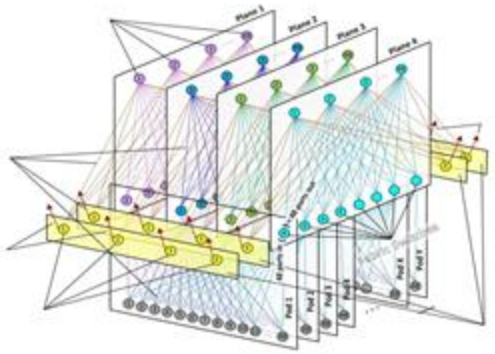
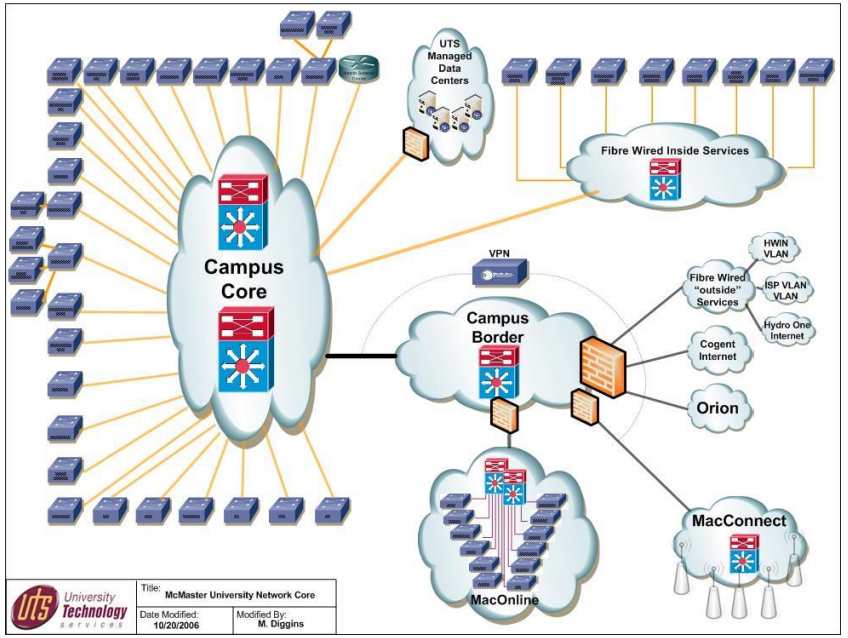
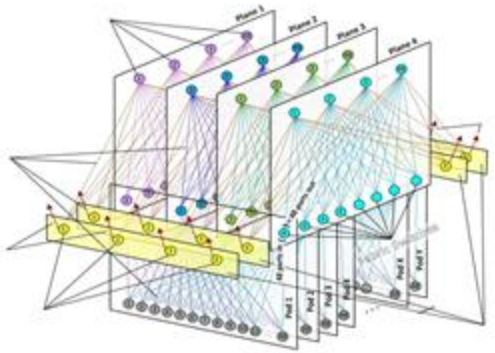


Large Systems:

Design +
Implementation +
Administration

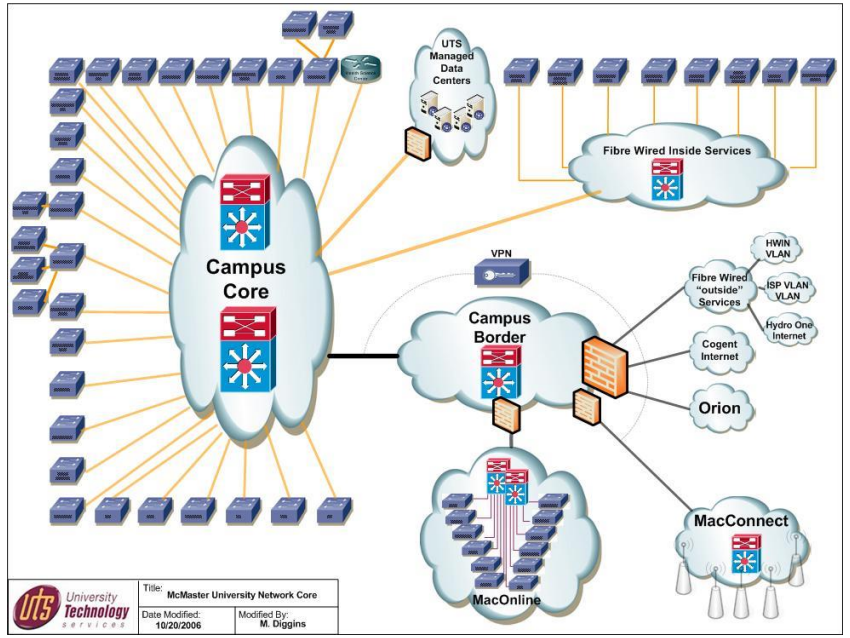
2024-2025





Large Systems:

Design + Implementation:



- Week4-L7: Fault Tolerance

Shashikant Ilager
shashikantilager.com

Recap: Replication

- Duplicate data or functionality on another server
- Reason: Performance
- Reason: Redundancy

