Department of Computer Science
University of California, Los Angeles

Computer Science 134: Distributed Systems

Fall 2024
Prof. Ryan Rosario

Lecture 2: October 2, 2024

Outline

1. From Last Time
2. My Motivation
3. The Challenge of Coordination
4. System Models
5. Architectures

1 From Last Time

2 My Motivation

3 The Challenge of Coordination

4 System Models

5 Architectures

Humble Bundle

For those of you that are into systems, there is current a Humble Bundle for systems books:

28 e-books for $25!

Last Time

Last time we introduced the concept of a distributed system, what it is, examples, and the various challenges associated with them.

When we write code on one machine on a single core, only one failure can happen at a time – in the code, or something physical.

Once we start using multiple threads or cores, and multiple machines, multiple faults can happen at the same time either on the same machine, or different machines.

More in the Recording

In the recording (1:01 - 1:38), I expanded on a few concepts, which is why it is 2 hours.

In particular, I talk more about fault tolerance and sensor networks.

It's only background knowledge, but might be of interest.

Multiple Faults

Unlike a simple piece of code executing on one core of one machine, in a distributed system faults:

- can occur in parallel or cascade (e.g. packet corruption *and* a CPU computation error)
- are dependent on the environment – a very complex environment where many processes and phenomena occur (non-deterministic)
- can occur at any logical level of the system, not just low-level physical machines.
- can spread across the entire system if not detected early (insidious)
- can be transient
- can depend on the order in which a process was executed (race condition)
- can occur long after the system was deployed

Multiple Faults (contd.)

Example of Multiple Parallel Faults: On March 11, 2011, a 9.1 earthquake struck Japan. The resulting damage and tsunami caused a nuclear accident at the Fukushima Daiichi nuclear plant.

Multiple Faults (contd.)

The power plant used electricity to pump in seawater to keep nuclear fuel "cool" and safe, and to produce steam to produce power. **In a catastrophe, backup power is used**.

1. Earthquake hits, power for the pumps fails and backup diesel generators fire up. Then a tsunami hits.
2. **Diesel Generators Fail**
   1. the unprotected diesel generators are flooded and fail
   2. additional generators in a basement fail due to as water flows down
   3. even more diesel generators on an elevated hill fail due to height of waves
3. **Batteries Fail**
   1. A bank of DC batteries fail. Water and electricity do not mix!

Why to Build a Distributed System?

So, if it's so hard, why build one using the goals we discussed earlier?

1. Distribution is **inherent** in many applications (e.g. sensor networks, email across the Internet)
2. **Resource sharing** – insufficient resources on a single node (e.g. big data, large DNNs)
3. **Improved performance** by locating resources close to users (e.g. sharding decreases latency)
4. **Better reliability** – partial failures do not crash the system.

The Golden Rule

If you do not need to distribute, don't!

It's hard. So only do it if it is truly necessary. Otherwise run the process on a single node.

Our Mental Model of Distributed Systems

We will discuss distributed systems with the following perspectives:

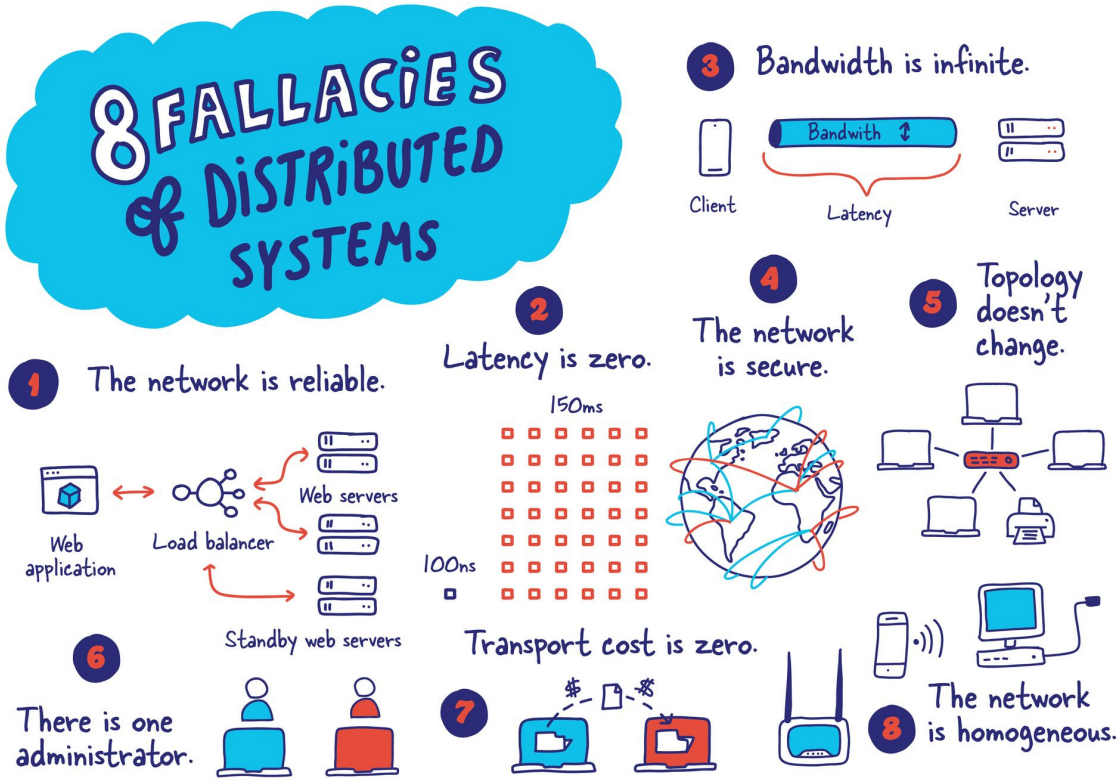
1. Architectural
2. Processes and implementation
3. Communication
4. Coordination
5. Consistency
6. Fault Tolerance
7. Security

Eight Fallacies of Distributed Systems

Beginners may make several incorrect assumptions when designing a distributed system (and the pay for it later):

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology does not change
6. There is one administrator
7. Transport cost is 0
8. The network is homogeneous

Eight Fallacies of Distributed Systems (contd.)

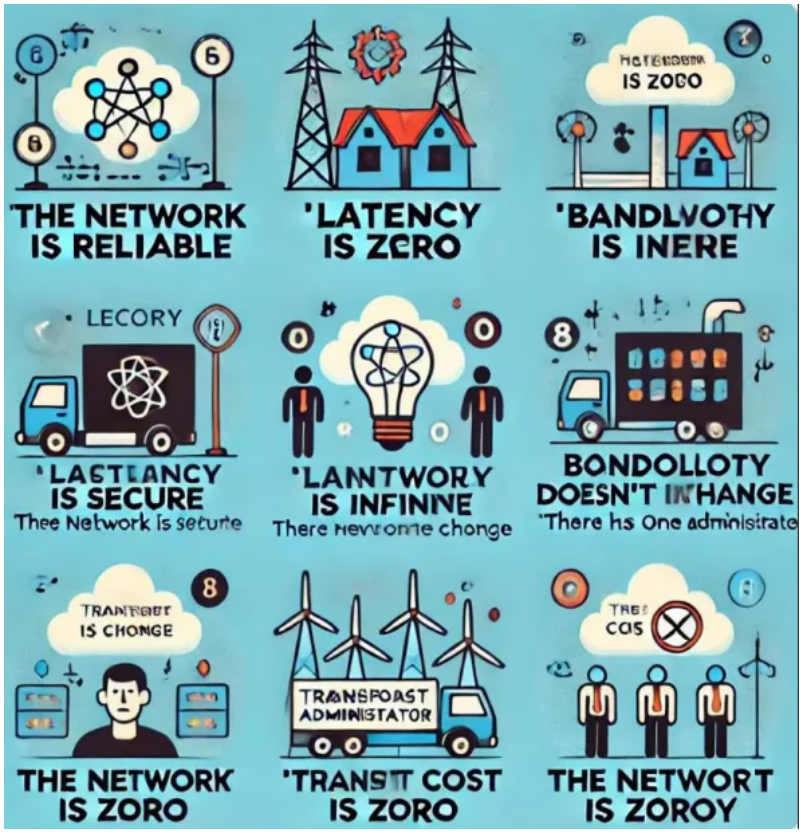


Source: Mahdi Yusuf, Architecture Notes blog

Eight Fallacies of Distributed Systems (contd.)

