Google

bittide kv-store

Calin Cascaval
May 11, 2022

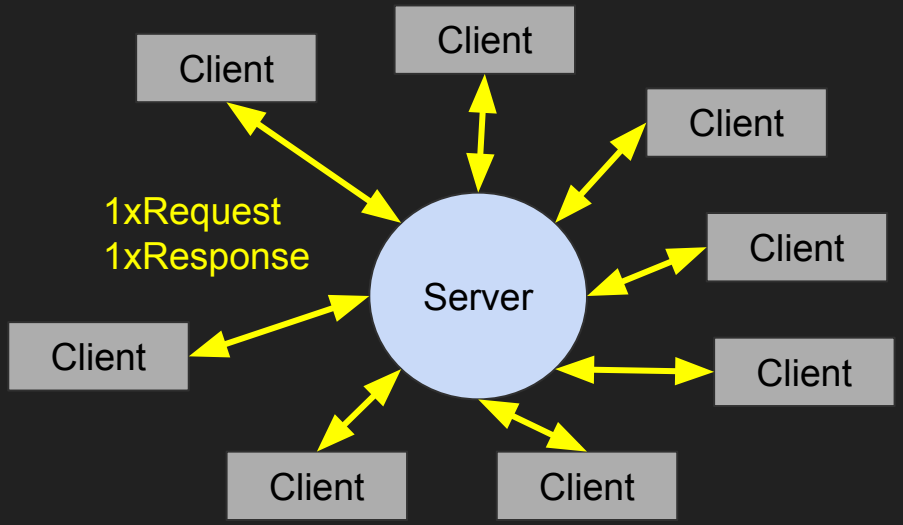
Overview

A bittide/aegir application that explores different configurations of microservices pipelines.

The main goal is to understand the different trade-offs required when statically scheduling various pipelines, such as:

- communication overhead
- “rigid” topologies
- overhead of sharding and replication

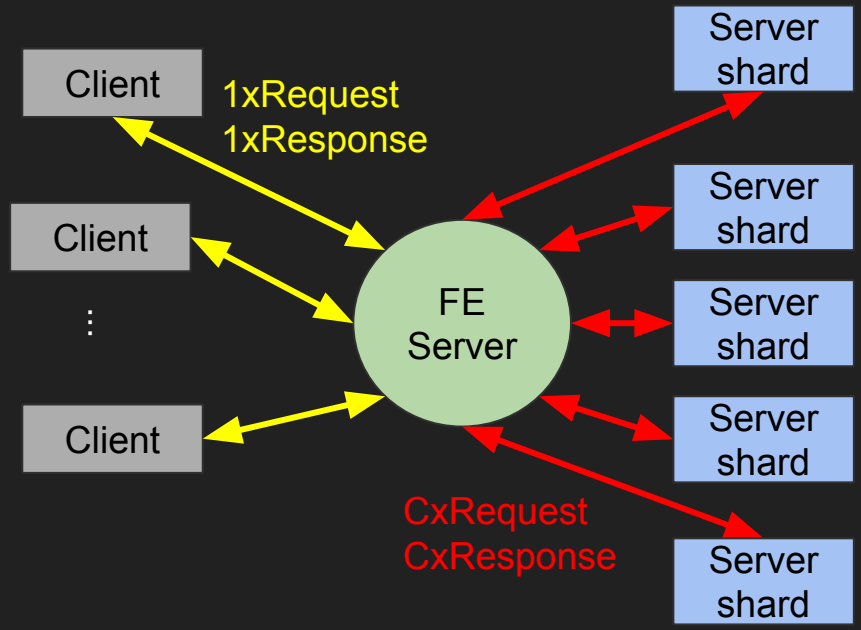
Topologies: single server



Each client is directly connected to the server:

- almost minimal, fixed comm size: Request from client, Response from server.
- latency of response: 2 cycles
- limited scaling: a single server must handle all requests.
- simple topology, simple code

Topologies: sharded server



Each client is connected to the front-end server. Clients are unchanged.

Front-end distributes requests to the shards and collects the responses.

Server shards may potentially serve requests from all clients.

- Comm: FE -> shard communication size directly proportional with # of clients.
- latency of response still fixed – 4 cycles.
- scaling:
  - shards can be added transparently – as good as the hash function.
  - a more complex FE could handle partitioning clients to limit the comm size.
- all partitioning hidden in the FE.

