Client-side Load Balancer for Twitter

Let's understand how Twitter performs client-side load balancing.

We'll cover the following

Introduction

In the previous lesson, we conceived the design of Twitter using a dedicated load balancer. Although this method works, and we’ve employed it in other designs, it may not be the optimal choice for Twitter. This is because Twitter offers a variety of services on a large scale, using numerous instances and dedicated load-balancers are not a suitable choice for such systems. To understand the concept better, let’s understand the history of Twitter’s design.

Twitter’s design history

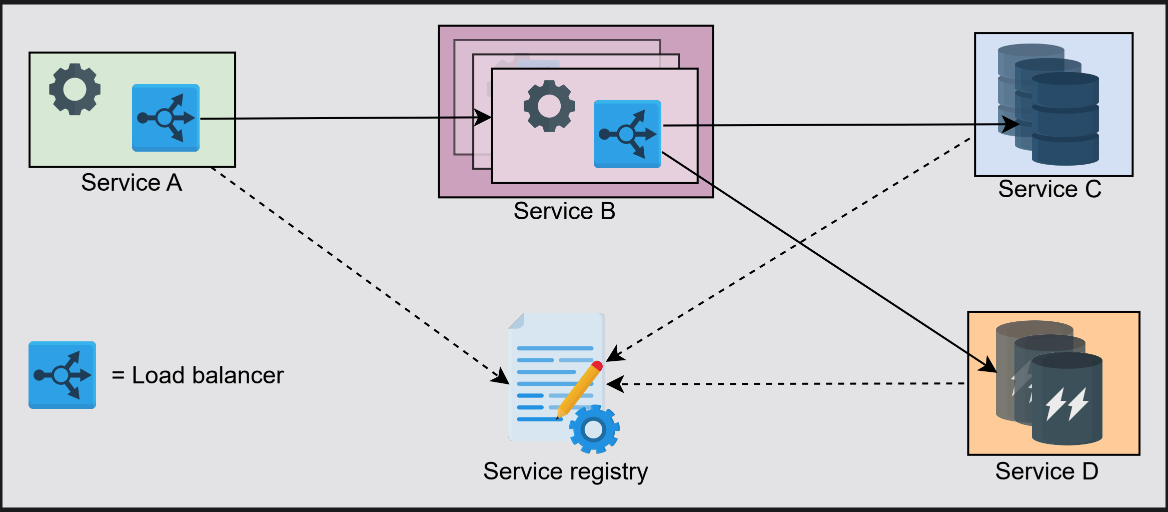
The initial design of Twitter included a monolithic (Ruby on Rails) application with a MySQL database. As Twitter scaled, the number of services increased and the MySQL database was sharded. A monolithic application design like this is a disaster because of the following reasons:

- A large number of developers work on the same codebase, which makes it difficult to update individual services.
- One service's upgrade process may lead to the breaking of another.
- Hardware costs grow because a single machine performs numerous services.
- Recovery from failures is both time-consuming and complex.

With the way Twitter has evolved, the only way out was many microservices where each service can be served through hundreds or thousands of instances.

Client-side load balancing

In a client-side load balancing technique, there's no dedicated intermediate load-balancing infrastructure between any two services with a large number of instances. A node requesting a set of nodes for a service has a built-in load balancer. We refer to the requesting node or service as a client. The client can use a variety of techniques to select a suitable instance to request the service. The illustration below depicts the concept of client-side load balancing. Every new arriving service or instance will register itself with a service registry so that other services are aware of its existence



How client-side load balancing works

In the diagram above, Service A has to choose a relatively less-burdened instance of Service B. Therefore, it will make use of the load balancer component inside it to choose the most suitable instance of Service B. Using t
same client-side load balancing method, Service B will talk to other services. It's

clear that Service A is the client when it is calling Service B, whereas Service B is the client when it is talking to Service C and D. As a result, there will be no central entity doing load balancing. Instead, every node will do load balancing of its own.

Advantages: Using client-side load-balancing has the following benefits.