Recap

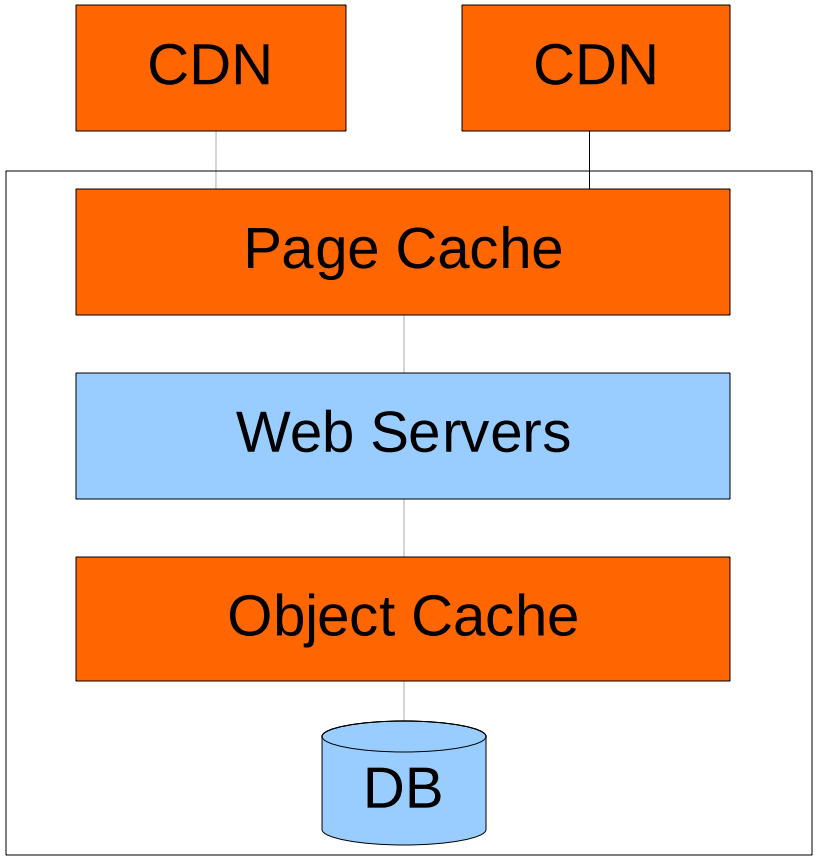
- What is:
  - Consistency
  - Availability
  - Partition Resistance?
- What is the CAP Principle?

- CAP Principle / Theorem
  - Can only have 2 out of 3!

Recap: CAP Principle

- C+A
  - Traditional database
  - MySQL, Spanner
- C+P
  - Read-only, or non-responding when partitioned
  - MongoDB, HBase, Redis, Memcachedb
- A+P
  - Always respond, even when outdated
  - CouchDB, Voldemort, Cassandra

Recap: Caching



E.g., Static Content: Images, CSS

E.g., composed pages

E.g., queries from DB

6

➢ Fault Tolerance

FT Requirements (Ch. 8)

- Availability: ready to be used
- Reliability: runs continuously without failure
- Safety: temporary failures are not catastrophic
- Maintainability: easy to repair

FT Metrics

- Availability:
  - Percentage of uptime, e.g. 99.999 %
  - Often referred to as number of nines: “5 nines”
  - 99.999 = 5.26 minutes per year
  - 99.9999 = 31.5 seconds per year (!)
- Reliability:
  - Mean Time Between Failures (MTBF)
  - Note: 1 ms down per hour is 99.9999 % availability
- Maintainability:
  - Mean Time To Repair (MTTR)

Types of Failures

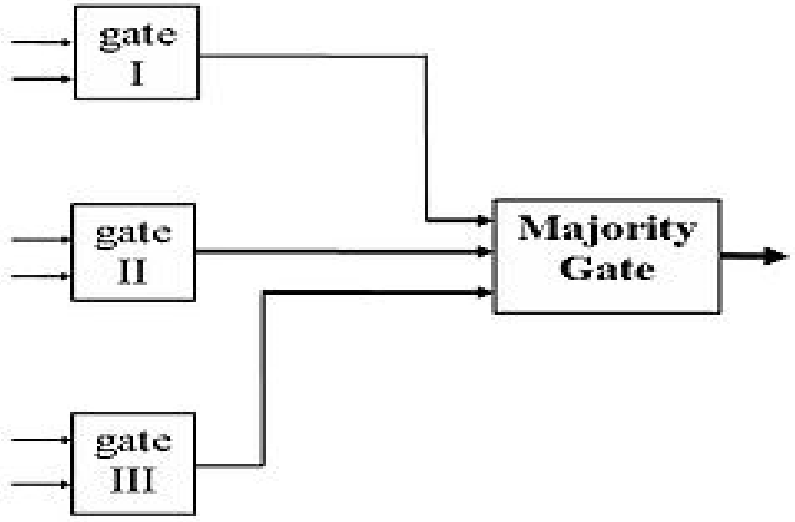
Type	Server Behaviour
Crash failure	Was working correctly, now halted.
Omission failure	Failed to respond
↳ Receive omission	Fails to receive incoming messages
↳ Send omission	Fails to send messages
Timing failure	Response outside specified interval
Response failure	Response is incorrect
↳ Value failure	Value of response is wrong
↳ State-transition failure	Deviates from normal flow
Arbitrary / Byzantine failure	Arbitrary responses at arbitrary times

Duration / Frequency of Failures

- Permanent
  - Appears and persists
- Intermittent
  - Appears now and then
- Transient
  - Appears and disappears forever

Failure Masking by Redundancy

- Physical redundancy
  - i.e., Replication
  - Double/Triple Modular Redundancy (DMR/TMR)
- Information redundancy
  - Error detection (parity bits, checksums..)
  - Error correction
- Time redundancy
- Software redundancy
  - N-version programming



