Distributed Architectures and Abstractions

CS 475, Fall 2019
Concurrent & Distributed Systems
RPC: High Level Approach

Caller Machine:
- User Code
- local call
- local return

Callee Machine:
- User Code
- local call
- work
- local return
Shared Fate

- Two methods/threads/processes running on the same computer generally have **shared fate**
- They will either both crash, or neither will crash
Split Brain in RPC

Split brain: Client thinks addPerson didn’t succeed, server did complete it though!
Split Brain in RPC

This gets even worse when you consider more complicated semantics

Who has the lock?
RPC Semantics

• No matter what we do, if we want RPC, we have networks, networks might have timeouts/failures
• How do we handle the potential for split brain?
  • If we don't hear a response, just freeze?
• What can the abstraction guarantee?
  • Leak some of this complexity through
Java RMI

- Synchronous (client method doesn’t return until server completes)
- At most once delivery
- Hence, in the event of a communication failure, an exception is thrown on your client
- Implications:
  - Client code needs to be aware that failures might happen (and exception might be thrown)
  - Client code needs to have some plan to handle when a message fails to get through (application specific)
Java RMI

• Threading model:
  • What happens when there are multiple simultaneous RMI requests to the same server?
  • RMI creates a *thread pool*, a set of threads ready to handle each request
  • Subsequent calls from the same client might or might not use the same thread
  • Subsequent calls from other clients might use the same thread as others
• Implications:
  • Can process multiple requests simultaneously
  • Need to be cognizant of thread safety
Java RMI

```java
public interface AddressBook extends Remote {
    public LinkedList<Person> getAddressBook() throws RemoteException;

    public void addPerson(Person p) throws RemoteException;
}

AddressBook book = new AddressBookServer();
AddressBook stub = (AddressBook) UnicastRemoteObject.exportObject(book, 0);
Registry registry = LocateRegistry.createRegistry(port);
registry.rebind("AddressBook", stub);

Registry registry = LocateRegistry.getRegistry("localhost", 9000);
AddressBook addressBook = (AddressBook) registry.lookup("AddressBook");
```
Java RMI

• Registration of a server makes it possible for a client to locate the server and bind to it
• Server location is done in two steps:
  • Locate the server’s machine.
  • Locate the server on that machine.
Split Brain in RPC

This gets even worse when you consider more complicated semantics

Who has the lock?
Sidebar: Heartbeat Protocols

- Allow client/server to remain aware of each other’s status
- For HW3: does client still have locks (client checking server, server checking client)

```
Client

lock("foo")

OK, stamp = 1

Hi, I’m stamp 1, still have foo

Server

Hmm, I guess server is gone, maybe lock is not valid

CRASH!
```
Sidebar: Heartbeat Protocols

- Allow client/server to remain aware of each other’s status
- For HW3: does client still have locks (client checking server, server checking client)

```
lock("foo")
OK, stamp = 1
Hi, I’m stamp 1, still have foo
OK
CRASH!

Hmm, I guess foo is no longer locked
```
Sidebar: Heartbeat Protocols

• We call these time-limited locks **leases**
• What does a lease guarantee?
  • If no network failures
    • Locks that are relinquished when client crashes
  • If network failures/delays:
    • Nothing
RPC Summary

• Expose RPC properties to client, since you cannot hide them
  • Application writers have to decide how to deal with partial failures
  • Consider: E-commerce application vs. game
Today

- Distributed Systems Architectures: How do we build a big thing from lots of little things?
- Today: How to compose some big blocks
- Next few weeks: How to build each of those blocks
- Reminders:
  - HW3 Posted
  - Prof Bell out of town next week
Distributed Systems Abstractions

• Goal: find some way of making our distributed system look like a single system
• Never achievable in practice
• BUT if we can come up with some model of how the world might behave, we can come up with some generic solutions that work pretty well
• And hopefully we can understand how they can go wrong
Abstractions & Architectures

• We can design architectures that embody some systems model, providing some framework code to make it easier to get some task done
• Case study example: web architectures
• Assumptions:
  • “one” server, many clients
  • Synchronous communication
  • Client is unlikely to be partitioned from a subset of servers; likely some subset of servers are partitioned from other servers
  • Client is mostly stateless
The good old days of web apps

HTTP Request
GET /myApplicationEndpoint HTTP/1.1
Host: cs.gmu.edu
Accept: text/html

HTTP Response
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
<html><head>...

web server

Runs a program

Give me /myApplicationEndpoint

Here’s some text to send back

Does whatever it wants

Web Server Application

My Application Backend
Brief history of Backend Development

• In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted

• Then… PHP and ASP
  • Languages “designed” for writing backends
  • Encouraged spaghetti code
  • A lot of the web was built on this

• A whole lot of other languages were also springing up in the 90’s…
  • Ruby, Python, JSP
Backend Frameworks

• Then: **frameworks**
• SailsJS, Ruby on Rails, PHP Symfony, Python Django, ASP.NET, EJB…
• MVC - separate presentation, logic and persistence
Scaling web architectures up

• What happens when we have to use this approach to run, say… Facebook?
• Tons of dynamic content that needs to be updated, petabytes of static content (like pictures), users physically located all over, lots of stuff to keep track of, where do we start?
Real Architectures

• For each layer...
  • What is it?
  • Why?