- Less hardware infrastructure/layers are required to do load balancing.
- Network latency will be reduced because of no intermediate hop.
- Client-side load balancers eliminate bandwidth bottlenecks. On the other hand, in a dedicated load balancing layer, all requests go through a single machine that can choke if traffic multiplies.
- Fewer points of failure in the overall system.
- There is no end users queue waiting for the resource (server) for the particular services because many load balancers are routing the traffic. Eventually, it increases the quality of experience (QoE).

Many real-world applications like Twitter, Yelp, Netflix, and others use client-side load balancing. We'll discuss the client-side load balancing techniques used by Twitter in the next section.

Client-side load balancing in Twitter

Twitter uses a client-side load balancer referred to as **deterministic aperture** that is part of a larger RPC framework called Finagle. **Finagle** is a protocol-agnostic, open-source, and asynchronous RPC library.

> **Note:** Consider the following question. What does the client-side load balancer of Twitter balance on? That is, what parameters are used to fairly balance the load across different servers?

Twitter primarily uses two distributions to measure the effectiveness of a load balancer:

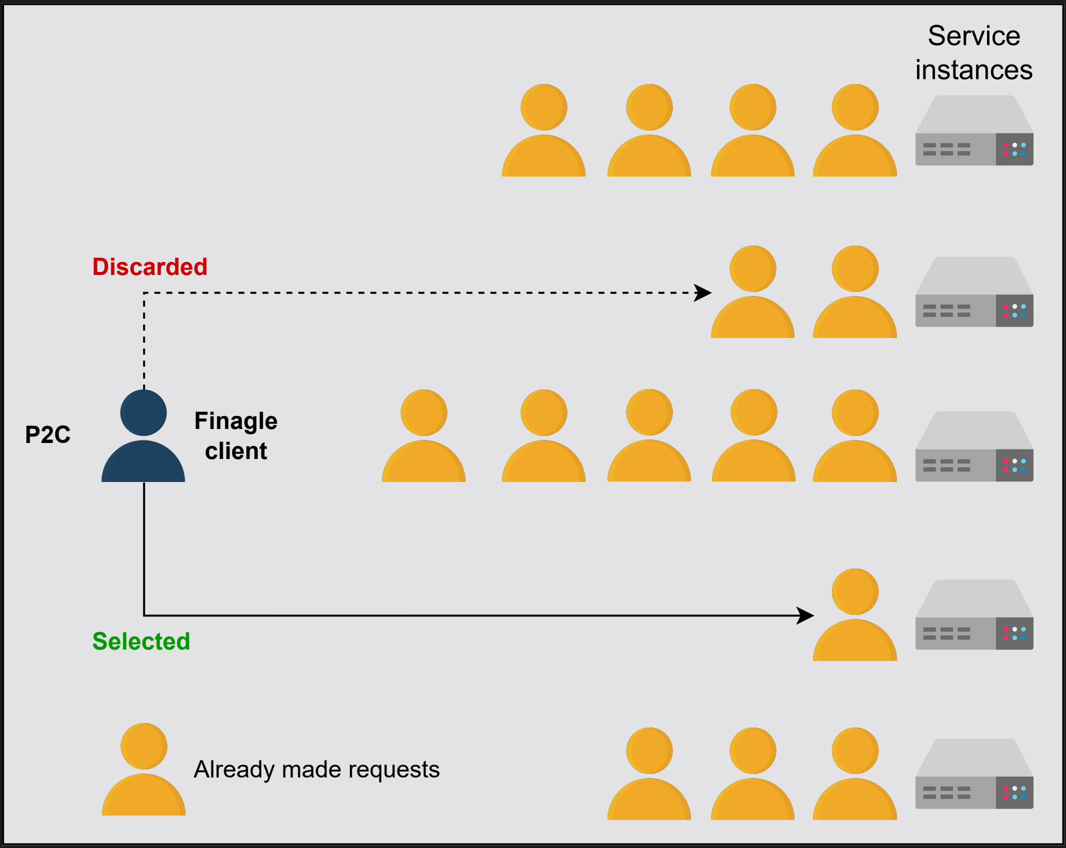
1. The distribution of requests (OSI layer 7)

2. The distribution of sessions (OSI layer 5)

Although requests are the key metric, sessions are an important attribute for achieving a fair distribution. Therefore, we'll develop techniques for fair distribution of requests as well as sessions. Let's start with the simple technique of Power of Two Random Choices (P2C) for request distribution first.