Source: Mahesh Saini, Experience Stack blog

Eight Fallacies of Distributed Systems (contd.)



Source: Mehmet Ozkaya

1 From Last Time

2 My Motivation

3 The Challenge of Coordination

4 System Models

5 Architectures

How I Accidentally Got into Distributed Systems

Picture it. Long ago. Somewhere in Boelter Hall. About 30 of us take a course, CS 246 - Web Information Management.

Our class project was to perform some kind of web research. Most of us wrote Facebook crawlers and did something with the data.

How I Accidentally Got into Distributed Systems (contd.)

Back then, only college students and alumni could join Facebook. The UCLA network had about $\approx$ 50,400 users at the time.

The site was fairly static – not as much JavaScript interactivity features. It was easy to scrape user data in one shot.

How I Accidentally Got into Distributed Systems (contd.)

The original architecture was a single-threaded scraper written in Python, using a very slow (at the time) library called BeautifulSoup.

The code:

- Traversed the friend graph using Breadth First Search, keeping track of profiles I already scraped
- Maintained a queue in RAM of profiles/friends to scrape
- Scraped the profile
- Parsed the data from the profile into a file
- Didn't save any intermediate steps (oops)
- Had poor exception handling

There was no separation of concerns here and it was a terrible design. Weeks of data loss and no progress followed.

How I Accidentally Got into Distributed Systems (contd.)

Many failures occurred:

1. Bugs in the code causing unhandled exceptions, losing work and the work queue (a Python list in RAM)
2. Unexpectedly corrupt or incomplete data from Facebook causing exceptions
3. Transient errors on the network, or Facebook HTTP server
4. Some HTTP requests blocked indefinitely
5. Exhausted memory somehow (2GB or so), causing disk thrashing that crashed the machine
6. **I was also being rate limited via IP address, reducing throughput**

Days and days passed with no progress. **I needed to speed this up – increase throughput and fault tolerance.**

How I Accidentally Got into Distributed Systems (contd.)

Instead use a message queue model:

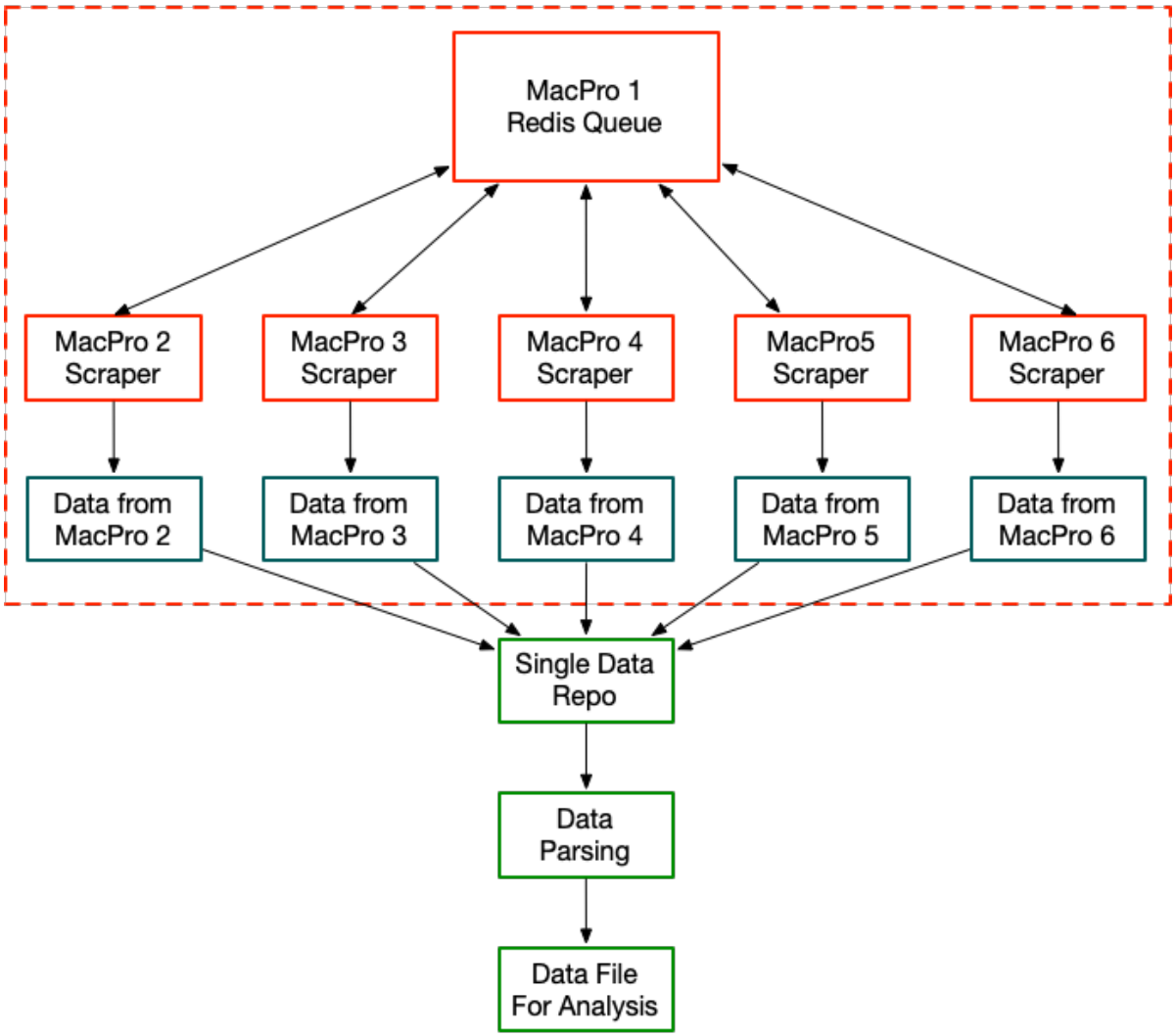
- Split the original scraper code into two components:
  1. the scraper itself, which saved the HTML to disk (What a concept!) and extracted friend lists online
  2. a separate offline parser that extracted data from the HTML pages and stored in a series of files
  3. (add better exception handling and TTLs to HTTP requests)

How I Accidentally Got into Distributed Systems (contd.)

- Add a work queue (Redis) in RAM, on a master node, keeping track of which profiles needed to be scraped. This was backed by the disk.
  1. the scraper pushed and pulled work (profile IDs) to/from the queue
  2. the scraper caught exceptions and pushed the profile to the back of the queue (transient error) with a TTL for retry
  3. if we exceed the TTL, write the profile ID to a failure log
  4. add recovery code to re-populate the queue if the node fails

To be nice, I made 1 request per second. Across the 5 machines, this became 5 requests per second. 3 weeks $\rightarrow$ a couple of days.