Redundancy and Types of Failures

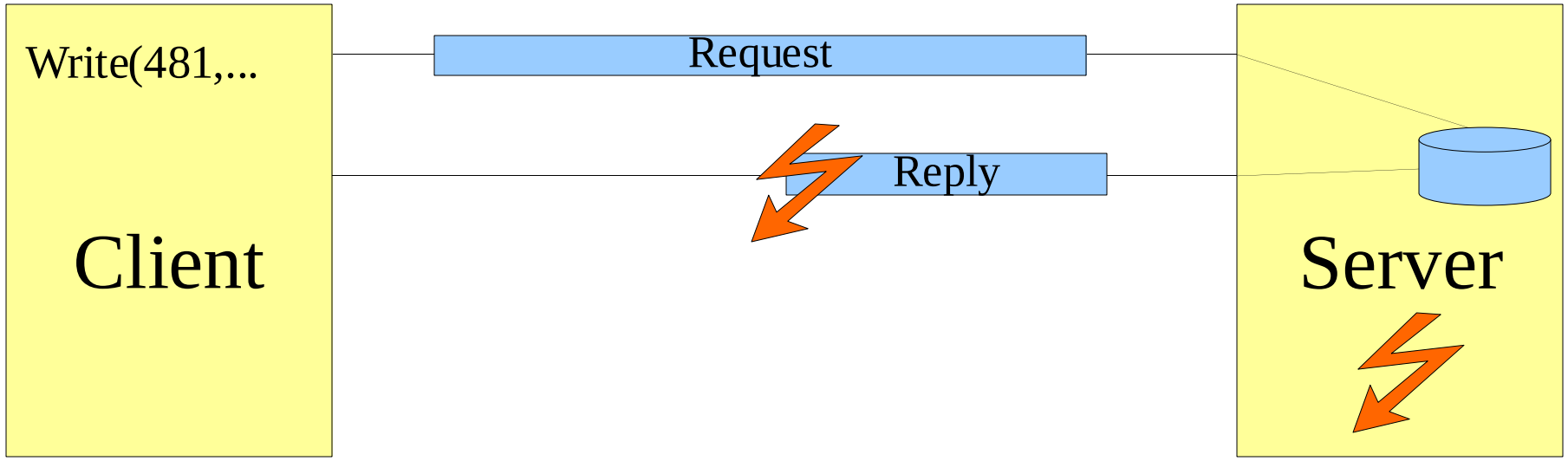
- How many replicas needed?
- If failures are Crash failures:
  - Need k+1 replicas to handle k failures
  - Last replica can provide service / answer
- If failures are Byzantine (e.g. wrong answers)
  - Need 2k+1 replicas to handle k failures
  - Need k+1 good replicas to outvote k bad ones!
    - k+1 (good ones) + k (bad ones) -> 2k+1 total replicas

Recall: Communication (Ch. 4)

- Client-to-Server
  - Hide distribution & failures
- Server-to-Server
  - If multiple Servers providing Service, you need
  1. Group communication
  2. Coordination
  - Despite communication & server failures

Client-Server Problems

- Recall Remote Procedure Call (RPC)
- What if requests or replies are lost?
- What if a server crashes?



RPC with Failures

5 classes of failures:

1. Client cannot locate server
2. Request message from client is lost
3. Server crashes after receiving request
4. Reply message from server is lost
5. Client crashes after sending request

1. RPC Client Cannot Locate Server

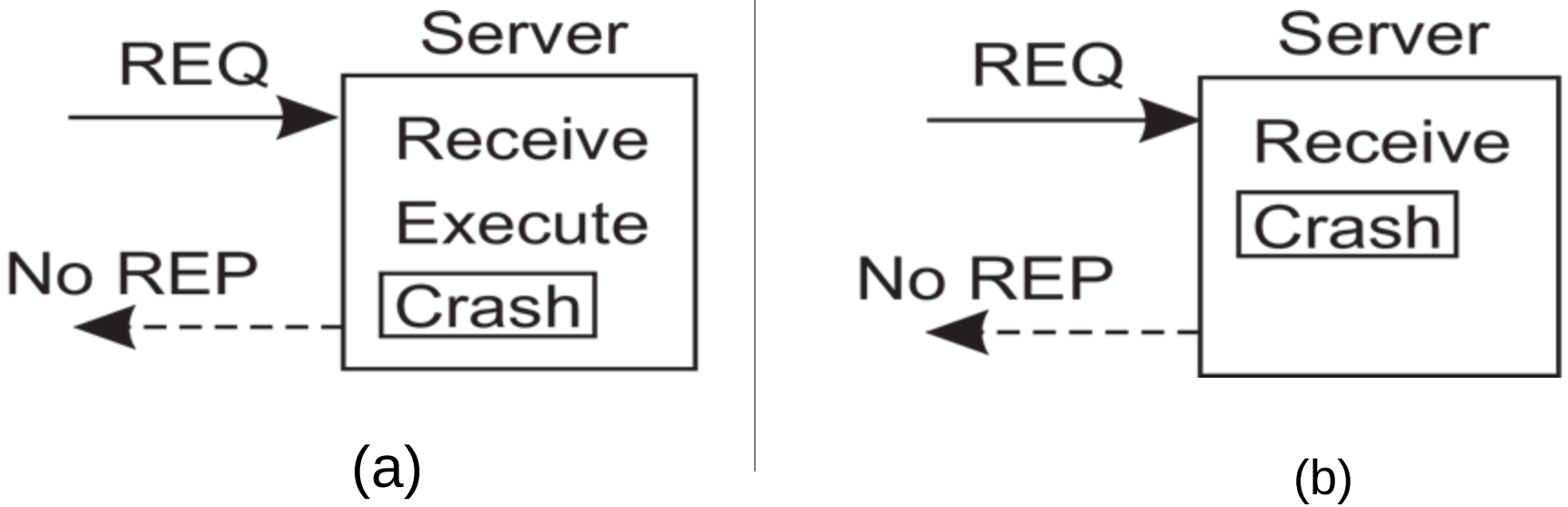
- E.g. All servers down
- E.g. Server has new version of API
- Issue: how to signal this when distribution is supposed to be transparent?

