BUILDING SCALABLE GAME SERVERS

WITH MICROSERVICES

DEBDATTA BASU

Your game made it big! Hurray!

- Your simultaneous user count is growing at 250% per month!!
- You have excellent user engagement and your users talk about your game and refer it to others.
- You are all set to have a mind blowing balance sheet for the year!

Mo’ money, Mo’ problems

- Your simultaneous user count is growing at 250% per month?!!
- You want a system that can keep up with your success, but will your current architecture scale?
- The easiest way to lose users is to offer inconsistent or laggy gameplay, and you need a solution fast.

Mo’ money, Easy solutions

- Throw bigger hardware at the problem.
- So you have a 16 core machine with 64GB Ram? How about a 32 core machine with 128GB RAM?
- Awesome. It works. Everybody is happy. For the next month or so.
- You are growing at 250% per month.

Vertical Scaling has limits

- The largest instance on Google Compute Engine has 32 cores and 120 GB of memory.
- An optimized server for a lightweight game will support ~400 concurrent users per core. We will saturate the above instance at ~13K concurrent users.
- A successful game needs to support ~200k concurrent users or more.

Vertical Scaling is expensive

- All or nothing purchase.
- No fine grained control over infrastructure spend.
- High percentage of wastage on idle time in case of bursty workloads.

Things break in unexpected ways

- Temporary network and instance failures are common on public cloud providers.
- Instability may be caused by bugs and corner cases in your code that show up in production.
- Need graceful degradation in the face of partial failure.

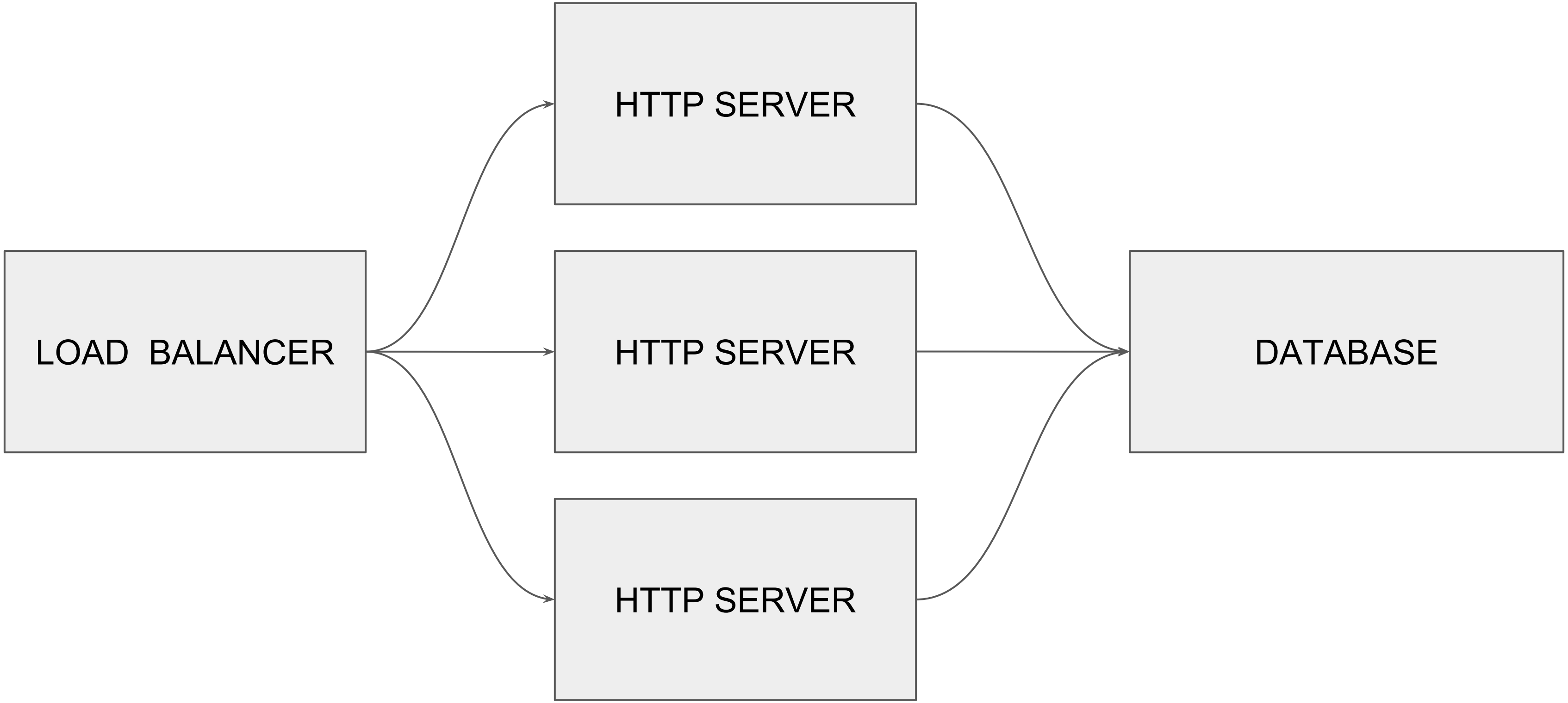
When one ox could not do the job, they did not try to get a bigger ox, but used two oxen. When we need greater computing power, the answer is not to get a bigger computer, but to build systems of computers and operate them in parallel.

