Engineering Distributed Systems

experiences, lessons, and suggestions
About Me

• Anant Bhardwaj

  • PhD student (on leave)
    • MIT Computer Science & Artificial Intelligence Lab (CSAIL)
    • w/ Sam Madden & David Karger

  • Founder & CEO (Instabase)
Today’s Talk

• Share experiences in designing real-world distributed systems

• Not about best practices or formal correctness

• More on taking pragmatic decisions
  • balancing simplicity, performance, scalability, reliability, and extensibility.
Instabase

- Based on MIT DataHub project

- A new architecture of data management where one can layer applications on a common shared platform that provides a way to manage diverse datasets with built-in sharing/collaboration and access control.

- Demo
Distributed Systems

• Problems [data or web requests] that are too large for a single machine
  • Can be solved if many machines co-ordinate (typically over network)
  • Making machines co-ordinate bring a new set of challenges

• How to make many machines co-ordinate?
  • Break the large problem into smaller sub-problems

• Services
  • A self-contained unit of functionality
  • Exposed via an Interface
A sample service interface

```c
256 struct IsDirReq {
    1: optional Connection con;
    2: optional string path;
}
260
261 struct IsDirResp {
    1: optional Status status;
    2: optional bool is_dir;
}
264
// service APIs
266 service FileService {
    ConnectResp connect(1: ConnectReq req);
    ReadFileResp read_file(1: ReadFileReq req);
    WriteFileResp write_file(1: WriteFileReq req);
    MkdirResp mkdir(1: MkdirReq req);
    RmResp rm(1: RmReq req);
    ListDirResp list_dir(1: ListDirReq req);
    IsDirResp is_dir(1: IsDirReq req);
    IsFileResp is_file(1: IsFileReq req);
```
Service Protocol

- Interface Definition Language
- Binary communication protocol
- We use Apache Thrift ([https://thrift.apache.org/](https://thrift.apache.org/))
  - Clean IDL (Interface Definition Language) syntax
  - Automatically generates language wrappers (C++, Java, Python, Go, ...)
  - Compact + fast serialization/deserialization
Many Internal Services

• Everything is broken into services
  • AccountService
  • FileService
  • TableService
  • ETLService
  • RouteService
  • ...

Benefits of Services

• Better control over scaling
  • Have more instances of FileService and only a couple of instances of AccountService

• Better control over performance optimization
  • Selectively optimize services for performance. RouteService needs to be fast

• Easy fault-tolerance
  • All services are stateless so in case of a failure – the request can be routed to another healthy instance
Challenges in Scaling

• A single server serving all FileService requests

```python
# open a file
f = ib.open('/instabase/demo/fs/Instabase%20Drive/files/datasets/hadoop.log')
data = f.read()  # reads the entire content
print(data)
```

• Let’s add more servers to scale (and for high availability)
  • What can go wrong?
  • What if ib.open() went to server 1 and f.read() to server 2?
Challenges in Scaling

• Many instances of FileService serving file requests
  • What can go wrong?

```python
# open a file
f = ib.open('/instabase/demo/fs/Instabase%20Drive/files/datasets/hadoop.log')
while f.tell() != -1:
    print("==== chunk begin =====")
    data = f.read(1024)  # reads only 1KB
    print(data)
    print("==== chunk end =====")
```
Challenges in Scaling

• RouteService

Route Service

/user/david = 10.20.30.40
/user/anant=10.20.50.60

/preparing

10.20.60.70

/in use

10.20.30.40
10.20.50.60

Machine Cluster

/user/zoltan

/user/david

/user/zoltan
Challenges in Scaling

Route Service 1
/user/david = 10.20.30.40
/user/anant=10.20.50.60

Route Service 2

Route Service 3

/user/zoltan

/user/david

Machine Cluster

Preparing
10.20.60.70

In Use
10.20.30.40
10.20.50.60
Consistency & Availability

• The problem wouldn’t occur if we had just one instance of Route Service

• But we care about availability and fault-tolerance
  • Brings consistency issues

• How to keep the state of two replicas consistent?
Consistency

• Consistency can be pushed to lower level
  • For example: to a key-value store

• Strong consistency has a huge cost
  • In many cases, eventual consistency is just fine.
Fault Tolerance, Failure, and Recovery

• Import a 1TB file into a Table

• A possible solution:
  Repeat until done (ETLService)
    Read 10MB chunk (FileService)
    Write to table (TableService)
  • How to recover from failure w/ minimum re-work?
  • Which service is responsible for recovery?
Distributed Systems

• Most of the academic work focus on
  • Consistency
  • Performance
  • Availability (fault-tolerance and reliability)

• Equally important
  • Simplicity
  • Extensibility
  • Maintainability
  • Debugability
Simplicity, Maintainability, and Extensibility

• The system is a composition of many independent, stateless services
  • not a monolithic statically linked executable

• Simpler software engineering
  • fewer dependencies, clearly specified (as service interface)
  • easy to test different versions

• Development of services decoupled
  • teams can work independently
  • allows reuse (even better with language-agnostic IDL)
Scaling

• Scaling brings new set of challenges
  • Scale services independently
    • Advantage: Performance, Availability
    • Problem: Consistency

• Push the consistency to the data layer
  • Don’t make instances of services co-ordinate
  • Push the co-ordination and consistency to the data layer (key value store, or databases)
Performance

• Have a rough estimate of latency
  • How many service (RPC) calls
    • add a few micro-seconds as network cost (assuming within datacenter call)
  • How many database calls
    • add tens to hundreds of milliseconds
  • How many cache calls (hits vs misses)
    • add a few micro-seconds to milliseconds (cache miss)

• Optimize only those parts that are performance critical
Debugging & Monitoring

• Logs are useful but not sufficient
  • Connecting logs in distributed infrastructure is hard

• Debug at service level
  • use status codes
  • service should return well defined status codes

• Monitoring
  • have periodic health checks
  • define timeouts
  • have recovery mechanism
Takeaway

• Identify the building blocks
  • FileService, TableService, RouteService, Cache, etc.
  • Have a rough estimate of latency of each of these

• Think carefully about interfaces in your system
  • Build a general system but don't imagine unlikely potential needs that aren't really there
  • Best approach: use your own infrastructure

• Infrastructure building is not just about scaling, performance, consistency, and availability
  • you need to balance simplicity, extensibility, maintainability, and debugability
Questions

• Please send any comments to anantb@instabase.com