Clients

Internet

External Cache

Web Servers

App Servers

Database servers

Internal Cache

Misc Services

Real Architectures

• For each layer…
  • What is it?
  • Why?
External cache

• What is it?
  • A proxy (e.g. squid, apache mod_proxy)
  • A content delivery network (CDN) e.g. Akamai, CloudFlare
External cache

- What is it for?
  - Caches outbound data
    - Images, CSS, XML, HTML, pictures, videos, anything static (some stuff dynamic maybe)
  - DoS defense
  - Decrease latency - might be close to the user
• What is it made of?
• Tons of RAM, fast network, physically located all over
• No need for much CPU
Front-end Tier

- Serves static content from disk, generates dynamic content by dispatching requests to app tier
- Speaks HTTP, HTTPS
Application Server Tier

- Serves dynamic pages
- Provides internal services
  - E.g. search, shopping cart, account management
- Talks to web tier over..
  - RPC, REST, CORBA, RMI, SOAP, XMLRPC… whatever
- More CPU-bound than any other tier
Database Tier

- Relational or non-relational DB
- PostgreSQL, MySQL, Mongo, Cassandra, etc
- Most storage
• Has tons of memory, right near the app servers to cache application-level (dynamic) objects.
Internal Services Tier

- Coordination services
- E.g. time keeping
- Monitoring & maintenance services

Diagram:
- Internet
- Clients
- External Cache
- Web Servers
- App Servers
- Database servers
- Internal Cache
- Misc Services
- Internal Services Tier
- Web Servers
- Database servers
Real Architectures

N-Tier Web Architectures

Separate out responsibilities with abstractions: each tier cares about a different aspect of getting the client their response.
How do we build big apps?

What happens when we want to add more functionality to our backend?
How do we build big apps?

Our Cool App

- Frontend
- Backend Server
- Database

Basic todo app with new feature to email todo reminders

What happens when we add more functionality?
How do we build big apps?

Our Cool App

- Frontend
- Backend Server
- Database

Basic todo app with new feature to email todo reminders PLUS something to find events on Facebook and create Todos for them

But we’re smart, and learned about modules, so our backend isn’t total spaghetti but rather…
How do we build big apps?

Our Cool App

Frontend

Backend Server

Mod 1
Mod 3
Mod 5

Mod 2
Mod 4
Mod 6

Database

Sweet: Our backend is not an unorganized mess, but instead just modules. Now how do we scale it? Run multiple backends?
Now how do we scale it?

We run multiple copies of the backend, each with each of the modules.
What's wrong with this picture?

- This is called the “monolithic” app
- If we need 100 servers…
- Each server will have to run EACH module
- What if we need more of some modules than others?
- How do we update individual modules?
- Do all modules need to use the same DB and language, runtime etc?
Microservices

Our Cool App

Frontend

“Dumb” Backend

AJAX

NodeJS, Firebase
Todos
REST service
Database

Google Service
Accounts
REST service
Database

Java, MySQL
Mailer
REST service
Database

Search Engine
REST service
Database

Analytics
REST service
Database

Java, Neo4J

Facebook Crawler
REST service
Database

C#, SQLServer

C#, SQLServer

Python, Firebase
What’s good about this picture?

- Spaghetti is contained
- Components can be developed totally independently
  - Different languages, runtimes, OS, hardware, DB
- Components can be replaced easily
  - Could even change technology entirely (or use legacy service)
- Can scale individual components at different rates
  - Components may require different levels of resources
Requirements for successful microservices

- 1 component = 1 service
- 1 business use case = 1 component
- Smart endpoints, dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design
Organization around business capabilities

Classic teams: 1 team per “tier”

Frontend
Orders, shipping, catalog

Backend
Orders, shipping, catalog

Database
Orders, shipping, catalog
Organization around business capabilities

Example: Amazon

Teams can focus on one business task
And be responsible directly to users

“Full Stack”

“2 pizza teams”
N-Tier Web Architectures

Separate out responsibilities with abstractions: each tier cares about a different aspect of getting the client their response.
Abstracting the tiers

• Take, for instance, this *internal cache*
• Can we build one really good internal cache, and use it for all of our problems?
• What is a reasonable model for the cache?
  • Partition: yes (get more RAM to use from other servers)
  • Replicate: NO (don’t care about crash-failures)
  • Consistency: Problem shouldn’t arise (aside from figuring out keys)
How much more can we abstract our system?