2. RPC Request Lost

- RPC middleware layer can resend
- If no reply has been received after T seconds
- If all requests are lost: back to Problem 1: Locate Server

3. RPC Server Crash

- When exactly did the server crash?



- Correct treatment differs for (a) and (b)!

In case (a) the RPC middleware layer should report an error to the client (non-transparent)
In case (b) the RPC middleware layer could just repeat the request to another or rebooted server (transparent). Can we?

RPC Semantics

- Want: “Exactly Once” semantics
  - Whatever happens, procedure only carried out once
  - Unfortunately, impossible to achieve, in general
- Why impossible?
  - Cannot tell, in general, whether call was executed
  - Idea was that RPC would replace normal procedure call transparently.
  - Turns out you need
    - Idempotent operations or
    - e.g. atomic transactions to make it work

Idempotent Requests

- Idempotent means
  - Can be repeated without altering state
  - Or having side-effects
  - E.g. read first 1024 bytes of file (when no writes)
- Not all requests can be made idempotent
  - E.g. transfer €100 from bank account

RPC Semantics

- Want: “Exactly Once” semantics
  - Whatever happe t once
  - Unfortunately, i ral
- Why impossible?
  - Cannot tell, in ge uted
  - Idea was that RP cedure call transparently.
  - Turns out you ne
    - Idempotent operations or
    - e.g. atomic transactions to make it work
  - Not generally applicable

RPC Semantics

- Alternative proposals:
  - At-Least Once
  - At-Most Once

RPC with Failures

5 classes of failures:

1. Client cannot locate server
2. Request message from client is lost
3. Server crashes after receiving request
4. Reply message from server is lost
5. Client crashes after sending request

4. RPC Reply Lost

- Simplistic strategy:
  - Repeat request
  - Works if requests are idempotent
- Again: only application-specific solutions

4. RPC Reply Lost

- Alternative protection
  - Add request ID in request message
  - Refuse request if duplicate
  - Downside: need to keep state per client
  - For how long?

5. Client Crashes After Request

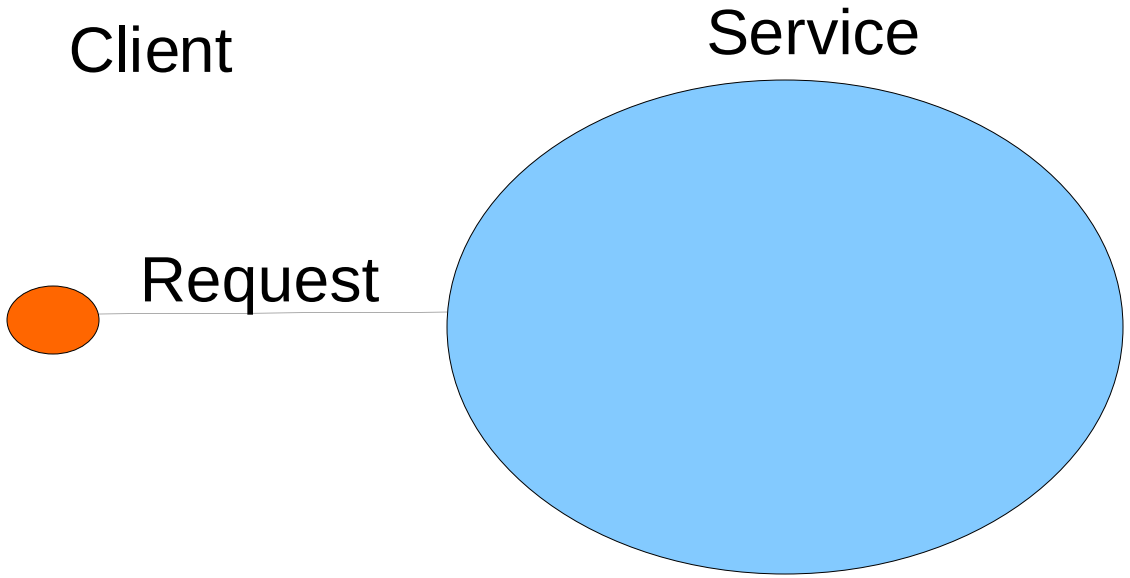
- Execution of request is now an orphan
- Can be wasteful if resource-intensive, long running
- Reply may confuse rebooted client
- 4 solutions
  1. Orphan extermination
  2. Reincarnation
  3. Gentle reincarnation
  4. Expiration

RPC & Distribution Transparency?