Request distribution using P2C

The **P2C technique** for request distribution gives uniform request distribution as long as sessions are uniformly distributed. In P2C, the system randomly picks two unique instances (servers) against each request, and selects the one with the least amount of load. Let's look at the illustration below to understand P2C.



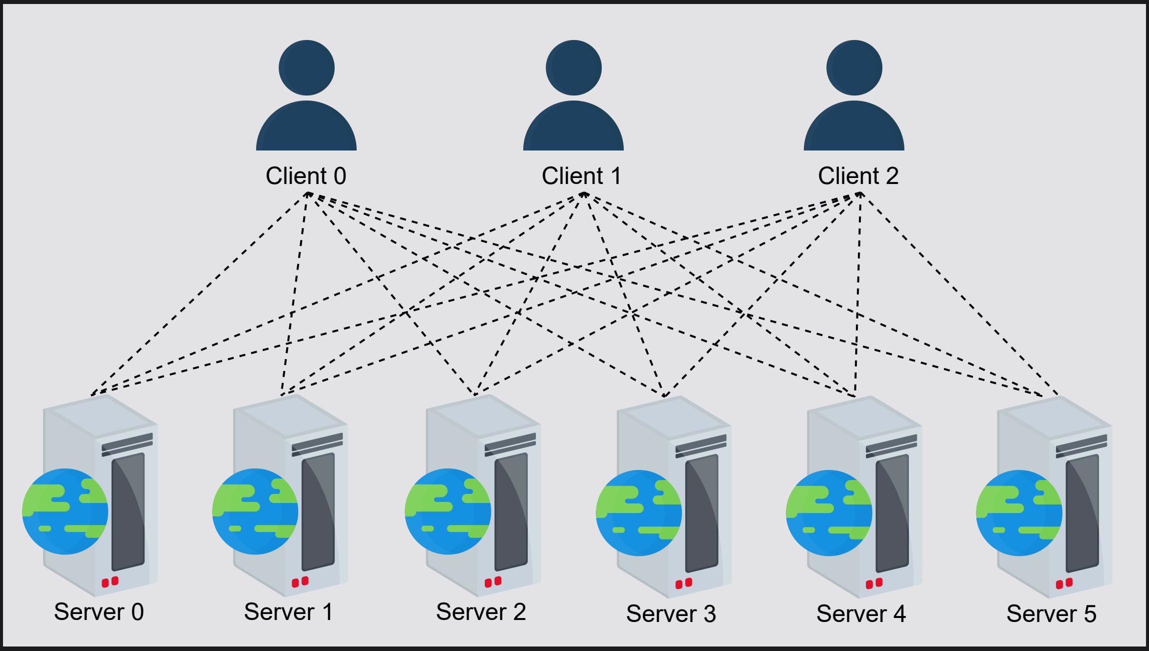
Power of two random choices

P2C is based on the simple idea that comparison between two randomly selected nodes provides load distribution that is exponentially better than random selection

Now that we've established how to distribute requests fairly, let's explore different techniques of session distribution.

