Introduction to distributed systems

Faculty Of Information Technology

Spring 2024

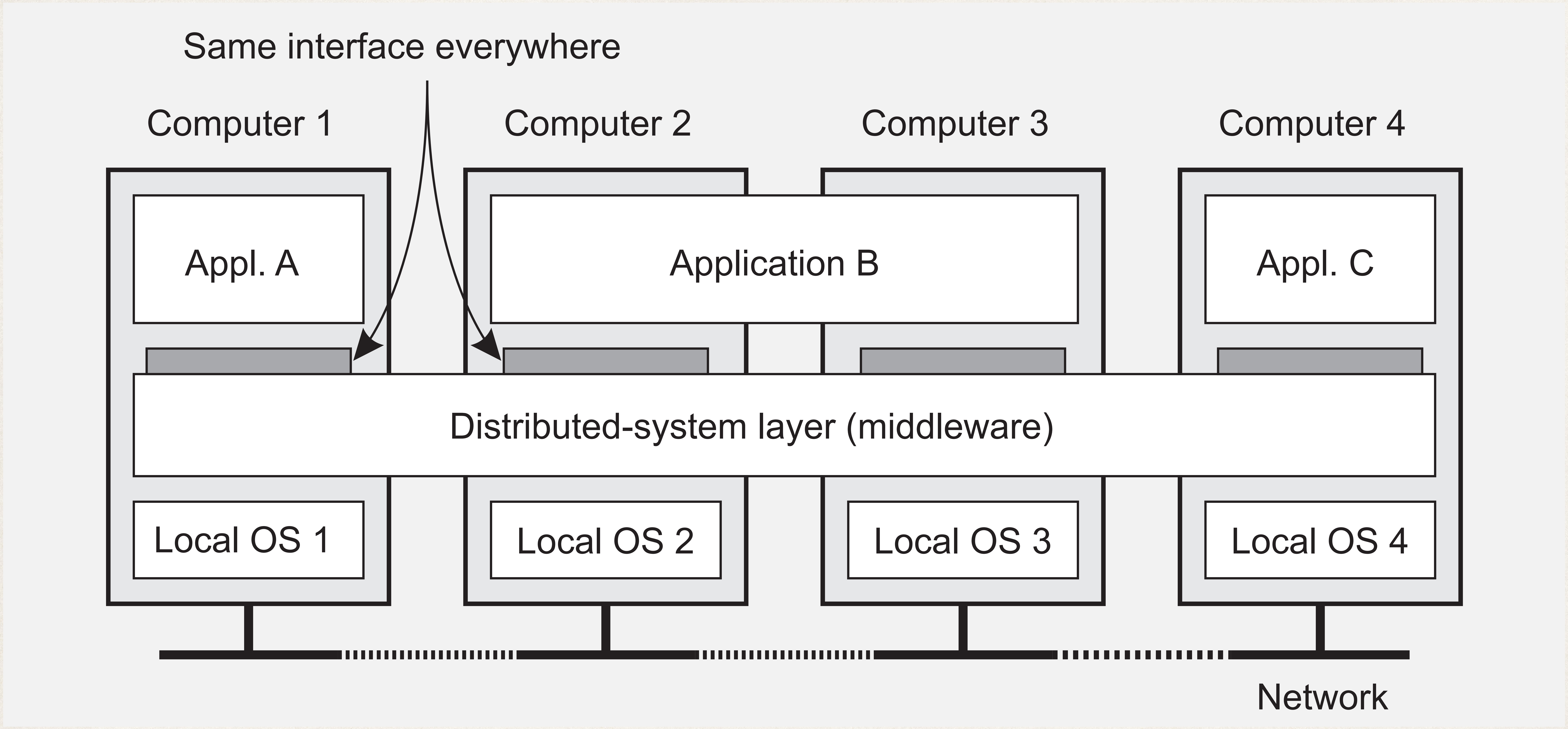
Distributed System: Definition

A distributed system is

a collection of autonomous computing elements that appears to its users as a single coherent system

Two aspects:

(1) independent computing elements and (2) single system $\Rightarrow$ middleware.