How I Accidentally Got into Distributed Systems (contd.)



1 From Last Time

2 My Motivation

3 The Challenge of Coordination

4 System Models

5 Architectures

A Thought Experiment

We could avoid bringing up some significant concepts (namely TCP) from CS 118 by considering two classic thought experiments in distributed systems:

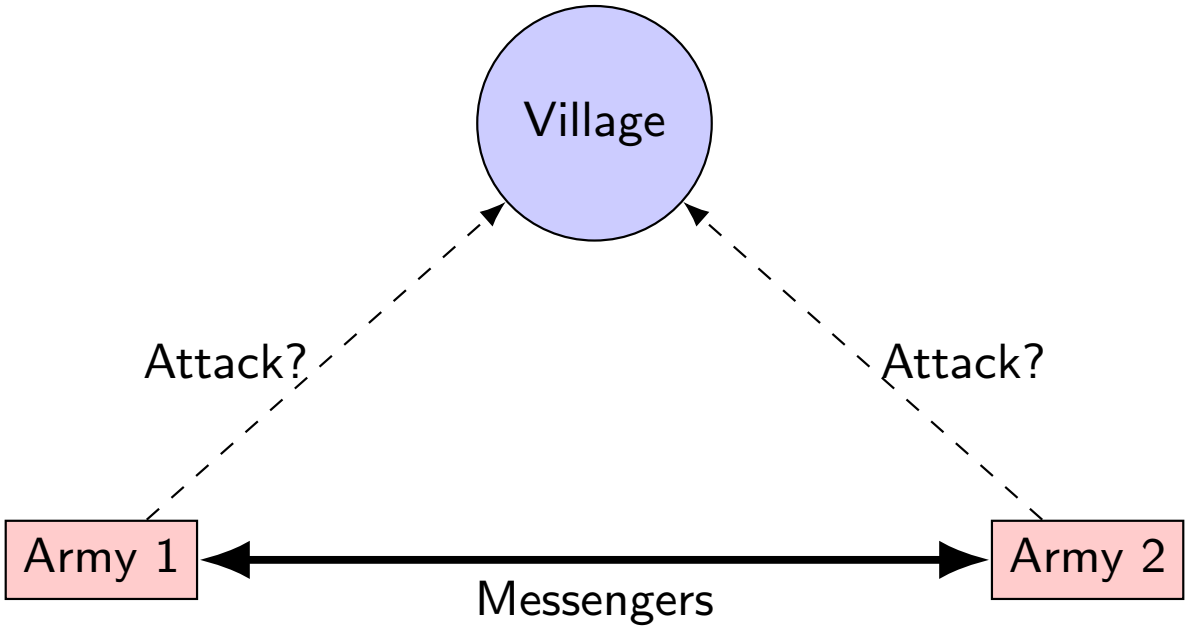
1. The Two Generals Problem
2. The Byzantine Generals Problem

The Two Generals Problem

Imagine 2 generals, each leading an army, who want to capture a village. The village defenses are strong. If only one army attacks, the **army** will be defeated.

But, if both armies attack at the same time, they successfully capture the village.

The Two Generals Problem (contd.)



Army 1	Army 2	Outcome
does not attack	does not attack	nothing happens
attacks	does not attack	army 1 defeated
does not attack	attacks	army 2 defeated
attacks	attacks	**village captured!**

The Two Generals Problem (contd.)

Trouble in the river...

The Two Generals Problem (contd.)

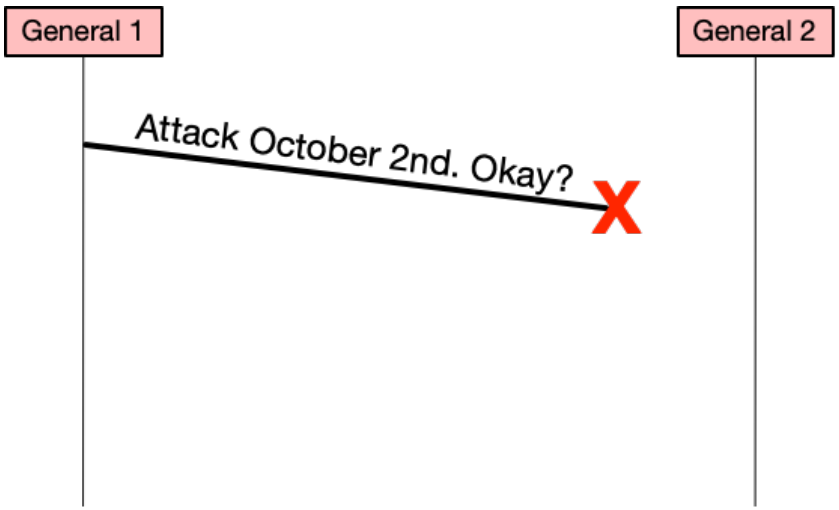
But there is trouble in the river... The two armies are camped kind of far from each other...

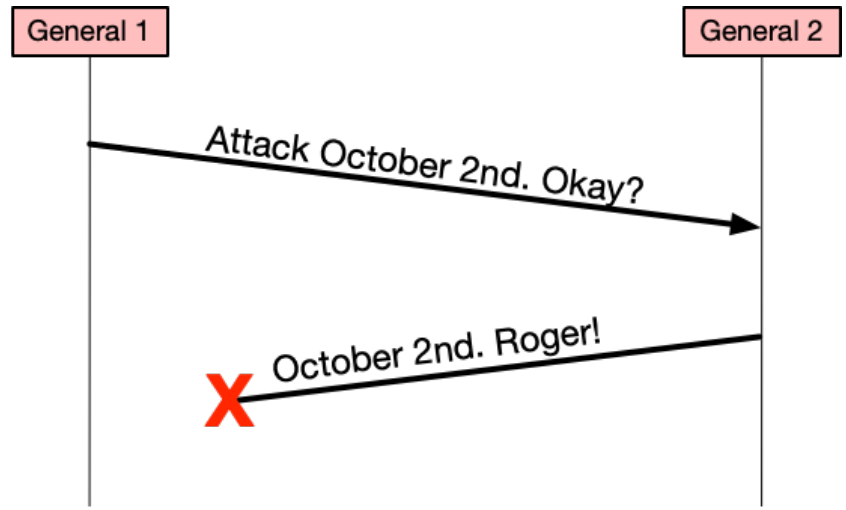
1. so they need to **coordinate** their attacks somehow, so they use messengers to **communicate** between the two armies...
2. the messengers must pass through the river, which is territory controlled by the village...
3. which means they sometimes get captured...

This means the message between from army i to army j may get lost. But, general i cannot know if general j received the message unless he gets an explicit reply from j.

The Two Generals Problem (contd.)

In general 1's perspective, the situation could be either of the following, but he doesn't know which:





The Two Generals Problem (contd.)

So what *protocol* should the generals use to coordinate the attack?

1. General 1 can just go ahead and attack without needing to receive a response from General 2 first, but then General 1 risks having his army captured.
2. General 1 can wait for a response from General 2, but it's possible that army 2 goes ahead and attacks and gets captured.

Using a single messenger, what might we try next? Why won't it work?

The Two Generals Problem (contd.)

1. General 1 always attacks, even if there is no positive response from General 2
   - Send many messengers to increase the probability that at least one gets through.
   - If all of them are captured, Army 1 will be captured.
2. General 1 attacks if and only if there is a positive response from General 2.
   - General 1 and his army is safe.
   - General 2 knows he is safe if and only if General 2's positive response is received by General 1.
   - Otherwise, General 2 is in the same situation as General 1 under strategy 1 above.