Session distribution

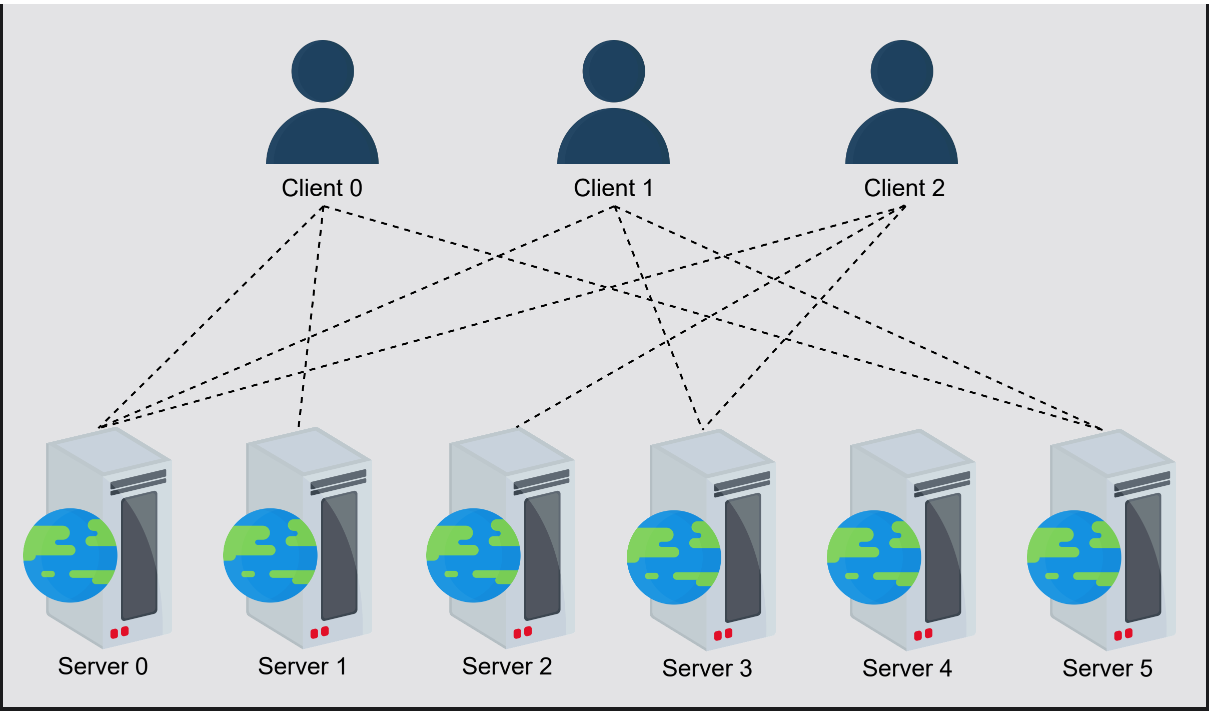
Solution 1: We'll start our discussion with the **Mesh topology**. Using this approach, each client (load balancer) starts a session with all of the given instances of a service. The illustration below represents the concept of mesh topology.



Mesh topology

Obviously, this method is fair because the sessions are evenly distributed. However, the even distribution of sessions comes at a very high cost, especially when scaled to thousands of servers. Apart from that, some unhealthy or stale sessions may lead to failures.

Solution 2: To solve the problem of scalability with solution 1, Twitter devised another solution called **random aperture**. This technique randomly chooses a subset of servers for session establishment.



Random aperture

Of course, random selection will reduce the number of established sessions as we can see in the diagram above. However, the question is, how many servers will be chosen randomly? This is not an easy question to answer and the answer varies, depending on the request concurrency of a client.

To answer the question above, Twitter installed a **feedback controller** inside the random aperture that decides the subset size based on the client's load. As a result, we're able to reduce the session load and ensure scalability. However, this solution isn't fair. We can see the imbalance in the illustration above such that **Server 4** has no session whereas **Server 0** has three sessions.

