e.g., the URL of the cached object) to shard across nodes. Note that a cache distributed across server racks will need ample east-west bandwidth; Chapter 7 discusses how to design the PoP network hardware for this.
- For a cache in flash storage, the big complication is *write endurance*. Each flash drive has a lifetime maximum number of bytes that can be written to it; after that, the drive becomes read-only. A typical cache, when confronted with an ever-changing set of popular content, will do a lot of writes to replace cold content with hot content. The cache simulation approach described in Chapter 5 can provide an estimate of the needed write endurance: assume each cache miss requires one write to storage, add up the object sizes for all the cache misses over the simulation, and extrapolate to an N-year period, where N is the planned capital depreciation lifetime of the flash drive.
- The write endurance problem is exacerbated by *write amplification*. A straightforward LRU cache implementation in flash will spend a lot of time overwriting small objects in random locations on the flash drive. Internally, the flash storage is divided into large blocks, and modifying even one byte within a block means erasing and rewriting the entire contents of the block. When the cache software overwrites N bytes in flash storage, the flash ends up writing $N * A$ bytes internally. A, the write amplification factor, depends on the specific access pattern, but it is always greater than or equal to one. And it is $N * A$, rather than N, that is subtracted from the drive's aforementioned lifetime write limit. The controllers in flash drives try to reduce write amplification by over-provisioning, managing a pool of free space, and buffering writes; but a cache that does a lot of small, randomly distributed

[19] A good starting point is Dropbox's blog post on Optimizing Web Servers for High Throughput and Low Latency.

overwrites will still have a large value of A. The cache developer can help reduce write amplification by designing a cache layout that minimizes random writes using, for example, a journaling design.[20]

- If objects in the cache can expire, this can result in a thundering herd of requests to the backend systems when hundreds of clients all try to fetch the same popular object that has just expired. This can be prevented by building a synchronization mechanism within the cache.
- For many online applications, cache deletion is a critical use case. For example, applications that support user-generated content need a way for moderators to delete content that violates the application's rules. With a distributed cache based on consistent hashing, it is possible that copies of a given object will end up in multiple cache nodes due to servers coming in and out of service. Therefore a good practice when deleting is to send the deletion request to every cache server, rather than just the server that the hash currently selects. There still can be problems if a cache server containing the object is offline at the time of deletion and revives the object when it comes back to life. One option for handling that case is to maintain a log of deletions and replay it against nodes that are being brought back into service. Another is to give each object inserted into the cache an expiration time N seconds in the future, thus incurring more frequent cache misses in exchange for a deletion safeguard.
- Finally, some applications will benefit from placing a middle tier of cache in between the backend systems and the edge, so that cache misses at the edge have a chance to be served from, say, a regional cache in each continent rather than traveling all the way to the backend datacenters. Simulations can help determine if this approach is useful for a specific application's access pattern.

[20] Meta's Reduced Insertion Point Queue is an example of a flash cache layout that supports LRU semantics with low write amplification. If the cache software uses such a layout to manage its storage in an append-only manner (with erasure of full blocks to reclaim space when the cache fills up), it can use the Zoned Namespace (ZNS) feature of compatible flash drives to take more direct control of the flash usage. That will reducing the need for over-provisioning, allowing more of the flash space to be used for cached data.

9. Traffic Steering

Previous chapters mentioned that clients would be sent to edge services in the nearest PoP via some unspecified magic. The technology for doing this is called *traffic steering*, and there are two main variants: DNS and anycast routing.

DNS

With DNS traffic steering, an edge service has a different IP address in each PoP. For illustrative purposes, assume that the edge service has the domain name app.example.com and is hosted in the following locations:[21]

PoP	Edge service IPv6 address
Seattle	2001:DB8:0001::10
Frankfurt	2001:DB8:0002::10
Singapore	2001:DB8:0003::10

To find the edge service, the client normally sends a DNS query for app.example.com to some intermediate DNS server. This intermediate server might be one run by the client's ISP, or perhaps a free public DNS service. Assuming that this intermediate DNS server does not have the answer in cache, it forwards the query to the DNS server for example.com.