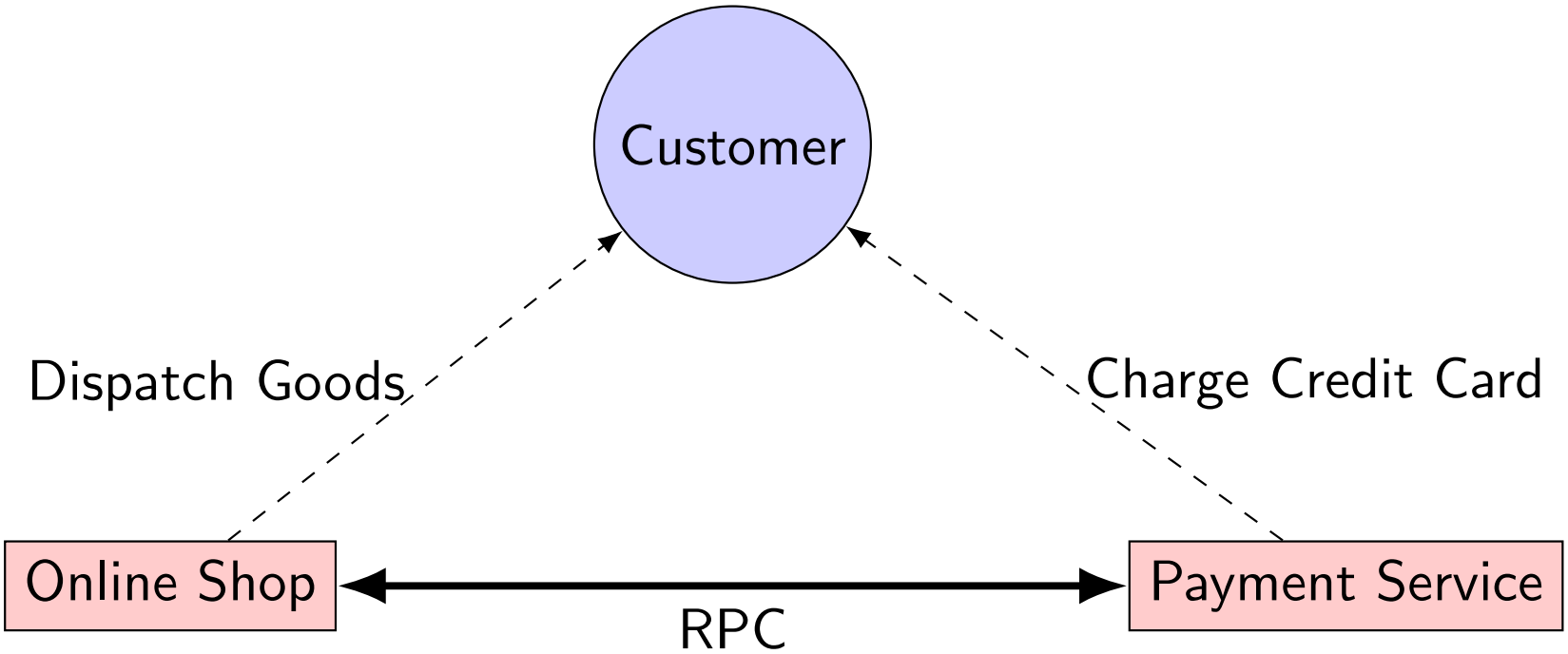
There is no common understanding between the two generals. The only way to understand is to communicate!

The Two Generals Problem (contd.)

Point: The generals can never be **certain**. This is not solvable.

But in some use cases, it can be solvable...

Similar to Two Generals, but Solvable



Online Shop	Payment Service	Outcome
does not dispatch	does not charge	nothing happens
dispatches	does not charge	shop loses money
does not dispatch	charges	angry customer
dispatches	charges	**everyone happy**

The Byzantine Generals Problem

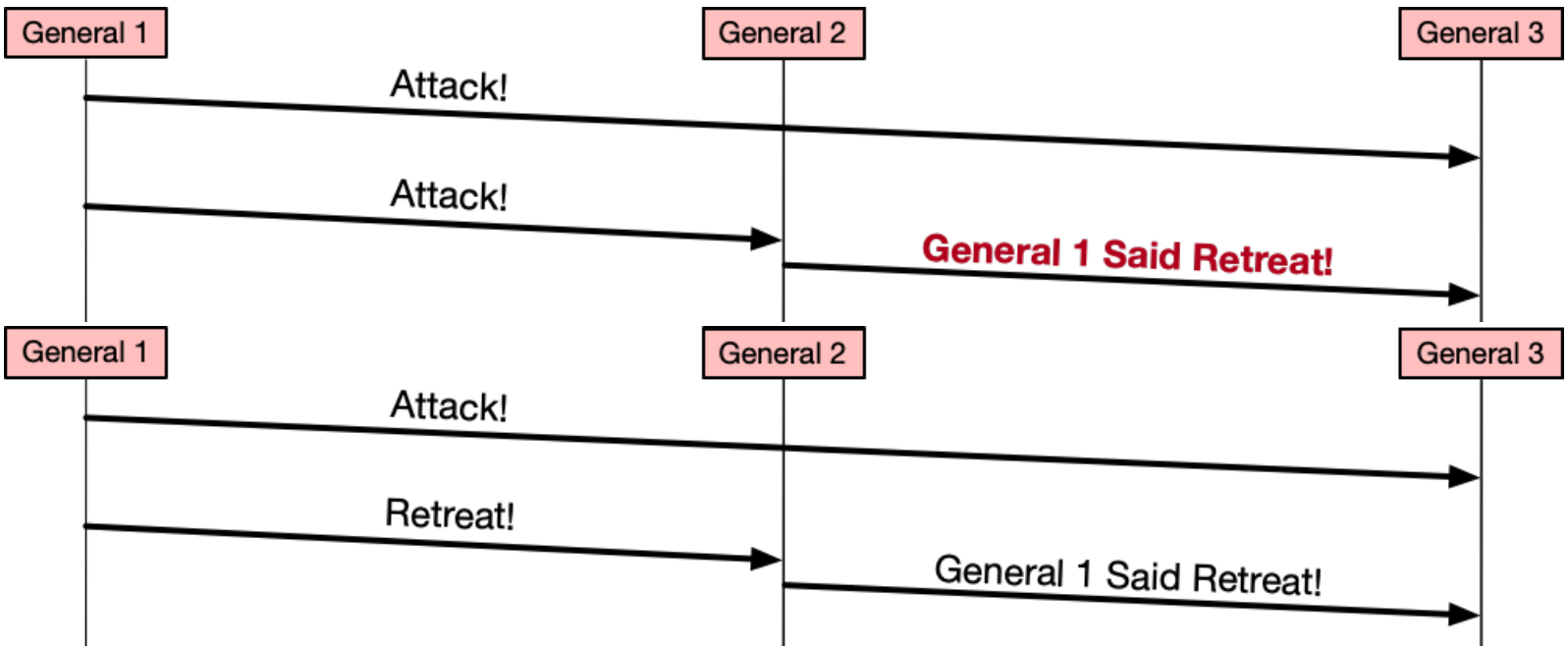
The Byzantine Generals Problem (contd.)

The Byzantine Generals Problem is similar to the Two Generals Problem except –

1. There are 3+ generals/armies.
2. When a message is sent, it is always received (no captured messengers).
3. One or more generals may be **traitors** – they intentionally mislead other generals. Generals are *malicious* or *honest*.