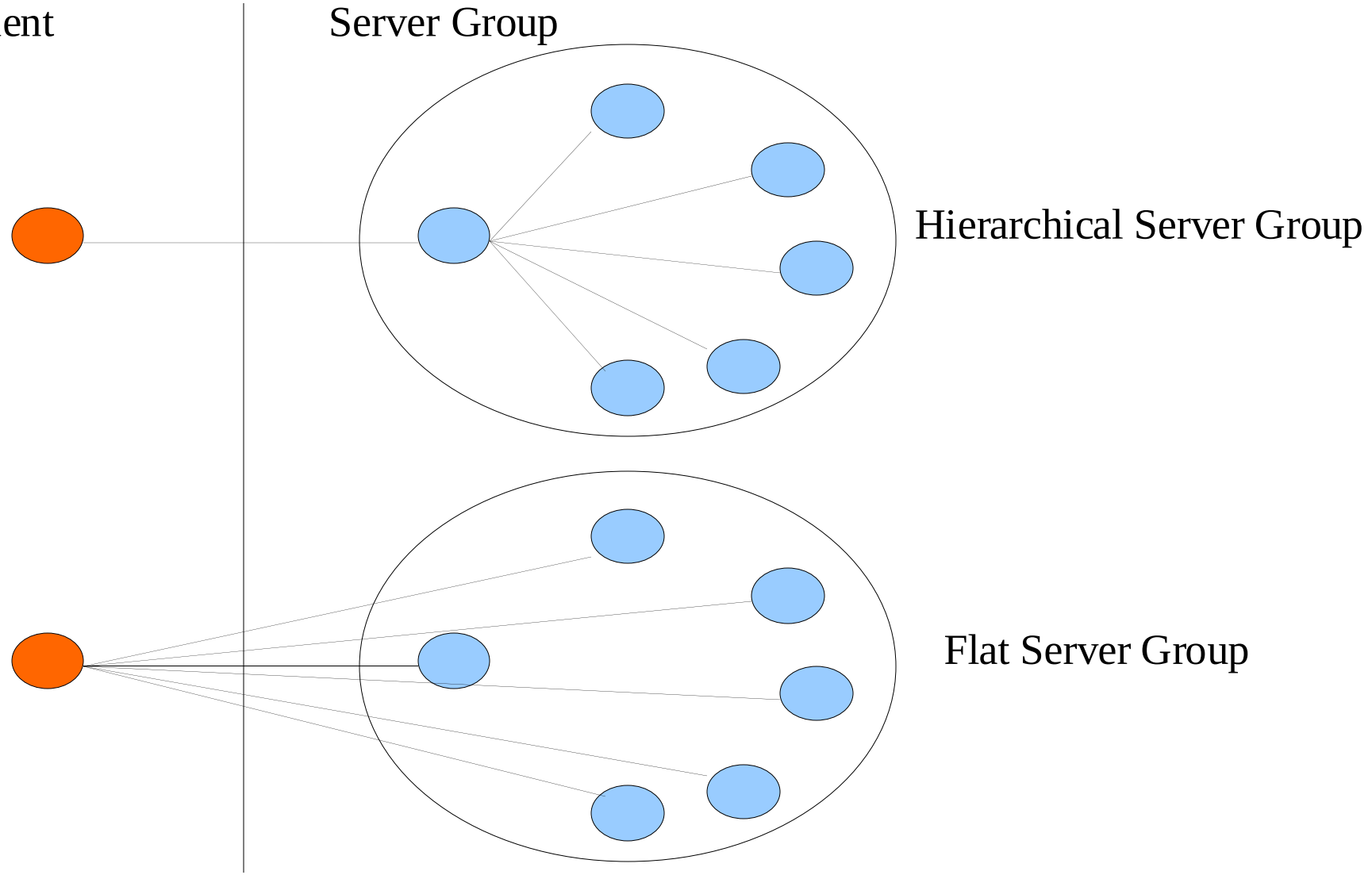
- Some say: Forget about full transparency
  - "A Note on Distributed Computing" Jim Waldo *et al*. Sun Microsystems
- Question remains how to handle crashes without big performance loss
  - E.g. Synchronous writes to disk expensive

FT Server-to-Server Communication

- Model



Service Implemented by Server Group



Server Group Problems

- Consider a flat group
- Want atomicity:
  - Request for operation must reach all non-faulty servers
- Note: assumes failure can be reliably detected
  - Not true for asynchronous systems (recall Intro)
  - Cannot distinguish slow from failed
  - When is a server considered failed?
- Assumes group composition is known

Server Group Problems

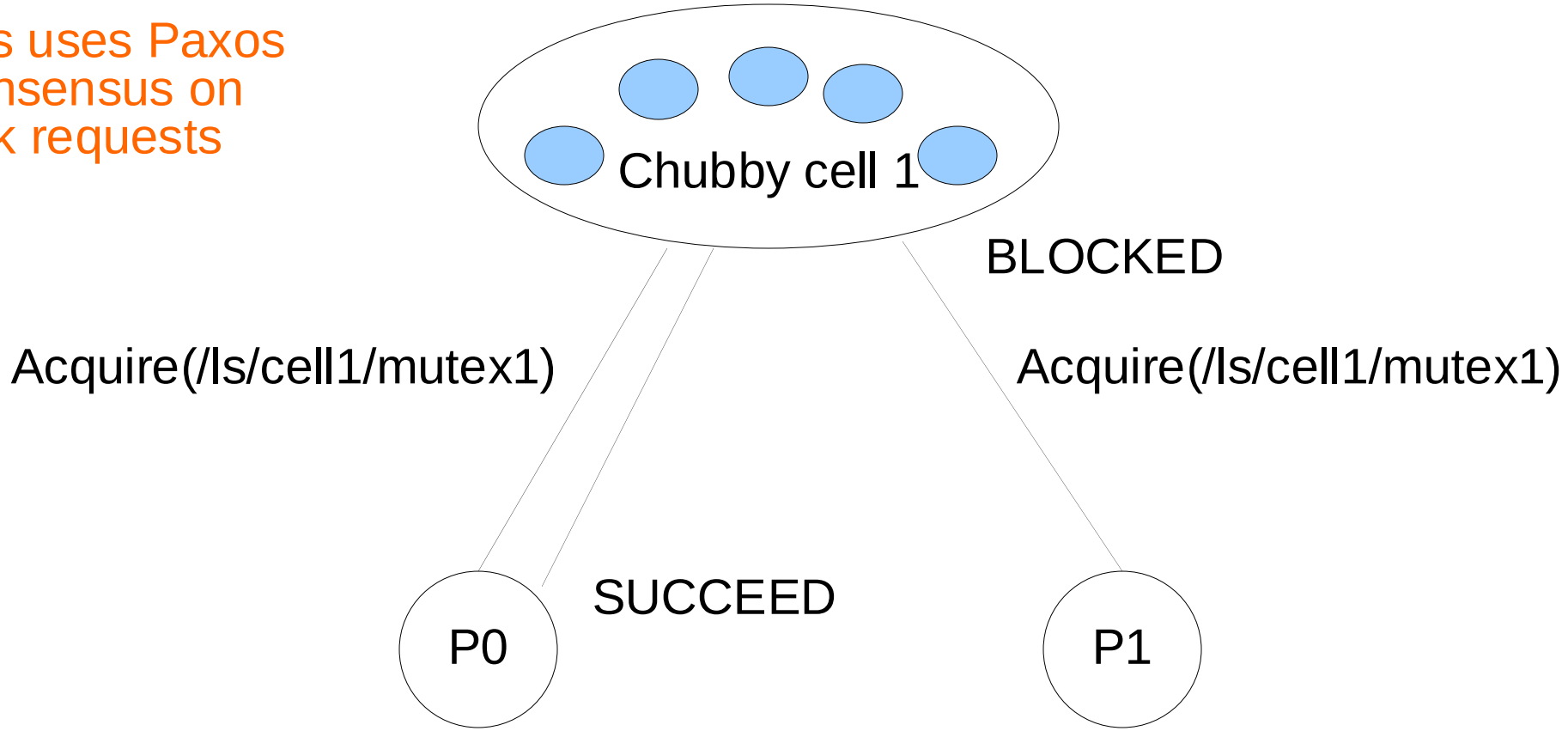
- Want total order:
  - each non-faulty server executes the same commands in the same order as every other non-faulty server
- In other words, the group must reach consensus on:
  - group composition and
  - operation order
  - while members and messages can be lost at any time...