Same interface everywhere
Computer 1
Computer 2
Computer 3
Computer 4
Appl. A
Application B
Appl. C
Distributed-system layer (middleware)
Local OS 1
Local OS 2
Local OS 3
Local OS 4
Network

What You Will Learn

- Building distributed systems requires some fundamental understanding of distribution and concurrency. It's needed because of the two essential problems in distributed systems that make them complex, as described below.

  - First, although systems as a whole operate perfectly correctly nearly all the time, an individual part of the system may fail at any time. When a component fails (whether due to a hardware crash, network outage, bug in a server, etc.), we have to employ techniques that enable the system as a whole to continue operations and recover from failures. Every distributed system will experience component failure, often in weird, mysterious, and unanticipated ways.

  - Second, creating a scalable distributed system requires the coordination of multiple moving parts. Each component of the system needs to keep its part of the bargain and process requests as quickly as possible. If just one component causes requests to be delayed, the whole system may perform poorly and even eventually crash.

- Luckily for us engineers, there’s also an extensive collection of technologies that are designed to help us build distributed systems that are tolerant to failure and scalable. These technologies embody theoretical approaches and complex algorithms that are incredibly hard to build correctly. Using these platform-level, widely applicable technologies, our applications can stand on the shoulders of giants, enabling us to build sophisticated business solutions.

Specifically, will learn:

- The fundamental characteristics of distributed systems, including state management, time coordination, concurrency, communications, and coordination

- Architectural approaches and supporting technologies for building scalable, robust services

The first four Lectures, advocate the need for scalability as a key architectural attribute in modern software systems. These Lectures provide broad coverage of the basic mechanisms for achieving scalability, the fundamental characteristics of distributed systems. This knowledge lays the foundation for what follows.

- The last 20 years have seen unprecedented growth in the size, complexity, and capacity of software systems. This rate of growth is hardly likely to slow in the next 20 years—what future systems will look like is close to unimaginable right now. However, one thing we can guarantee is that more and more software systems will need to be built with constant growth—more requests, more data, and more analysis—as a primary design driver.

- Scalable is the term used in software engineering to describe software systems that can accommodate growth. In this Lecture we'll explore what precisely is meant by the ability to scale, known (not surprisingly) as scalability. we'll also describe a few examples that put hard numbers on the capabilities and characteristics of contemporary applications. Finally, we'll describe two general principles for achieving scalability, replication and optimization, and examine the link between scalability and other software architecture quality attributes.

What Is Scalability?

Intuitively, scalability is a pretty straightforward concept. If we ask Wikipedia for a definition, it tells us, “Scalability is the property of a system to handle a growing amount of work by adding resources to the system.” We all know how we scale a highway system—we add more traffic lanes so it can handle a greater number of vehicles. Think of any physical system—a transit system, an airport, elevators in a building—and how we increase capacity is pretty obvious.

Put very simply, and without getting into definition wars, scalability defines a software system's capability to handle growth in some dimension of its operations. Examples of operational dimensions are:

- The number of simultaneous user or external (e.g., sensor) requests a system can process
- The amount of data a system can effectively process and manage
- The value that can be derived from the data a system stores through predictive analytics
- The ability to maintain a stable, consistent response time as the request load grows

- Increasing a system’s capacity in some dimension by increasing resources is called scaling up or scaling out. In addition, unlike physical systems, it is often equally important to be able to scale down the capacity of a system to reduce costs.