The Byzantine Generals Problem (contd.)

Here, General 3 receives two contradictory messages from Generals 1 and 2 and does not know that General 2 is malicious. From General 3's point of view, both of these are possible:



The Byzantine Generals Problem (contd.)

So, we have many social problems here:

- The honest generals do not know who the malicious generals are.
- The malicious generals can collude.
- **The best we can do is get the honest generals to agree.**

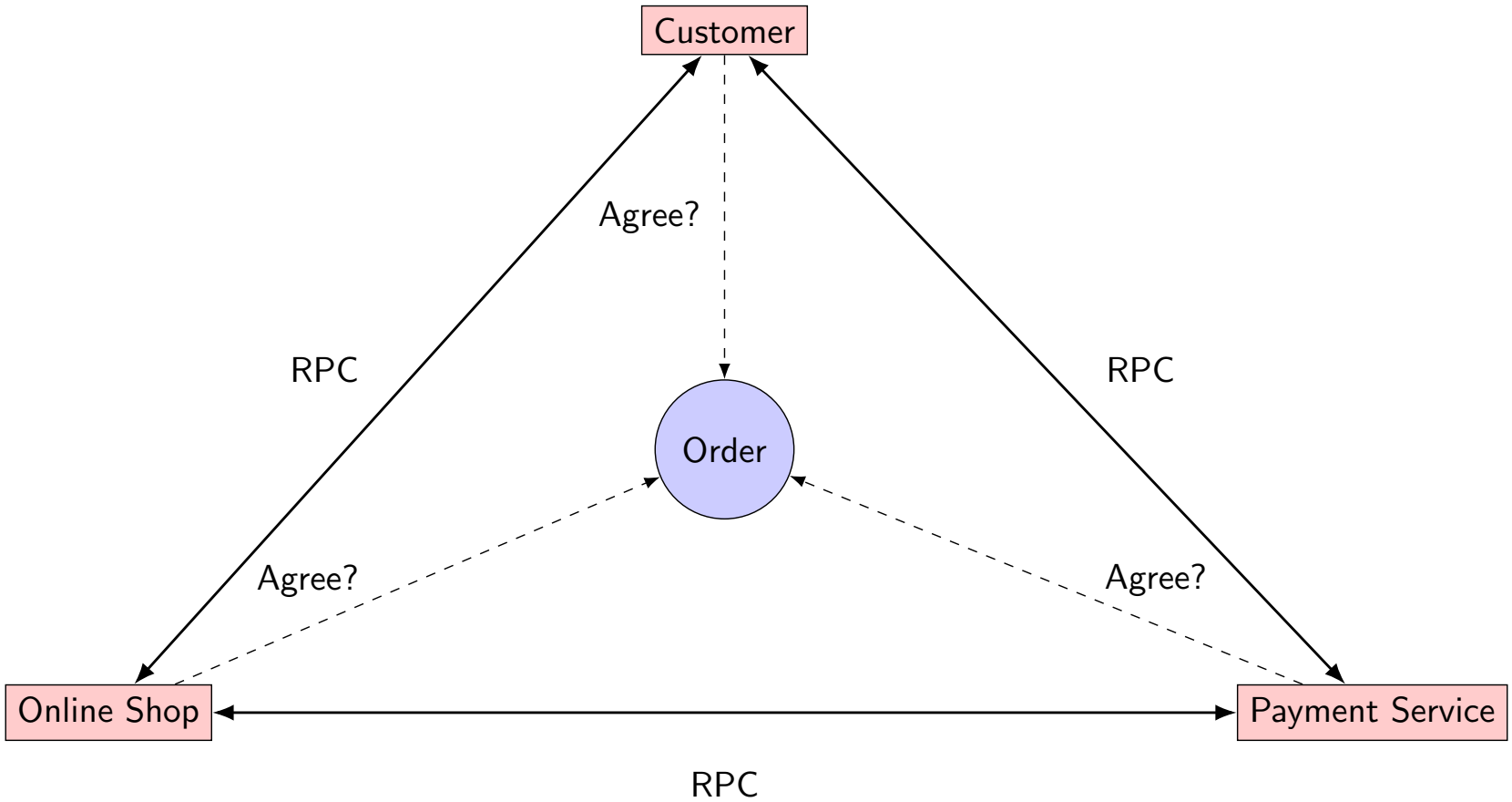
This is difficult. It can be mathematically proven that in a system with malicious generals an communication delay, this problem can be solved only if fewer than 1/3 of the generals are malicious.

The Byzantine Generals Problem (contd.)

The Byzantine Generals Problem is easier, though still challenging, if generals used digital signatures (cryptography), this would allow General 2 to prove to General 3 what General 1 said.

In practical terms, distributed systems can have trust issues when components are not all managed by the same entity.

The Byzantine Generals Problem (contd.)



The Byzantine Generals Problem (contd.)

Systems that explicitly acknowledge the possibility of malicious actors are called *Byzantine fault tolerant*.

This concept is important to for blockchain and cryptocurrency, where the system tries to provide guarantees even if some of the actors are trying to cheat or undermine the system.

We will discuss this later in the quarter.

The Byzantine Generals Problem (contd.)

The Byzantine empire is named after the capital city Byzantium of Constantinople (now Istanbul, Turkey). The word Byzantine is used in the sense of "excessively complicated, bureaucratic, devious." The exact etymology is unclear.

There is no evidence that Byzantine generals were any more prone to intrigue or conspiracy than those elsewhere.

1 From Last Time

2 My Motivation

3 The Challenge of Coordination

4 System Models

5 Architectures

System Models

When designing a distributed system, a *system model* specifies our assumptions about what faults may occur.

Our two thought experiments:

1. Two Generals Problem was a model of networks and lossy links
2. Byzantine Generals Problem was a model of node behavior.

In a real system both the nodes and the networks may be faulty.

System Models (contd.)

We need a model consisting of:

- Network behaviors (e.g. message loss)
- Node behaviors (e.g. crashes)
- Timing Behavors (e.g. latency)

We must choose a model for each. A series of assumptions against which we discuss faults.

System Models: Network

Even in the most carefully engineered systems with redundant links, things can go wrong for physical reasons, overloading, DDOS attacks etc.

Most of our systems assume that the network provides bidirectional message passing between a pair of nodes. This is called *point-to-point* or *unicast* communication.

Later we will learn more about *broadcast* or *multicast* communication.

System Models: Network (contd.)

Most distributed algorithms assume of these network models under point-to-point communication:

1. **Reliable** links
2. **Fair-loss** links
3. **Arbitrary** links

A **network partition** occurs when a link drops/delays all messages for an extended period of time.

System Model: Node

An algorithm executes on a node assuming one of the following:

1. **Crash-stop / fail-stop**.
2. **Crash-recovery / fail-recovery**.
3. **Byzantine / fail-arbitrary**.

System Model: Timing/Synchrony

We assume one of the following for nodes and networks:

1. **Synchronous:**
2. **Partially Sychronous:**
3. **Asynchronous:**

Note that these definitions are different than in other parts of computer science.

1 From Last Time

2 My Motivation

3 The Challenge of Coordination

4 System Models

5 Architectures
- Software Architecture
- System Architecture

Architecture

Distributed systems are complex. We need to make sure that the components are well organized.

The architectural style is defined by how the components:

1. are connected to each other (physically and logically)
2. exchange data with each other
3. are jointly configured

Architecture (contd.)

In a distributed system we must consider both **software architecture** (how processes run) and **system architecture** how we organize the components.