Grace Hopper
Built Harvard Mark 1 in 1944

Back to the drawing board

- Use a swarm of computers.
- Dynamically spin up and down servers based on load.
- Pay only for what you use and minimize idling of infrastructure.
- Be horizontal, not vertical

Horizontal scaling is easy with HTTP



Horizontal scaling is harder for stateful services

- Each running game has to maintain it state in memory and process it in real time.
- The servers are not passive responders of requests, but are actors that can react in interesting ways to user input.
- The simple load balancer -> stateless server stuff will not work here.

Enter Microservices

- The key to scalability is decomposition and work distribution.
- The key to fault tolerance is redundancy.
- Microservices are an approach breaking down a system into independent units of functionality that collaborate with each other.
- They offer an elegant solution to both decomposition and redundancy.

The Icing on the cake

- Cleaner and more understandable codebase.

- Language and platform agnostic. Build different parts of the system on a platform most suited to that particular task.

- For example, build performance critical services in C++ and use Node for everything else.

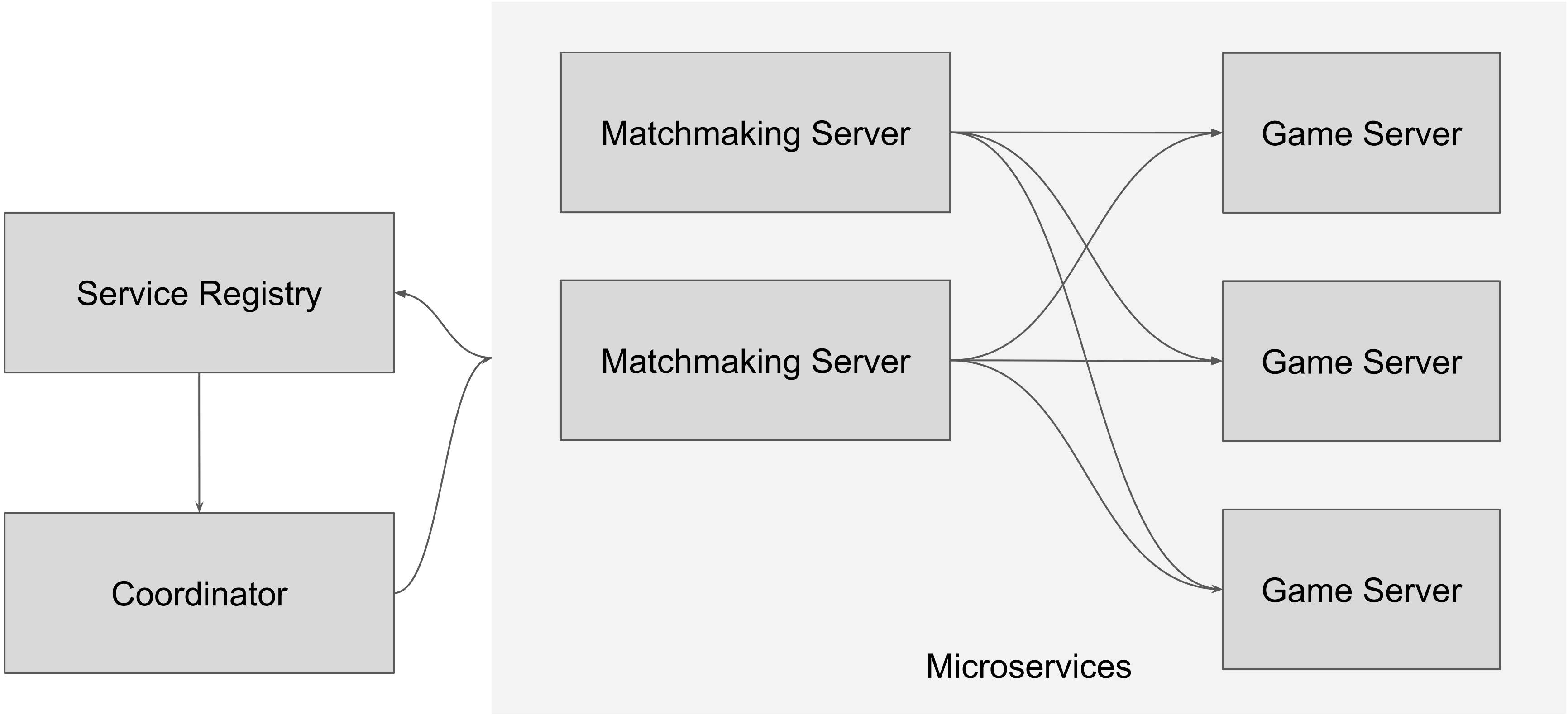
Principles of efficient decomposition

- Encapsulate logical system components into independent services.
- Minimize dependencies between systems.
- Loose coupling between services through asynchronous notification and messaging.

A typical 2 player game has

- A matchmaking service that matches players of similar skill in real time. Manages a queue of waiting players and assigns them to games.
- A game service that manages the actual running games. Each game runs on a single instance of a game service, and a game service can host multiple games.
- A leaderboard service that manages global and local leaderboards and keeps them up to date in real time.
- A discovery service that manages and supervises all running services.

Architecture Diagram



Joining a game - The basic protocol

- The client connects to the discovery service that sends a list of running matchmaking services.
- The client connects to one of the matchmaking services and requests a match.
- The matchmaking service finds a matching player and stores this information with a corresponding game id. It also finds a suitable game service instance to handle the game.
- It then sends a token containing the game id and the location of the game service to the client. The client connects and plays the game.

Discovery and Health checking

- All services register themselves by updating their records in a database.
- All services send periodic reports containing performance metrics that are useful for efficient routing.
- The discovery service monitors all registered services by sending periodic health checks and marking unavailable nodes in the database.

Scaling

- Spin up and down game service instances with varying load. Games are completely independent and embarrassingly parallel.
- The matchmaking service may become a bottleneck with higher loads.
- With skill based matchmaking, partition the skill level into ranges and send each range into a different instance.
- Partition using various application specific metrics.

In Conclusion

- Break the system into independent services.
- Single Responsibility Pattern - Do one thing and one thing well.
- Intelligently route work among a swarm of worker nodes.
- Automate health checking and monitoring.

DIVIDE AND CONQUER