Replication

Servers are sharded, and shards have replicas

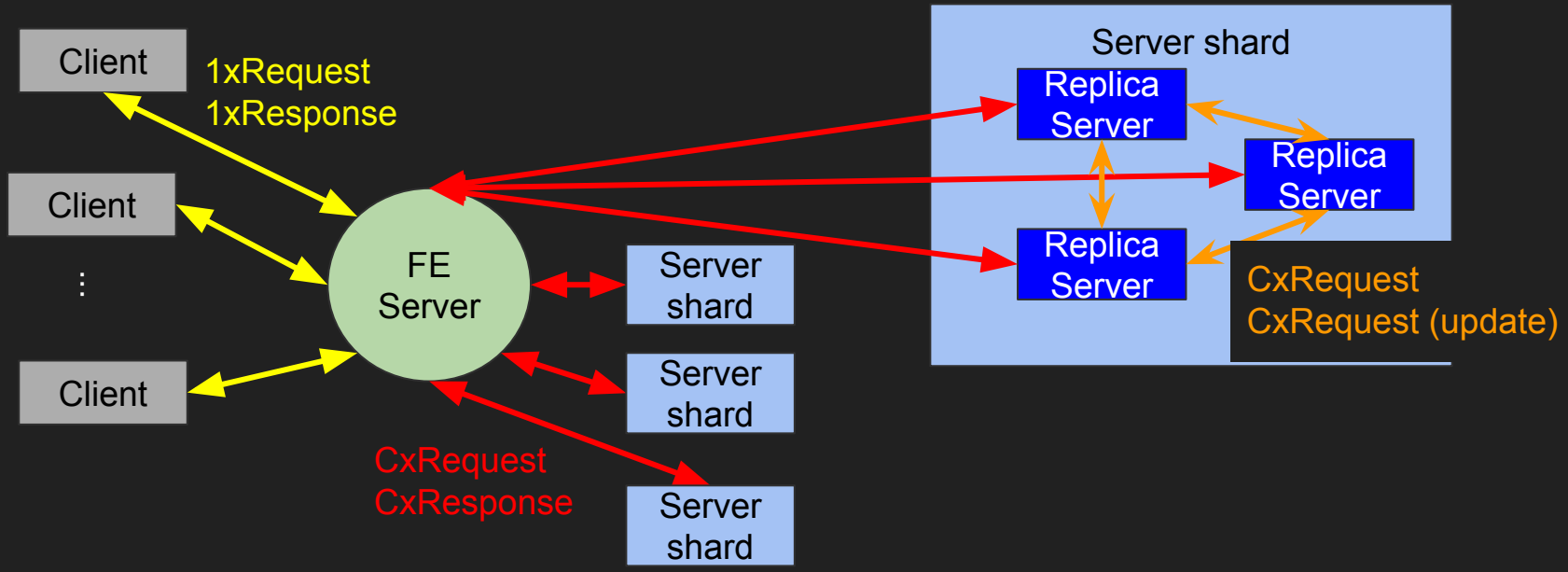
A front-end node manages the shards and their replicas

Options:

1. the front-end sends requests to all replicas of a shard and uses the response from the first one that replies
2. the front-end sends requests to one of the replicas, and replicas manage consistency between themselves

We build option 2: there is no point to spray replicas, since communication is deterministic – there is no “first one”!

Topologies: replicated server (and sharded)



Topology: replicated server

Each client is connected to the front-end server. Clients are unchanged.

Front-end distributes requests to the shards (one of the replicas) and collects the responses:

- shards are selected just as before, by the hash function
- a replica is selected randomly from the set of replicas
- FE is connected to all replicas

Server shards (each replica) may potentially serve requests from all clients.

Comm: FE -> shard communication size directly proportional with # of clients.

Latency of response: same as the sharded topology – 4 cycles.

Scaling:

- shards can be added transparently – as good as the hash function.
- a more complex FE could handle partitioning clients to limit the comm size.
- replicas deal with managing their state (see discussion on replication protocol).

All partitioning hidden in the FE.

All replication hidden in the shard; the FE does need connections to all replicas.

- shards can be scaled independently
- a more hierarchical design with a designated shard server is possible, introduces latency

Replication protocol

Each replica server does the following (each cycle):

- process all replica updates
- serves the request(s)
- sends the request(s) to all replicas

No separate consistency protocol:

- replica updates are processed exactly in 1 cycle
- if FE sends a request for a key, it will take 1 cycle to hit a replica server; processing the updates before serving the request guarantees that the most up-to-date value is served.

Scalability summary

Static scheduling => all resources are dedicated and will “execute” every cycle.

In principle, the distribution of inputs will dictate the number of shards/replicas, similar to how systems are provisioned today.

- a good hash function to distribute the requests evenly is key
- worst-case handled through communication overhead: servers must be able to potentially handle all clients. Smarter partitioning is certainly possible

Latencies are known and fixed

- each level in the hierarchy adds 2 cycles

Persistent state

Servers have state (the kv-store) that is serialized and deserialized every cycle

- does not scale for database-like stores – unless we have infinite cycle length
- current bittide support for persistent state: partition and reserve a slice of memory – doesn’t scale either, in particular if we’re considering virtualization.
  - identify such actions and reserve nodes
  - persistent storage that partitions better than memory

Failures

Single server:

- server failure is fatal!

Sharded topology:

- loss of data on the failed shard; FE changes hashing function
- FE server failure is fatal; no state is lost (FE is stateless)

Replicated sharded server:

- tolerates R-1 replica failures in a shard without loss of data (duh!); FE changes the replica selection function
- FE server failure is fatal; no state is lost (FE is stateless)

Client failures:

- who cares?

Brittleness

Communication size between FE and shards, and between replicas in a shard depends on the number of clients:

- in principle, we can define the “server bandwidth” and partition clients to servers. Fixed partitioning!

Computation size depends on the maximum number of requests a server can get.

Other issues

From Rob's doc:

- Data dependent routing: depends on the FE hash function (much as it is today)
- Caching - not addressed
- Request sizing: yep, fixed (maximum) sized data exchanged. Heavy use of Optional types to send lots of None.
- Failures: looks reasonable to me!
- Brittleness: server comm sizing depends on # of clients

Future work

Scheduling optimizations:

- explore existing input distributions (traces from memcached?) and look for opportunities to improve the time spend in shards
- explore using replicas in shards to improve concurrency

Brittleness – decouple the number of clients from the server side

- fixed rates of service per shard, i.e., partition the clients and smarten the FE to handle the client partitions.
- queues in the FE and trade-off latency

Caching?