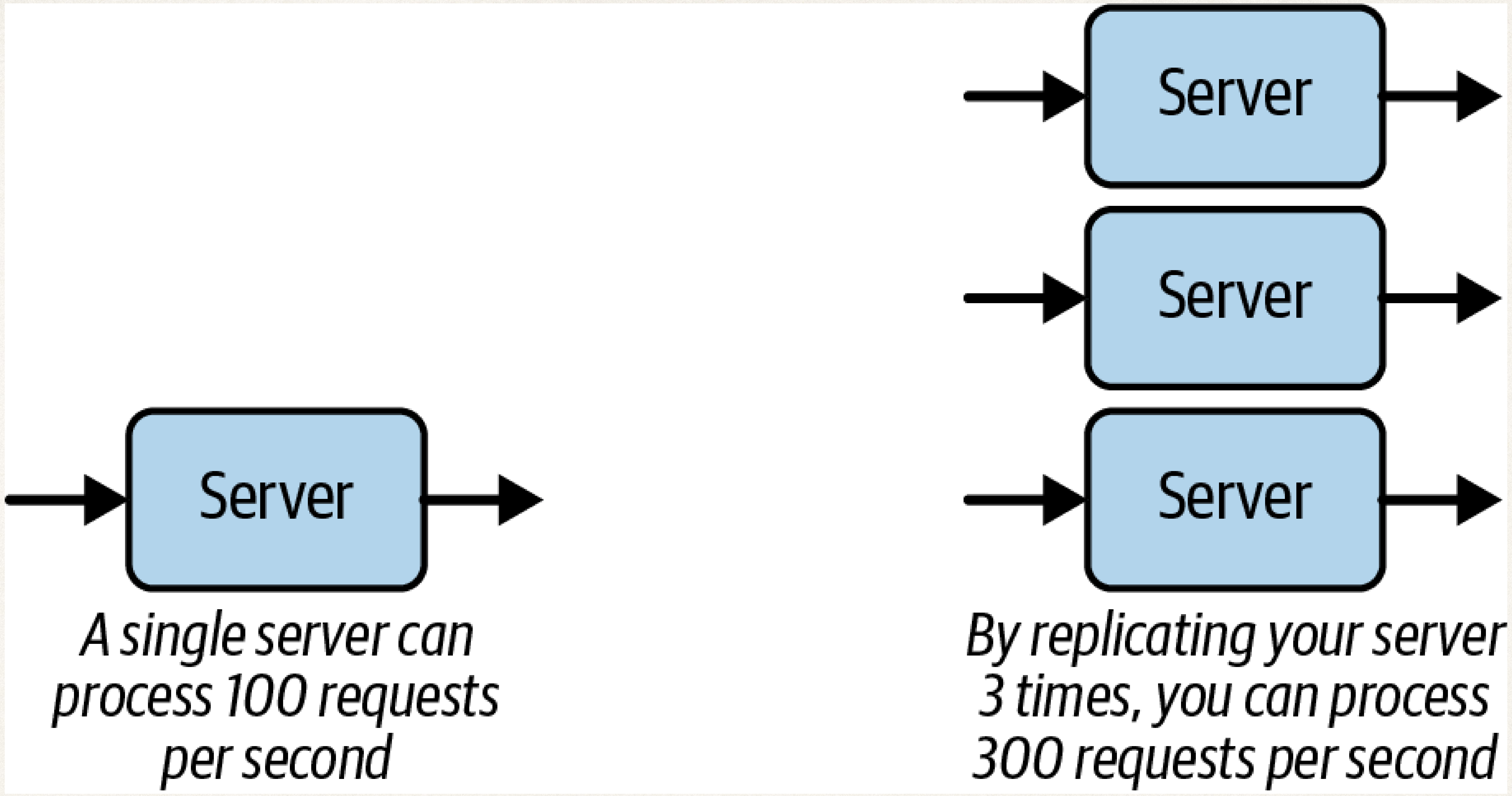
- The example of this is Netflix, which has a predictable regional daily load that it needs to process. Simply, a lot more people are watching Netflix in any geographical region at 9 p.m. than are at 5 a.m. This enables Netflix to reduce its processing resources during times of lower load. This saves the cost of running the processing nodes that are used in the Amazon cloud, as well as societally worthy things such as reducing data center power consumption.