- At its most basic… what does a program in a distributed system look like?
  - It runs concurrently on multiple nodes
  - Those nodes are connected by some network (which surely isn’t perfectly reliable)
  - There is no shared memory or clock
- So…
  - Knowledge can be localized to a node
  - Nodes can fail/recover independently
  - Messages can be delayed or lost
  - Clocks are not necessarily synchronized -> hard to identify global order
Back to reality

• That’s a little TOO abstract - given that system, how can we define a good way to build one?
• In practice, we need to make assumptions about:
  • Node capabilities, and how they fail
  • Communication links, and how they fail
  • Properties of the overall system (e.g. assumptions about time and order)
Designing and Building Distributed Systems

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions.

Generally: Stronger assumptions -> worse performance
Weaker assumptions -> more complicated

Strength

System model
Asynchronous
Synchronous

Failure Model
Crash-fail
Partitions

Consistency Model
Eventual
Sequential

J. Bell
GMU CS 475 Fall 2019
Timing & Ordering Assumptions

• No matter what, there will be some latency between nodes processing the same thing
• What model do we assume though?
  • Synchronous
    • Processes execute in lock-step
    • We (the designers) have a known upper bound on message transmission delay
    • Each process (somehow) maintains an accurate clock
  • Asynchronous
    • Opposite - processes can run out of order, network arbitrarily delayed
Modeling network transmissions

- Assuming how long it can take a message to be delivered helps us figure out what a failure is.
- Assume (for instance), messages are always delivered (and never lost) within 1 sec of being sent.
- Now, if no response received after 2 sec, we know remote host failed.
- Typically NOT reasonable assumptions.
How do we know what the *actual* order of these events was?
Modeling clocks

• Our synchronous system (and any other really) needs some notion of time
• Are we doing things at the same time? Are they in order?
• One option… synchronize clocks (i.e. via NTP)
• NTP is a great way to automatically synchronize computer clocks, with accuracy of?
  • ~1ms between in local networks, ~10ms across internet
Modeling Clocks: Global Ordering

How do we know what the actual order of these events was?
Lamport Clocks

• Idea:
  • We just care about “happens-before”
  • NOT the infinitesimal granularity of time
  • Don’t track time in an absolute sense
  • Just track time relative to things that you care about
  • Within a process:
    • You can keep track locally
  • Between processes:
    • Send a message… and assume that when you receive it is after when you sent it
Modeling Clocks: Global Ordering

Node1 sends m1 after b, Node2 receives it before c, then sends m2 after c, which is received by node3 after d

Hence, the global order is: 

\[ a \ b \ c \ d \ e \]
Modeling Clocks: Global Ordering

Not all events are related

*a* and *f* are *concurrent* - we don’t know which happened first (maybe didn’t care?)
**Modeling Clocks: Global Ordering**

**Implementation**: Each node maintains a logical clock $L$
- $L$ is incremented by 1 before each event at each process
- When a process sends a message $m$, it sends its current time counter
- When a process receives a message, it updates its clock $L$ from that message as the max of it and its time
Logical Clocks

• Lamport clocks are a *logical clock*
• Encodes a causality relationship
• They are relative: if e1 happened before e2, then L(e1) < L(e2)
• Note: similarity between global clocks and linearizability, Lamport clocks and sequential consistency
Back to synchronous models...

- So, it’ll be possible to ensure that one event doesn’t occur before another
- It won’t be free
  - We need to keep these clocks in sync by sending messages back and forth!
- We now have a new way to fail...
  - If these clock messages start getting dropped
Asynchronous Systems

- The exact opposite
- Do not rely on *any* timing assumptions
- Processes execute at their own rates
- Messages can be delayed indefinitely
- No useful clocks
To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions.

Generally: Stronger assumptions -> worse performance
Weaker assumptions -> more complicated

System model | Failure Model | Consistency Model
---|---|---
Synchronous | Byzantine | Sequential
Asynchronous | Partitions | Crash-fail
                |          | Eventual
Nodes in our system

• Nodes…
  • Run programs
  • Store data in volatile memory (lost on crash) and stable storage (persists past some crashes)
  • Have some clock (may or may not be accurate)
• Key assumptions:
  • Nodes should typically behave *deterministically* given the same inputs/messages/system state
  • Nodes are well-behaved, and fail by crashing (which they might be able to self-recover from)
Communication Links in our System

• Does the network ensure ordering?
  • E.g. FIFO

• Is the network reliable?
  • E.g. will all messages eventually be delivered?

• Typically we make no other assumptions about the network

• Fail via *partition*
Byzantine Failures

The robot devil will return in lecture 26
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/

- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.

- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.