Consensus Solutions

- Failures can be detected in practice
- Paxos [Lamport, 1989]
  - Used by Google in various services
  - Used by XtreemFS, Ceph, Chubby
- Raft [Ongaro and Ousterhout, 2014]
  - Designed to be more understandable than Paxos
  - Formally proven correct
  - Many implementations exist
    - https://raft.github.io/#implementations

MutEx at Google: Chubby

Cell replicas uses Paxos to reach consensus on order of lock requests



Source: "Distributed Systems" 5th Ed, Coulouris et al, p. 941

Coordination in/with etcd

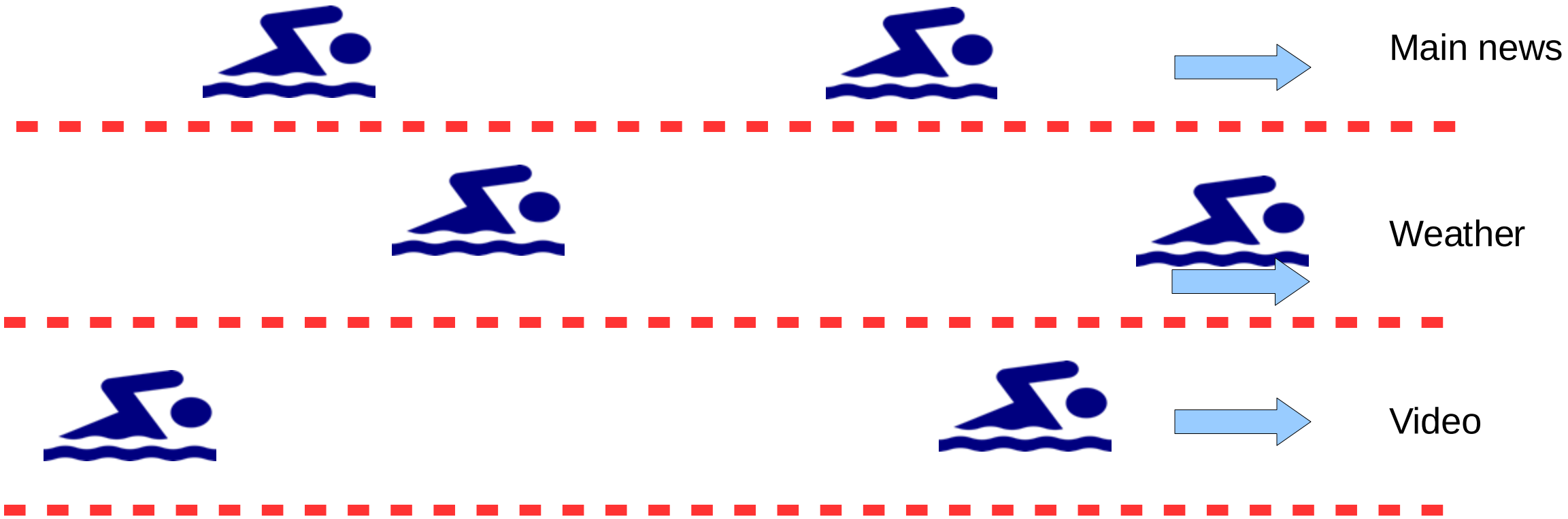
- etcd cluster uses RAFT
- RAFT elects leader in cluster
- NOTE:
  - etcd needs an internal leader for FT via RAFT
  - clients can use etcd to elect leaders for own app (via atomic key,value writes)
  - Not same!