Scalability Basic Design Principles

The basic aim of scaling a system is to increase its capacity in some application-specific dimension. A common dimension is increasing the number of requests that a system can process in a given time period. This is known as the system's throughput. Let's use an analogy to explore two basic principles we have available to us for scaling our systems and increasing throughput: replication and optimization.

We basically replicate the software processing resources to provide more capacity to handle requests and thus increase throughput, as shown in Figure below. These replicated processing resources are analogous to the traffic lanes on bridges, providing a mostly independent processing pathway for a stream of arriving requests.

Luckily, in cloud-based software systems, replication can be achieved at the click of a mouse, and we can effectively replicate our processing resources thousands of times. We have it a lot easier than bridge builders in that respect. Still, we need to take care to replicate resources in order to alleviate real bottlenecks. Adding capacity to processing paths that are not overwhelmed will add needless costs without providing scalability benefit.


Server
A single server can process 100 requests per second
Server
Server
Server
By replicating your server 3 times, you can process 300 requests per second

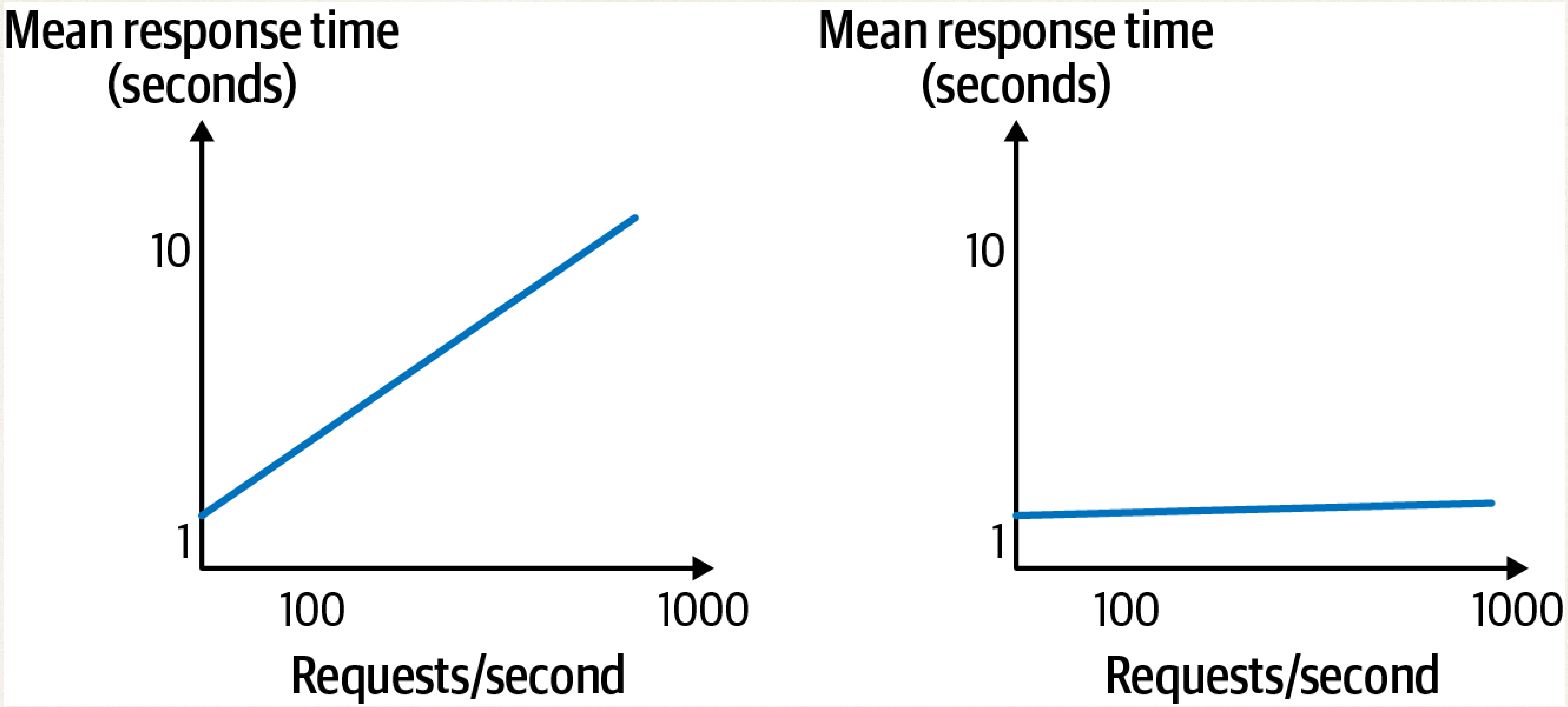
Increasing capacity through replication

The second strategy for scalability can also be illustrated with our bridge example. In Sydney, some observant person realized that in the mornings a lot more vehicles cross the bridge from north to south, and in the afternoon we see the reverse pattern. A smart solution was therefore devised—allocate more of the lanes to the high-demand direction in the morning, and sometime in the afternoon, switch this around. This effectively increased the capacity of the bridge without allocating any new resources—we optimized the resources we already had available.

We can follow this same approach in software to scale our systems. If we can somehow optimize our processing by using more efficient algorithms, adding extra indexes in our databases to speed up queries, or even rewriting our server in a faster programming language, we can increase our capacity without increasing our resources. The example of this is Facebook's creation of (the now discontinued) HipHop for PHP, which increased the speed of Facebook's web page generation by up to six times by compiling PHP code to C++.

Scalability and Costs

- Let's take a trivial hypothetical example to examine the relationship between scalability and costs. Assume we have a web-based (e.g., web server and database) system that can service a load of 100 concurrent requests with a mean response time of 1 second. We get a business requirement to scale up this system to handle 1,000 concurrent requests with the same response time. Without making any changes, a simple load test of this system reveals the performance shown in Figure below (left). As the request load increases, we see the mean response time steadily grow to 10 seconds with the projected load. Clearly this does not satisfy our requirements in its current deployment configuration. The system doesn't scale.