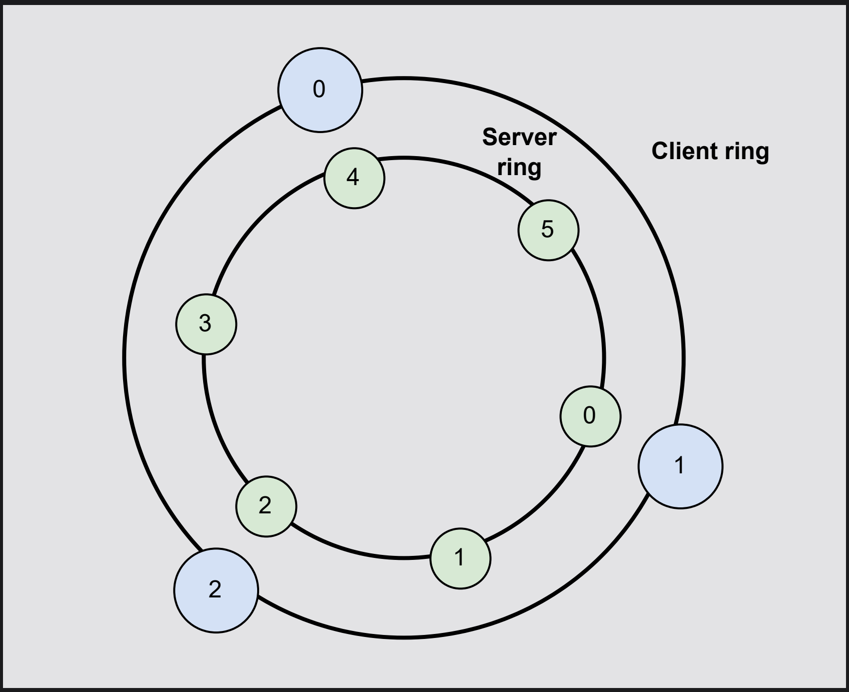
This imbalance creates problems such as idle servers, high load on a few servers, and so on. Therefore, we need a solution that is both fair and scalable.

Solution 3: We learned from solution 2 that we can scale by subsetting the number of session establishments. However, we failed to do the session distribution fairly. To resolve the problem, Twitter came up with a **deterministic aperture** solution. The key benefits of this approach are:

- Little coordination is required between clients and servers to converge on a solution.

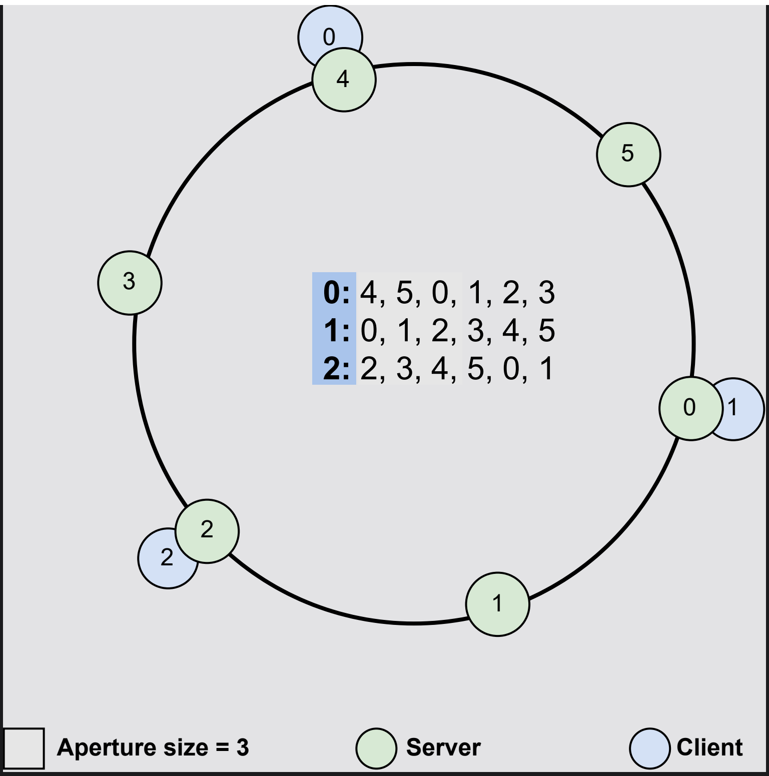
- Minimal disruption occurs if there are any changes in the number of clients or server instances for whatever reason.

For this solution, we represent clients and servers in a ring with equally distanced intervals, where each object on the ring represents the node or server labeled with a number. The illustration below shows what Twitter refers to as the discrete ring coordinates (that is, a specific point on the ring).