Fault Isolation

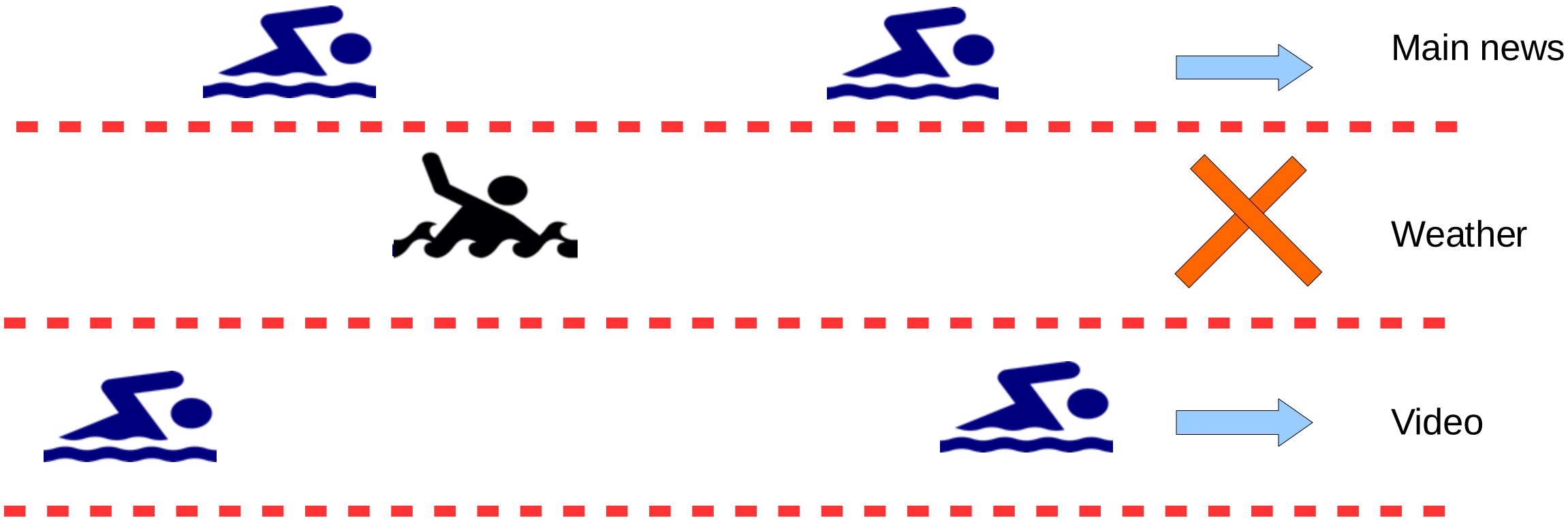
- Do not want your whole system to fail when one component fails
- Need to segment / partition your system for fault tolerance
- E.g. a news Web site
  - Partition into e.g. Main news, Weather, Videos
  - Main news continues to be served when Weather, Videos down
- Partition is called a failure domain or swim lane