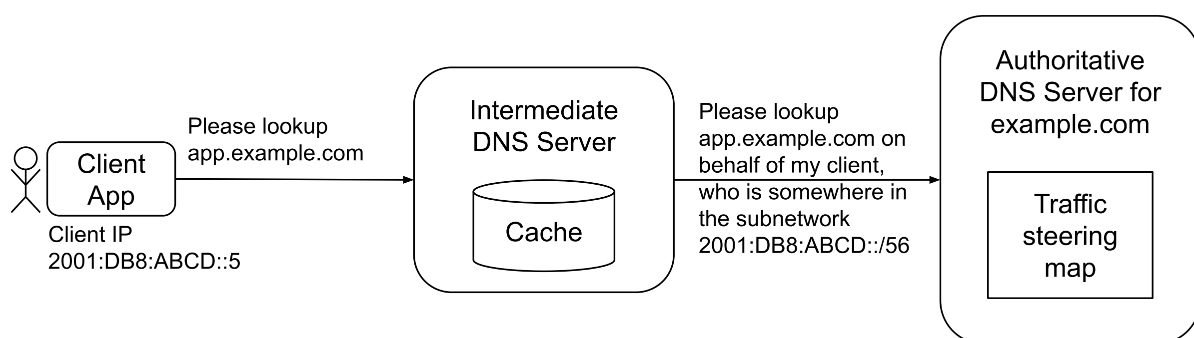


Figure 15: DNS-based traffic steering

The DNS server for example.com, called an *authoritative server*, is configured to return different answers for app.example.com, depending

[21] For simplicity, this example shows only IPv6 addresses. In common practice, the service will have both IPv4 and IPv6 addresses, and clients that support both will use the Happy Eyeballs algorithm from RFC 8305 to choose which one to use.

on where the client is. The authoritative DNS server looks at the source IP address for the request, determines which PoP is the best one to serve requests coming from that location, and returns the IP address of the service in that PoP. For example, if the DNS request has originated in Europe, the authoritative DNS server might choose to return the address of the service in Frankfurt, 2001:DB8:0002::10. And if the DNS request has originated in North America, the authoritative DNS server might choose to return the address of the service in Seattle, 2001:DB8:0001::10. To keep the lookups fast, the authoritative server often will use a precomputed table that maps client IP prefixes to the preferred PoP for each.

While this process sounds straightforward, there are complications that must be solved in the design of the authoritative DNS server:

- dealing with intermediaries that obscure the client's location,
- defining how exactly to choose the "best" PoP for a client,
- and adapting to capacity and availability limitations.

DNS Intermediaries

In figure 15, note that the query arriving at the authoritative DNS server comes from the intermediate DNS server, rather than directly from the client. If the authoritative server tries to choose a PoP based on the source address of the query, it will end up with a PoP close to the intermediate server — which might not be anywhere near the client.

To solve that problem, intermediate DNS servers are allowed, although not required, to pass along information about the original DNS client's location. The mechanism for this is the EDNS Client Subnet extension, documented in RFC 7871. An intermediate DNS server can encode its client's IP address as an extra field in the DNS request that it sends on to the next DNS server. For privacy purposes, the intermediate resolver sends only the first 24 bits of the client's IPv4 address, or the first 56 bits for IPv6.[22] Note, though, that some operators of intermediate DNS servers, including at least one major public DNS service[23], take an even stricter privacy stance by not sending the Client Subnet extension at all.

[22] To enable DNS traffic steering to interoperate with DNS caching intermediate servers that send the EDNS Client Subnet in forwarded requests also use the subnet value as part of the cache key when caching the responses.

[23] 1.1.1.1 DNS Resolver FAQ, Cloudflare.

Choosing the Best PoP for a Client

If the application provider has implemented a RUM data collection system, as described in Chapter 4, that data can be used to build a mapping from client IP prefixes to preferred PoPs.

In practice, there are other criteria in addition to network performance that affect the preferred choice of PoP. For example, if two PoPs offer similar performance for a client, but the client's ISP peers with the application provider at only one of those PoPs, that one is preferable. Or if the PoP that ranks as the fastest based on the RUM data has just come back online after an extended outage and its caches are empty, the application operator may want to readmit traffic slowly.

A good practice is to run a continuous, offline map-building process that applies such business rules to compute the currently preferred destination for each client IP prefix, and publish the output of this process to the authoritative DNS servers once every few minutes. The authoritative DNS server can then specify a small Time-to-Live (TTL) value for its responses, to limit how long clients and intermediate DNS servers may cache the results.