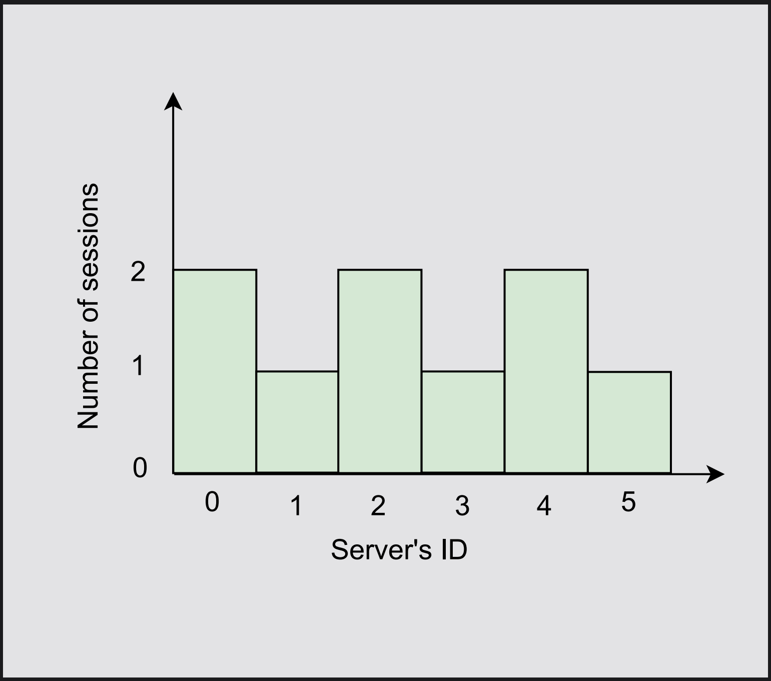
Ring coordinates

Now, we compose client and server rings to derive a relationship between them. We’ve also defined an aperture size of 3. Each client will create a session with three servers in the ring. Clients select a server by circulating clockwise in the ring. This approach is similar to consistent hashing, except that here we guarantee equal distribution of servers across the ring. Let’s see the illustration below where clients have sessions with servers chosen from the given list of servers, respectively.



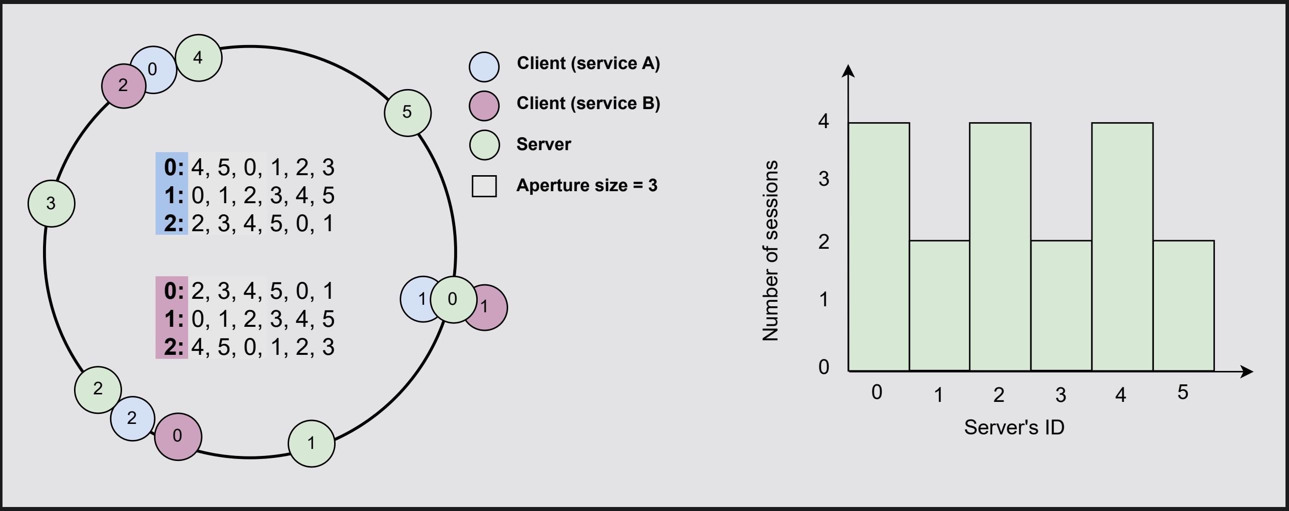
Merge client and server ring coordinates

In the diagram above, client 0 establishes sessions with servers 4, 5, and 0. Client 1 establishes sessions with servers 0, 1, and 2, whereas Client 2 establishes sessions with servers 2, 3, and 4. When the aperture rotates or moves in the given subset of arrays of the same size, each client's service has new servers available to create sessions. The number of sessions established per server is shown in the figure below.