Scaling an application; non-scalable performance is represented on the left, and scalable performance on the right

Some engineering effort is needed in order to achieve the required performance. Figure above (right) shows the system's performance after this effort has been modified. It now provides the specified response time with 1,000 concurrent requests. And so, we have successfully scaled the system.

A major question looms, however. Namely, how much effort and resources were required to achieve this performance? Perhaps it was simply a case of running the web server on a more powerful (virtual) machine. Performing such reprovisioning on a cloud might take 30 minutes at most. Slightly more complex would be reconfiguring the system to run multiple instances of the web server to increase capacity. Again, this should be a simple, low-cost configuration change for the application, with no code changes needed. These would be excellent outcomes.

However, scaling a system isn't always so easy. The reasons for this are many and varied, but here are some possibilities:

- The database becomes less responsive with 1,000 requests per second, requiring an upgrade to a new machine.
- The web server generates a lot of content dynamically and this reduces response time under load. A possible solution is to alter the code to more efficiently generate the content, thus reducing processing time per request.
- The request load creates hotspots in the database when many requests try to access and update the same records simultaneously. This requires a schema redesign and subsequent reloading of the database, as well as code changes to the data access layer.
- The web server framework that was selected emphasized ease of development over scalability. The model it enforces means that the code simply cannot be scaled to meet the requested load requirements, and a complete rewrite is required. Use another framework? Use another programming language even?

There’s a myriad of other potential causes, but hopefully these illustrate the increasing effort that might be required as we move from possibility (1) to possibility (4).

Now let’s assume option (1), upgrading the database server, requires 15 hours of effort and a thousand dollars in extra cloud costs per month for a more powerful server. This is not prohibitively expensive. And let’s assume option (4), a rewrite of the web application layer, requires 10,000 hours of development due to implementing a new language (e.g., Java instead of Ruby). Options (2) and (3) fall somewhere in between options (1) and (4). The cost of 10,000 hours of development is seriously significant. Even worse, while the development is underway, the application may be losing market share and hence money due to its inability to satisfy client requests’ loads. These kinds of situations can cause systems and businesses to fail.