Source: Scalability Rules, 2nd Ed. Abbott and Fischer, 2017, Ch. 9

Swim Lanes



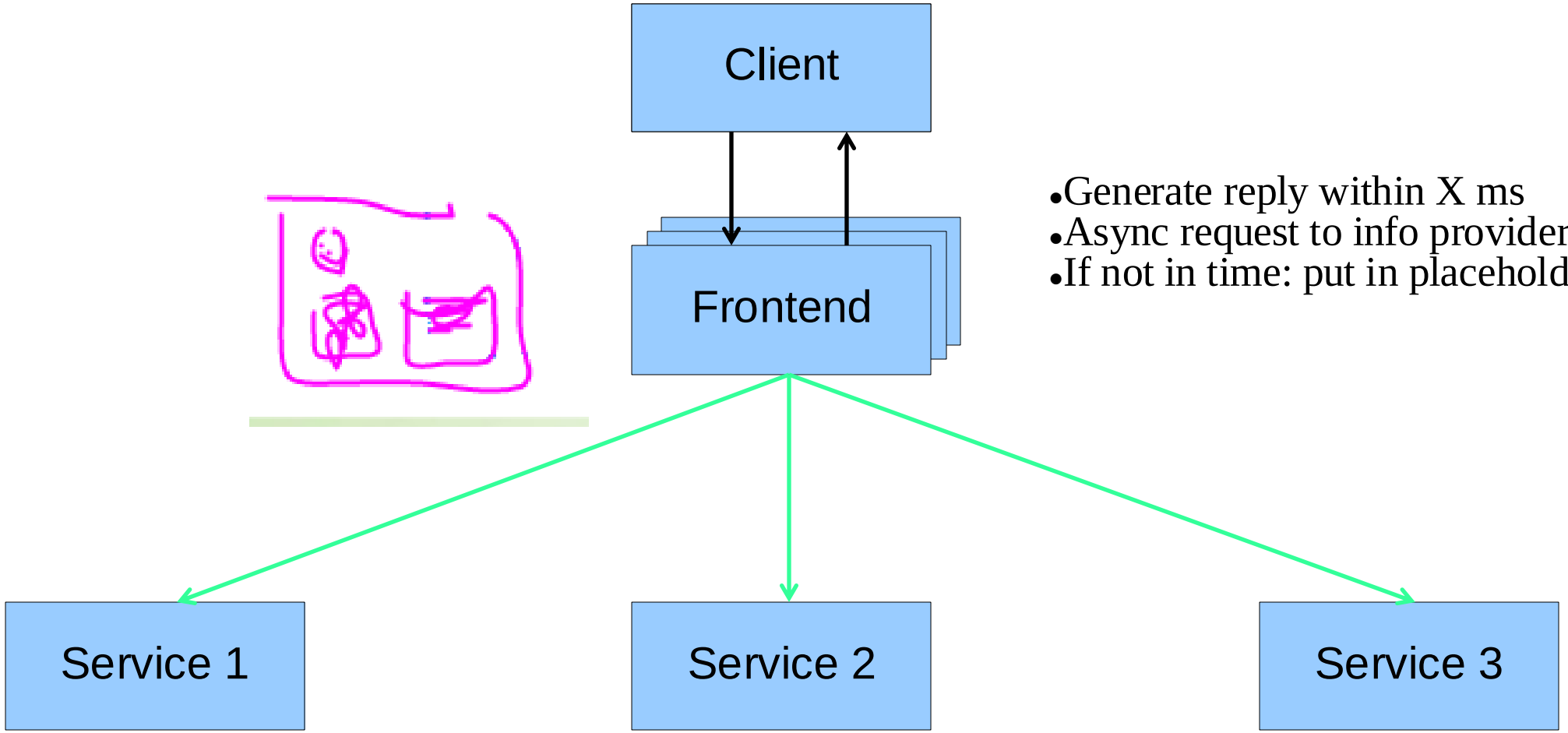
Swim Lanes



Swim Lane Independence

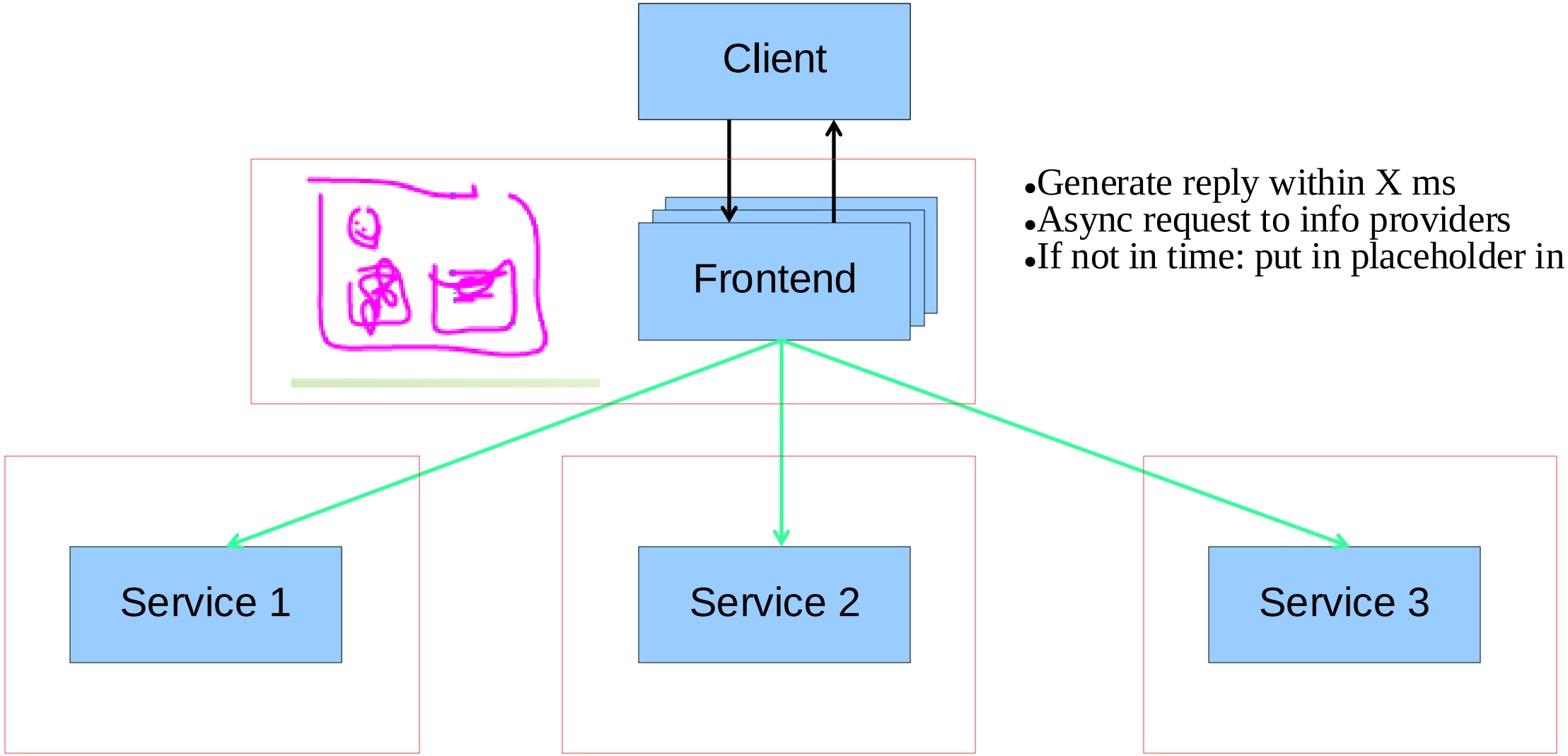
- Components in a swim lane must not depend on components in other swim lanes
- I.e. separate network, machines, software
- Also no synchronous RPCs to software in different lane
  - Would block your lane if other lane halted
  - Use async/deferred synchronous RPCs
  - E.g. try to get Weather data, if no reply in time, do not include on Web page
  - Ability to turn dependencies on/off

Modern Web service



- Generate reply within X ms
- Async request to info providers
- If not in time: put in placeholder in reply

Modern Web service: Swim Lanes



- Generate reply within X ms
- Async request to info providers
- If not in time: put in placeholder in reply

Swim Lane Advantages

- Availability:
  - One failure does not kill whole system
- Incident detection:
  - If a failure occurs in a swim lane, only need to debug that swim lane
- Scalability:
  - Fault-isolated services can grow independently
- Cost
  - Spend less on less critical components
  - Smaller problem size, easier engineering

Source: Scalability Rules, 2nd Ed. Abbott and Fischer, 2017, p. 127