Software Architecture

The three most important software architectural styles:

1. Layered Architectures
2. Service-Oriented Architectures
3. Publish-Subscribe Architectures

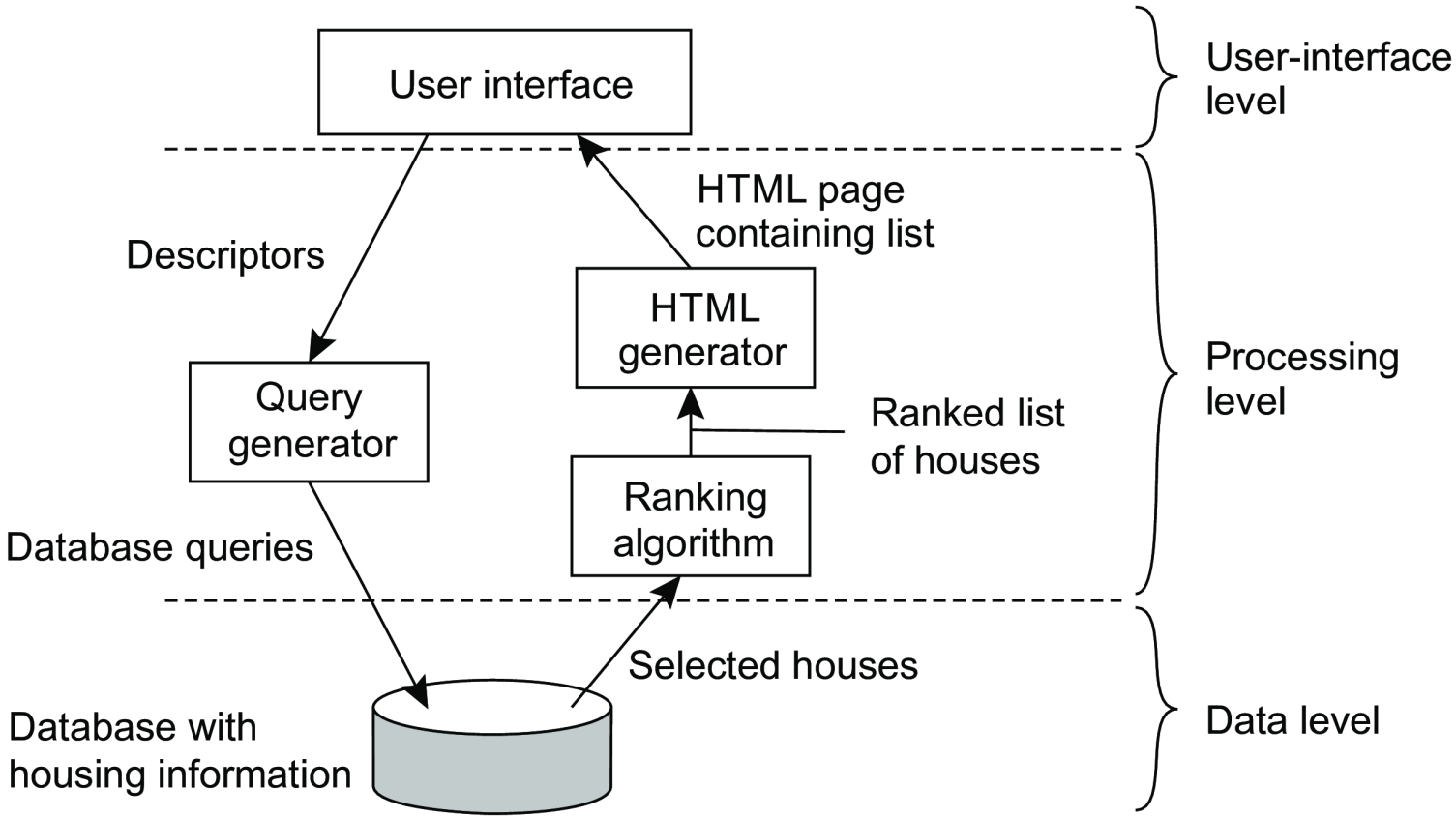
Realistically, systems use a hybrid of these styles. **For now, let's assume a client/server model.**

Layered Architecture

The layered architecture contains three well-defined levels:

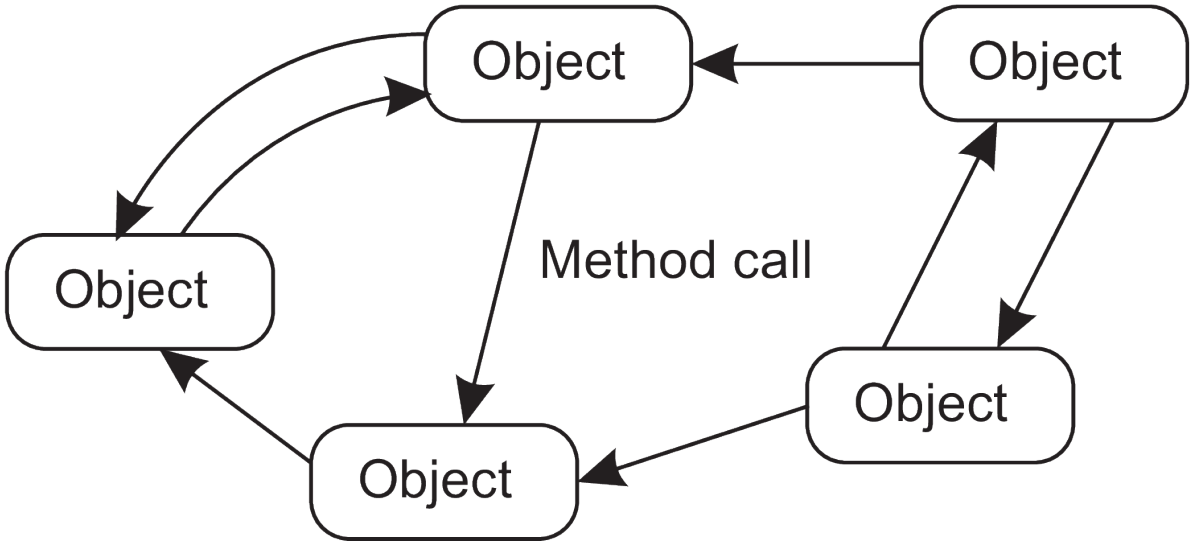
1. Application-interface
2. Processing
3. Data

Layered Architecture (contd.)