Scalability and Architecture Trade-Offs

When we focus on scaling a system, we must also consider how our design choices affect other important aspects like performance, availability, security, and manageability. These factors are often overlooked but crucial for a well-functioning system.

Performance

There’s a simple way to think about the difference between performance and scalability. When we target performance, we attempt to satisfy some desired metrics for individual requests. This might be a mean response time of less than 2 seconds, or a worst-case performance target such as the 99th percentile response time less than 3 seconds.

Improving performance is in general a good thing for scalability. If we improve the performance of individual requests, we create more capacity in our system, which helps us with scalability as we can use the unused capacity to process more requests.

However, it’s not always that simple. We may reduce response times in a number of ways. We might carefully optimize our code by, for example, removing unnecessary object copying, using a faster JSON serialization library, or even completely rewriting code in a faster programming language. These approaches optimize performance without increasing resource usage.

An alternative approach might be to optimize individual requests by keeping commonly accessed state in memory rather than writing to the database on each request. Eliminating a database access nearly always speeds things up. However, if our system maintains large amounts of state in memory for prolonged periods, we may (and in a heavily loaded system, will) have to carefully manage the number of requests our system can handle. This will likely reduce scalability as our optimization approach for individual requests uses more resources (in this case, memory) than the original solution, and thus reduces system capacity.

Availability

Availability and scalability are in general highly compatible partners. As we scale our systems through replicating resources, we create multiple instances of services that can be used to handle requests from any users. If one of our instances fails, the others remain available. The system just suffers from reduced capacity due to a failed, unavailable resource. Similar thinking holds for replicating network links, network routers, disks, and pretty much any resource in a computing system.

Things get complicated with scalability and availability when state is involved. Think of a database. If our single database server becomes overloaded, we can replicate it and send requests to either instance. This also increases availability as we can tolerate the failure of one instance. This scheme works great if our databases are read only. But as soon as we update one instance, we somehow have to figure out how and when to update the other instance. This is where the issue of replica consistency raises its ugly head.

In fact, whenever state is replicated for scalability and availability, we have to deal with consistency.

Security

Security is a complex, highly technical topic worthy of its own book. No one wants to use an insecure system, and systems that are hacked and compromise user data cause CTOs to resign, and in extreme cases, companies to fail.

The basic elements of a secure system are authentication, authorization, and integrity. We need to ensure data cannot be intercepted in transit over networks, and data at rest (persistent store) cannot be accessed by anyone who does not have permission to access that data. Basically, I don't want anyone seeing my credit card number as it is communicated between systems or stored in a company's database.

Hence, security is a necessary quality attribute for any internet-facing systems. The costs of building secure systems cannot be avoided, so let's briefly examine how these affect performance and scalability.

In general, security and scalability are opposing forces. Security necessarily introduces performance degradation. The more layers of security a system encompasses, then a greater burden is placed on performance, and hence scalability. This eventually affects the bottom line—more powerful and expensive resources are required to achieve a system's performance and scalability requirements.

Manageability

As the systems we build become more distributed and complex in their interactions, their management and operations come to the fore. We need to pay attention to ensuring every component is operating as expected, and the performance is continuing to meet expectations.

Scaling a system invariably means adding new system components—hardware and software. As the number of components grows, we have more moving parts to monitor and manage. This is never effort-free. It adds complexity to the operations of the system and costs in terms of monitoring code that requires developing and observability platform evolution.