Adapting to Capacity and Availability Changes

If a PoP is getting overloaded, the authoritative DNS server can send less work there. Clients already talking to an edge service in that PoP may be stuck there, but new client sessions will be directed to other PoPs by the time the TTL has expired. In practice, not all clients and intermediaries on the Internet pay attention to DNS TTLs, so fully draining the long tail of traffic from a PoP often takes longer than expected.

If a PoP suddenly goes offline — for example, because of a power failure or software bug — the authoritative DNS server can stop including that PoP in DNS answers, but new clients will continue being steered there for at least the TTL. For services that want a reduced disruption in this situation, one option is to have each PoP's service VIPs ready to turn on at some other location, so that DNS clients who learn the VIPs before the TTL expires have a place to land.[24] Note, though, that clients who

[24] Note for network engineers: it is possible to automate this failover by having a backup site advertise a shorter prefix that contains the prefix used by the PoP's VIPs. While the PoP is alive, its longer, more specific prefix will be the preferred destination.

already had connections to the failed PoP will experience errors when new packets on those connections get routed to a new location that is not expecting them.

Anycast Routing

The common alternative to DNS traffic steering is anycast routing. In the anycast routing design, a given edge service has the same public, virtual IP address (VIP) in many PoPs. The client talks to that shared VIP, and the packets end up at the closest PoP, based on the network's notion of "closest."

Building upon the example from the DNS section, we add a VIP to the edge service in all locations. (It is useful to also give the edge service a unique IP address in each PoP, for use in monitoring and troubleshooting.)

PoP	Edge service IPv6 address
Seattle	2001:DB8:0F00::10
Frankfurt	2001:DB8:0F00::10
Singapore	2001:DB8:0F00::10

Choosing the Best PoP for a Client

With anycast routing, the process of connecting a client to the closest PoP is automatic, although "closest" in this case basically means the smallest number different networks between the source and destination.[25] Two hops across an ocean are treated as a shorter path than three hops across town. Operators sometimes need to apply *traffic engineering*: router configuration overrides to move incoming and outgoing traffic to the desired paths.

Some organizations using anycast routing have opted for a hybrid "regional anycast" scheme in which DNS determines the client's location and returns a region-specific anycast VIP that is advertised by the PoPs in that region. This helps prevent edge cases where the fewest-hops path would have taken the traffic across an ocean and back.[26]

[25] In BGP, the routing protocol that the edge routers use to exchange routing information with the outside world, the number of Autonomous Systems between point A and point B is one of the primary criteria for choosing a route, but this can be overridden by configuring a higher or lower "local preference" value to prioritize or deprioritize certain routes.

[26] See, for example, Regional IP Anycast: Deployments, Performance, and Potentials.

An advantage of anycast routing is that it inherently aligns traffic steering with the application provider's peering relationships, in contrast to DNS-based steering where that alignment must be done in the map-building software. In addition, anycast tends to help with DDoS defense by distributing incoming attacks against multiple PoPs, in contrast to DNS-based steering where an attacker can point at a PoP-specific IP address to try to overload that site.

Adapting to Capacity and Availability Changes

A drawback of anycast traffic steering is that it operates like an on-off switch. There is not a way to shed load from a PoP by directing new client connections elsewhere while keeping existing connections pinned to the PoP. Similarly, when bringing a new PoP online, there is not a way to start routing requests to it without breaking existing connections.[27] However, it is possible to shift load among PoPs by applying traffic engineering to the route advertisements.

Choosing a Traffic Steering Strategy

DNS-based and anycast-based traffic steering both have unique advantages and challenges. Historically, DNS has offered more precise and flexible control, as well as lower latency for clients[28], although this may change in the future if more operators of intermediate DNS servers stop supporting for the EDNS Client Subnet feature for privacy reasons. Anycast makes it easy to align traffic steering with network topology and capacity — but harder to align traffic steering with non-network considerations such as PoP server capacity. Anycast also has become popular among CDN operators because of its DDoS-defense advantages.[29]

Both techniques have ardent champions and opponents. A pragmatic way for an organization to choose a steering technology is based on the expertise required. Steering via DNS requires software engineers to build data pipelines and control-loop algorithms, whereas steering via anycast requires network engineers to do traffic engineering.