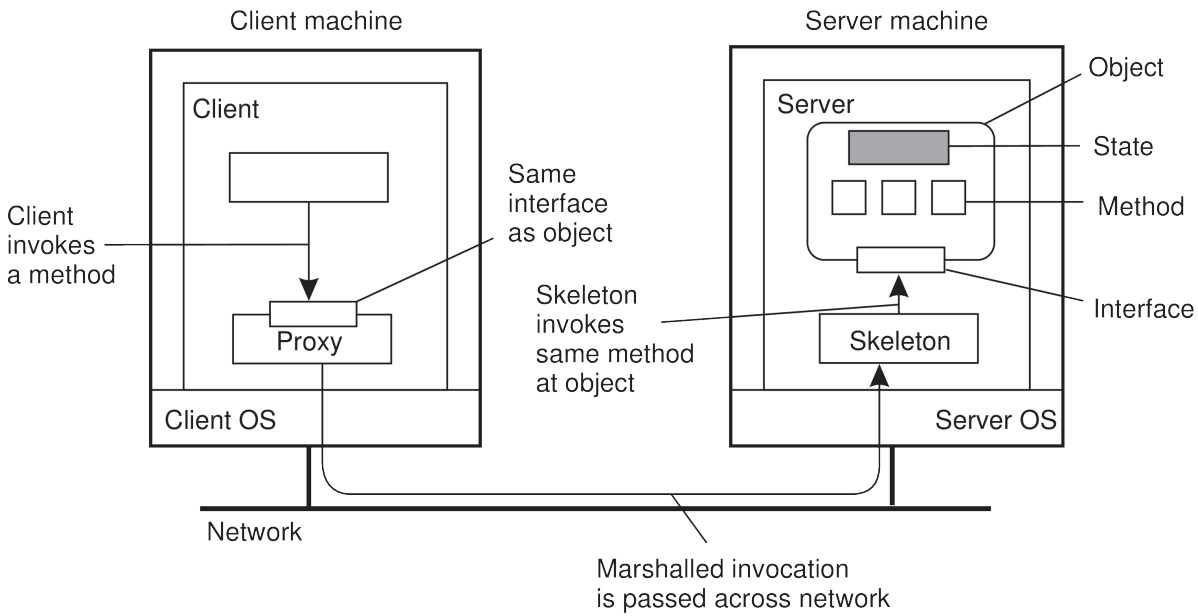
Service-Oriented Architectures

In a service-oriented architecture, there is some *object* that the client (user, or other node) needs to access and client and server can be *anywhere* on the network. State is maintained only on each node.



Service-Oriented Architectures (contd.)

In a service-oriented architecture, there is some *object* that the client (user, or other node) needs to access and client and server can be *anywhere* on the network. State is maintained only on each node.



Service-Oriented Architectures (contd.)

Encapsulation is important with services. We should separate concerns as best as we can.

Point: In a service-oriented architecture, the system is constructed as a composition of many services. These services may contain many microservices.

Service-Oriented Architectures (contd.)

The most popular communication mechanism used between client and server is Representation State Transfer (REST). RESTful communication architectures (application layer):

1. All resources are identified by a single naming scheme (URL)
2. All services offer the same primitive interface: `PUT`, `POST`, `GET`, `DELETE`. (the endpoint may have a more descriptive name)
3. Messages sent to and from the service are fully-described.
4. After execution, the component forgets everything about the caller/client.

Since state is maintained on a single component in this architecture, REST provides *state transfer*.

Publish-Subscribe Architecture (PubSub)

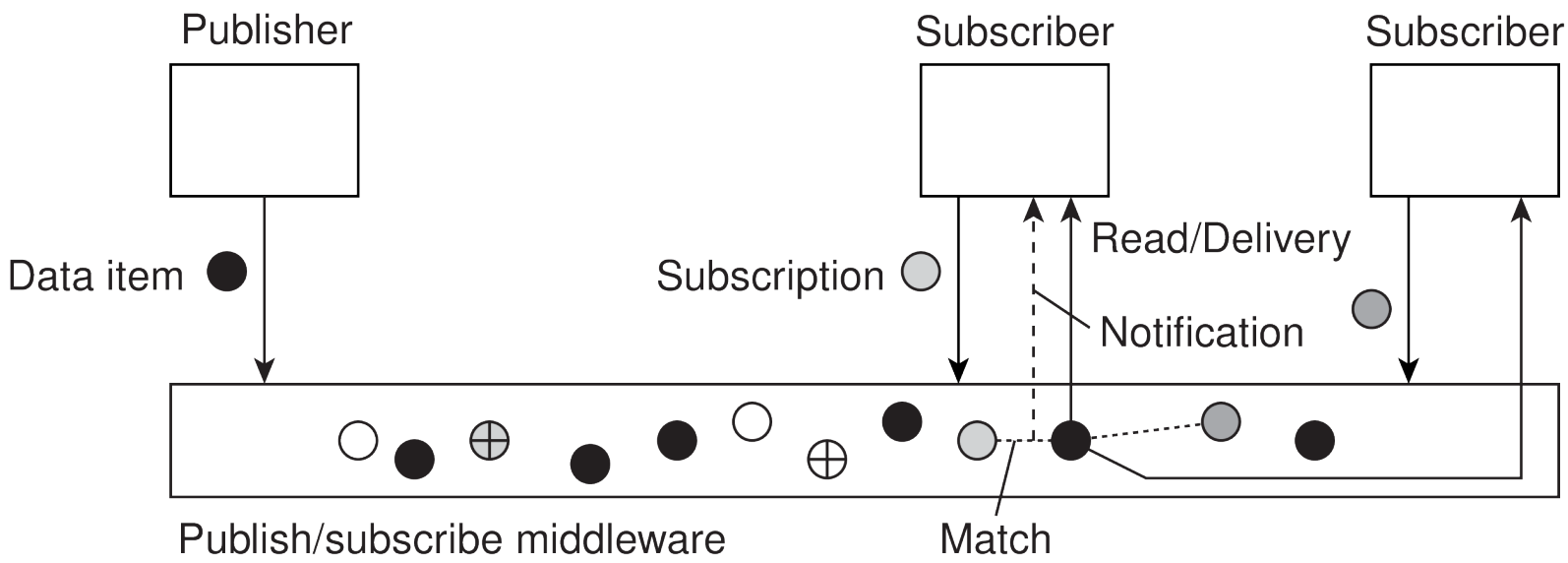
In PubSub:

- A **subscriber** can express interest in one or more *topics* of messages (e.g. direct messages vs. tweets, or literal topics), or **content-based** a series of key-value pairs that more fully describe what is of interest.
- A **publisher** introduces new messages into the system.
- A middleware layer handles the distribution of messages to the subscriber:
- 1. It sends the data directly to the subscriber (push model).
  2. It sends a notification to the subscriber (push) and the subscriber fetches (pulls) the message.

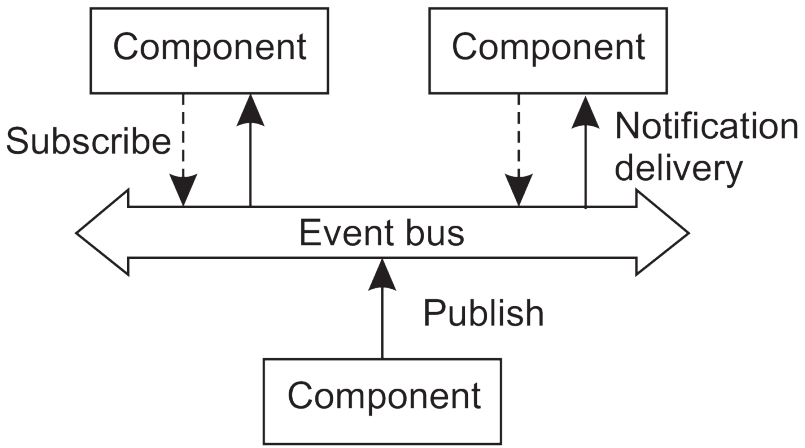
Publish-Subscribe Architecture (PubSub) (contd.)

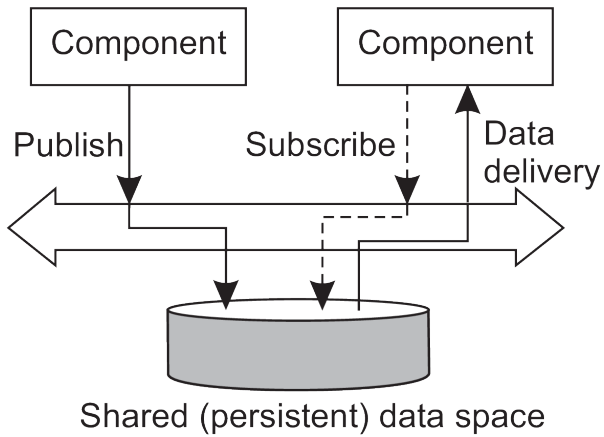
Is there a solution for the possible problem in sub-bullet 1 and 2?