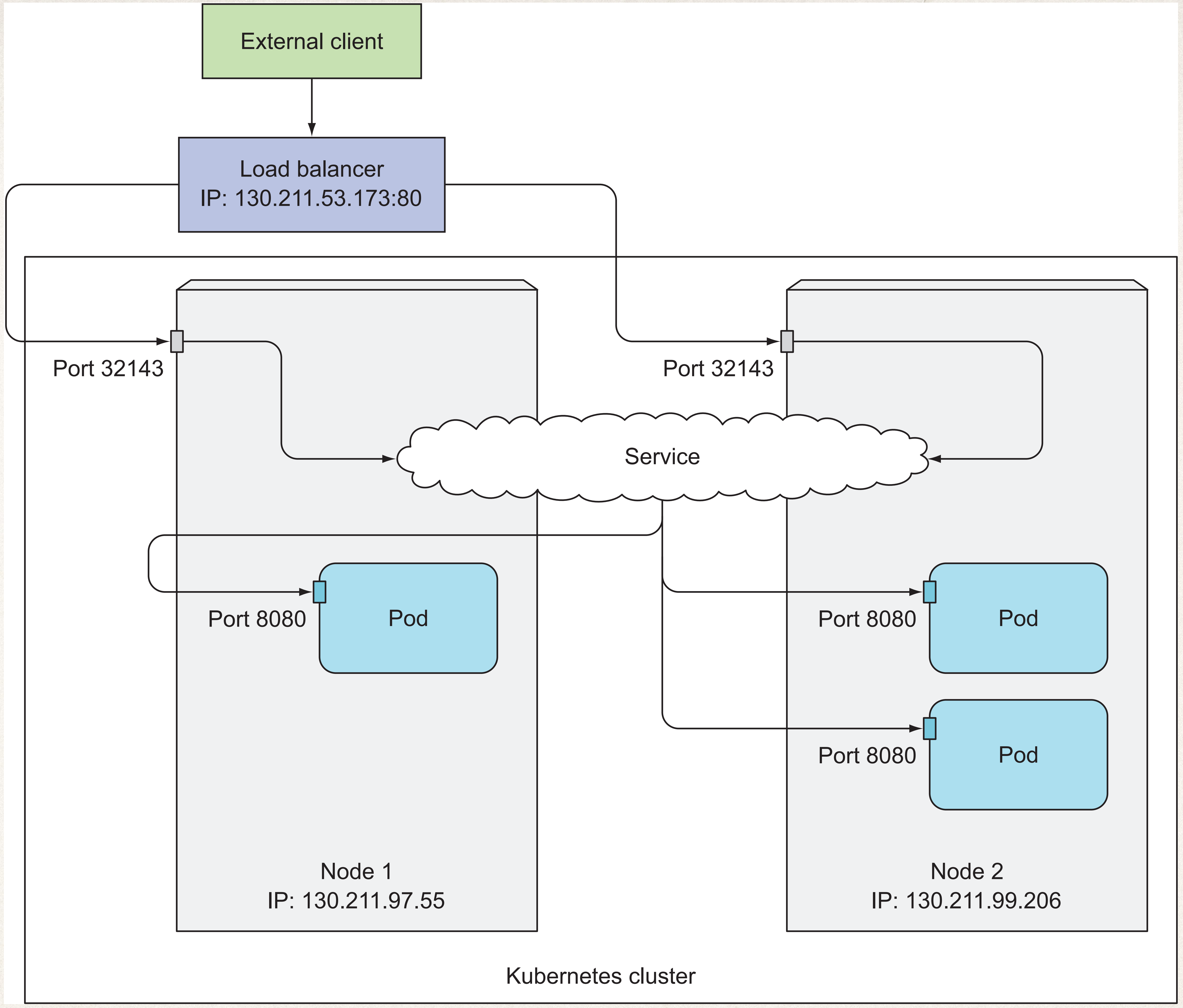
The only way to control the costs and complexity of manageability as we scale is through automation. This is where the world of DevOps enters the scene. DevOps is a set of practices and tooling that combine software development and system operations. DevOps reduces the development lifecycle for new features and automates ongoing test, deployment, management, upgrade, and monitoring of the system. It’s an integral part of any successful scalable system.

Kubernetes Load Balancers

Kubernetes is an open-source container orchestration platform that is widely used in the industry for managing containerized applications. Kubernetes doesn't deal with individual containers directly. Instead, it uses the concept of multiple co-located containers. This group of containers is called a Pod. These pods need to be accessed by the users. To achieve this, Kubernetes provides a Load Balancer service that distributes traffic across multiple pods and ensures high availability and scalability.