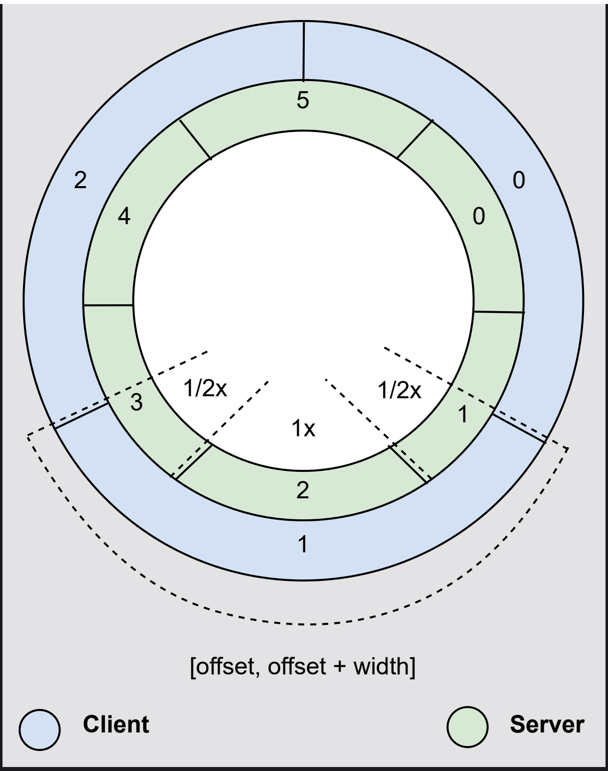
Number of sessions established per server

We can see that there's no idle server or server with a high load in the histogram graph above. Thus, we achieved a fair distribution for the clients belonging to the same service. It's clear that we've half resolved the problem by improving the session distribution among servers as compared to the random aperture (solution 02). However, the problem of unfair distribution can escalate as the number of clients grows. In that case, some servers will get crowded when more than one client (service) talks to the back-end servers. Let's see the illustration below to understand the problem.



Two different clients creating session with backend servers

The illustration above depicts that the problem compounded as we increased the number of clients (services). The solution is continuous ring coordinates where we derive relationships between clients and servers using overlapping ring slices, rather than specific points on the rings. The important thing here is that the overlapping of rings can be partial since servers on the ring may not have enough instances to divide among the clients equally. Let's look at the illustration below.



Overlapping among clients' services and servers

The illustration above shows that clients 0 and 1 (symmetrically) share the same server (server 1). It means that there is fractional overlapping concerning the ring coordinates. However, this overlapping can lead to asymmetric sharing as well. Symmetric or asymmetric, the load balancing can be done through P2C. For instance, in the diagram above, servers 1 and 3 will receive half the load that server 2 gets. This means that a server can receive more or less load, depending on its overlapped slice size with a client.

Now, we know how to map clients to the subset of the servers. We'll use P2C with continuous ring coordinates. The client will pick points (coordinations) in its range instead of picking the instance of the server. The selection process of clients to create sessions with backend servers are as follows:

1. Pick two coordinates (offset, offset + width) within its range and map these coordinates to the discrete instances (servers).
2. Select the one instance with the most negligible load from two chosen instances (P2C) by its degree of intersection.

Note: Continuous aperture is scalable and fair at the cost of an additional session over boundary node due to partial overlapping. It is also able to achieve little coordination and minimal disruption.

Let's look at the below table to get an overview of the discussed approaches.

Summary of Solutions for Session Distribution

Approach	Scalability	Fair distribution	Cos
Mesh topology	✗	✓	
Random aperture	✓	✗	