---

[27] This is not a problem, of course, for applications that do not depend on long-lived client connections. For example, DNS servers themselves are often load-balanced with anycast routing.

[28] See, for example, the graphs in the "Initial results" section of Intelligent DNS based load balancing at Dropbox.

[29] Cloudflare and Cachefly, for example, have written about their use of anycast traffic steering to spread incoming DDoS attacks across a larger defensive surface area.

Some application providers add an additional step: once the primary traffic steering system chooses a PoP and the clients sends its first request, the server-side software knows with greater certainty where the client is located. If necessary, the server can then tell the client to talk to a different PoP for subsequent requests — e.g., by rewriting links to point to some hostname like app-seattle.example.com that bypasses the normal traffic steering. This technique can improve performance if the DNS or anycast steering has made a bad decision, although it causes operational headaches for web-based applications if third parties start hot-linking to PoP-specific URLs.

After implementing any traffic steering mechanism, the application provider should monitor the actual client-to-server network performance and compare it to the best possible performance. If the RUM data indicates that the closest available PoP is a 10 millisecond RTT away from a client, but the traffic steering is sending the client to a PoP 300 milliseconds away, it is time to start troubleshooting.

10. Rollout and Operation

Like any major infrastructure project, launching an edge network for the first time is a complicated and risky endeavor. Some recommendations for the initial deployment:

- Launch incrementally. Turn up one PoP and start sending a small test group of users through it. If the edge network is using anycast routing for traffic steering, it is best to use DNS to send only the test group to the anycast address while keeping the majority of users unaffected.
- Do no harm. Examine all available product and engineering metrics to ensure that nothing has broken for the users in the test group.
- Verify the wins. Use RUM data and product metrics to confirm that the PoP has delivered the expected improvements.[30]
- Scale up. Steer more users to the first PoP until it is operating at its planned workload. Turn up the rest of the PoPs. Test drains and failover between PoPs.

After launching:

- Monitor continuously. Track the performance and reliability of the edge services in all PoPs. Measure proxies' connection reuse and caches' hit rate, especially after launching changes to the application or infrastructure.
- Continue scaling. Incrementally add edge connectivity, servers, and backbone bandwidth as needed. As the edge network grows more complex, buy or build software that tracks of the utilization of every network link and predicts when to order more capacity.
- Enter into peering agreements with external networks. This is an ongoing process due to the long tail of potential peers and the time

[30] Many organizations have robust A/B testing systems that measure both expected and unexpected results of changes. These A/B systems usually identify test cohorts based on things like cookies or user IDs, whereas edge traffic steering maps users into groups based on completely different attributes such as client IP prefix. But, while difficult, it can be valuable to develop an integration between between the A/B-test and traffic-steering systems, in order to quantify edge improvements using the same methodology and tooling as all of one's other product initiatives.

needed to build connections across companies (both the technological and the interpersonal kinds of connections).

- Track costs. As described in Chapter 7, track the effective cost per unit of peak bandwidth usage for each network link. Similarly, track the cost per peak concurrent user for edge compute.
- Define criteria for expansion. After building out the initial PoP sites, how should the organization decide when/whether to expand into additional locations? For example, should the decision to build a PoP in a new geography be driven by user growth in that area, or by monetization potential, or perhaps by application speed metrics? The right answer is inherently organization-specific, but it is helpful to codify an internal decision-making framework.

Finally, after successfully building an edge network, some application providers may decide to take the next big step closer to the user: extending their edge presence into client ISPs' networks. Several companies have built "off network" server appliances that run their edge server software but are physically located in consumer ISPs' datacenters.[31] This setup can be a win for both parties. In exchange for providing the datacenter space and power for the appliances, the ISP saves on incoming bandwidth costs because the cache substantially reduces the amount of data they need to receive from outside their own network. In exchange for providing the servers and software, the the application provider gets closer to the client for better performance and also saves on outbound data transfer costs. Running server appliances off-network adds significant security and operational challenges, but for application providers operating at very large scale it can be an effective way to improve service for users.

[31] See, for example, Netflix's Open Connect Appliance and Meta's Facebook Network Appliance.