Load Balancers in Kubernetes are used to distribute traffic across multiple pods running the same application. They work by distributing incoming traffic across multiple backend pods using a round-robin or random algorithm.

Next figure below show how HTTP requests are delivered to the pod. External clients connect to port 80 of the load balancer and get routed to the implicitly assigned node port on one of the nodes. From there, the connection is forwarded to one of the pod instances.



An external client connecting to a LoadBalancer service

Load Balancers provide several benefits, including:

High Availability: Load Balancers ensure that traffic is always routed to available pods, even if some of the pods are down or unreachable. This ensures high availability and uptime for the application.

Scalability: Load Balancers can distribute traffic across multiple pods, which allows for horizontal scaling of the application. As more traffic comes in, additional pods can be added to the backend, and the Load Balancer will distribute the traffic accordingly.

Security: Load Balancers can provide SSL termination(Secure Sockets Layer, a security protocol that creates an encrypted link between a web server and a web browser), which ensures that traffic is encrypted between the client and the Load Balancer. This enhances security and protects against potential attacks.

Kubernetes Load Balancers provide several features that benefit networking, including:

Service Discovery: Load Balancers provide a single endpoint that can be used to access multiple pods running the same application. This simplifies service discovery and makes it easier to access and manage multiple pods.

Load Balancing Algorithms: Load Balancers can use different algorithms to distribute traffic across multiple backend pods.

This allows for more fine-grained control over traffic distribution and can help optimize performance.

Health Checks: Load Balancers can monitor the health of backend pods and automatically route traffic away from unhealthy or failing pods. This ensures that traffic is always routed to healthy and available pods, which enhances reliability and uptime.

Load balancers ensure that containerized applications have improved reliability and uptime by distributing traffic across multiple instances of a service and preventing any one instance from becoming overloaded. This prevents any one instance from becoming overwhelmed. This results in a system that is both highly available and resilient, meaning that it can withstand spikes in traffic and manage high volumes of traffic without experiencing any downtime.

Kubernetes Load Balancers are able to provide more advanced features thanks to tools like HAProxy, which enable them to perform tasks like SSL termination, content-based routing, and session persistence. This further improves security by encrypting traffic and routing it to the appropriate backend service based on the criteria that have been specified.

Exploring HAProxy

HAProxy is a popular choice for use as a load balancing software in production environments because of its dependability, scalability, and high-performance capabilities. It is an open-source programme. It is possible to deploy it either on-premises or in the cloud, and it can be used for many different kinds of applications and protocols, including HTTP, TCP, and UDP.

HAProxy is built with a multi-process architecture, which enables it to manage a large number of concurrent connections and requests. This is a key feature of the product. It uses a single master process that manages multiple worker processes, each of which can handle multiple connections simultaneously. This master process is managed by another master process. Because of its architecture, HAProxy is capable of horizontal scaling, which enables it to manage large volumes of traffic while also preserving its high availability.

Advantages of HAProxy Kubernetes Ingress Controller

- Scalability: HAProxy has the ability to scale horizontally to manage large volumes of traffic, making it possible to guarantee that containerized services are always accessible and quick to respond.
- Load Balancing: It is achieved through the use of HAProxy's proprietary load balancing algorithms, which ensure that incoming traffic is efficiently distributed across all backend services.
- Security: HAProxy is capable of terminating SSL connections, encrypting and decrypting traffic, and protecting against man-in-the-middle attacks and eavesdropping.
- Reliability: The health checks and automatic failover capabilities offered by HAProxy guarantee that containerized services will continue to be accessible and responsive at all times, even in the event that the underlying backend server fails.
- Flexibility: HAProxy's content-based routing enables developers to define routing rules based on specific criteria, such as URL path, HTTP headers, or source IP address, which allows for more granular control over how traffic is routed. HAProxy's other benefit is that this gives users more options.