Publish-Subscribe Architecture (PubSub) (contd.)



Publish-Subscribe Architecture (PubSub) (contd.)





Publish-Subscribe Architecture (PubSub) (contd.)

The two architectures on the previous slide involve the concepts of **temporal coupling** and **referential coupling**.

1. **Temporal**
   - **Coupled**: both nodes must be online to communicate.
   - **Decoupled**: one system does not need to be online.
2. **Referential Coupling**
   - **Coupled**: both systems have an explicit *name* to refer to each other.
   - **Decoupled**: identity of nodes does not matter.

Publish-Subscribe Architecture (PubSub) (contd.)

The two PubSub architectures we considered are **referentially decoupled**.

	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

What are some examples of the others?

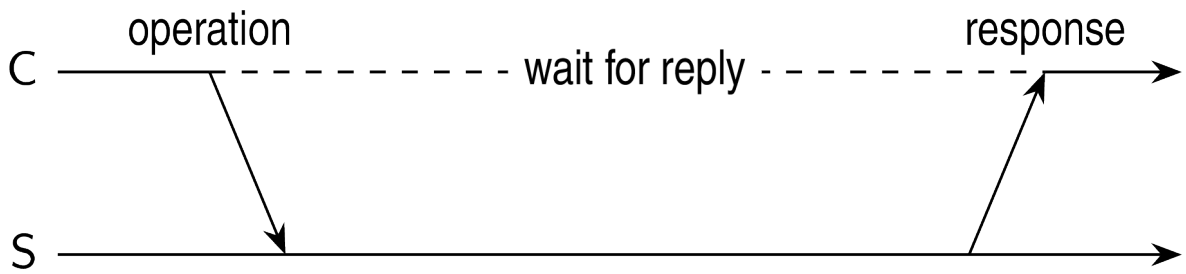
System Architecture

There are a few system architectures:

1. Layered
   - Client-server
   - Multitiered
2. Peer-to-Peer
3. Hybrid

Client-Server

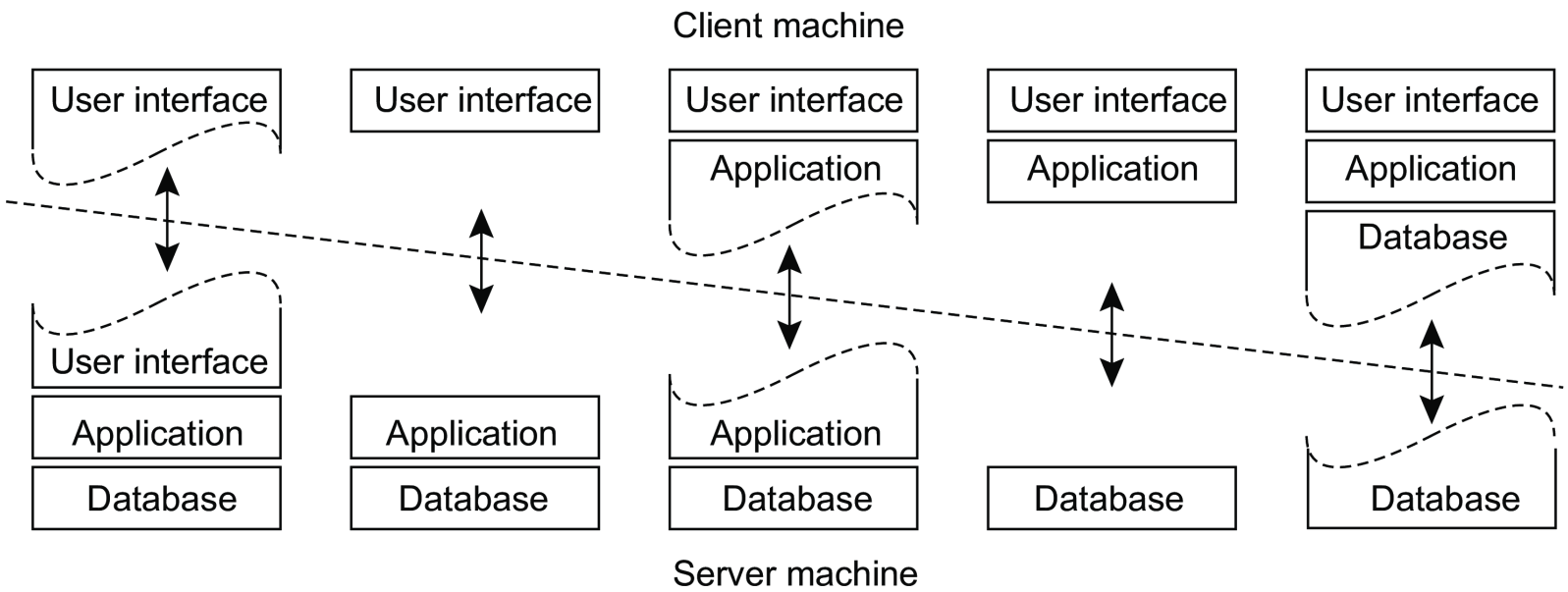
Client-Server is the most basic, and works quite well over reliable links.



Note that a node may be a server in some contexts and a client in others.

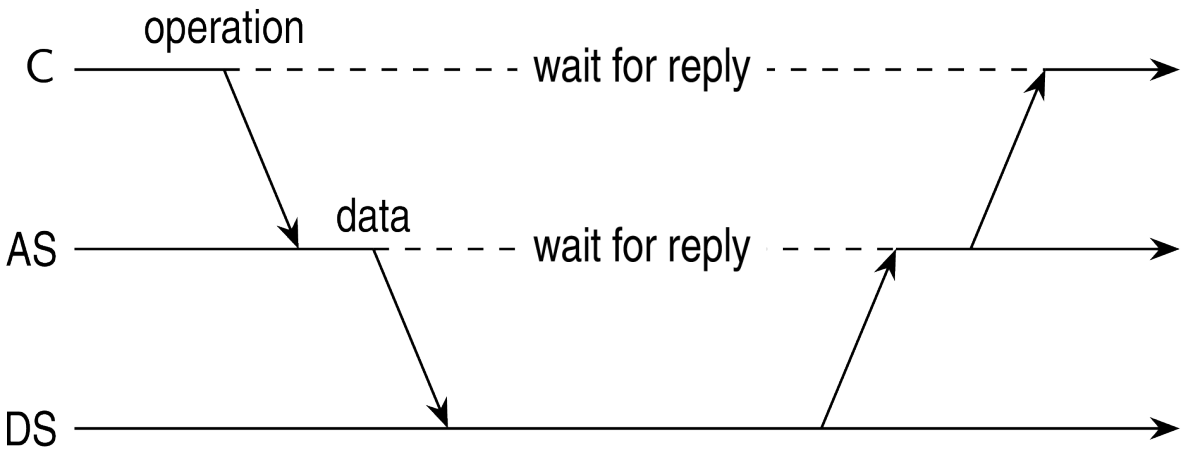
Multitiered / *n*-Tiered

In its simplest form, we have a **client** node that that performs part of the user interface. and the **server** does everything else. Two tiered:



Multitiered / *n*-Tiered (contd.)

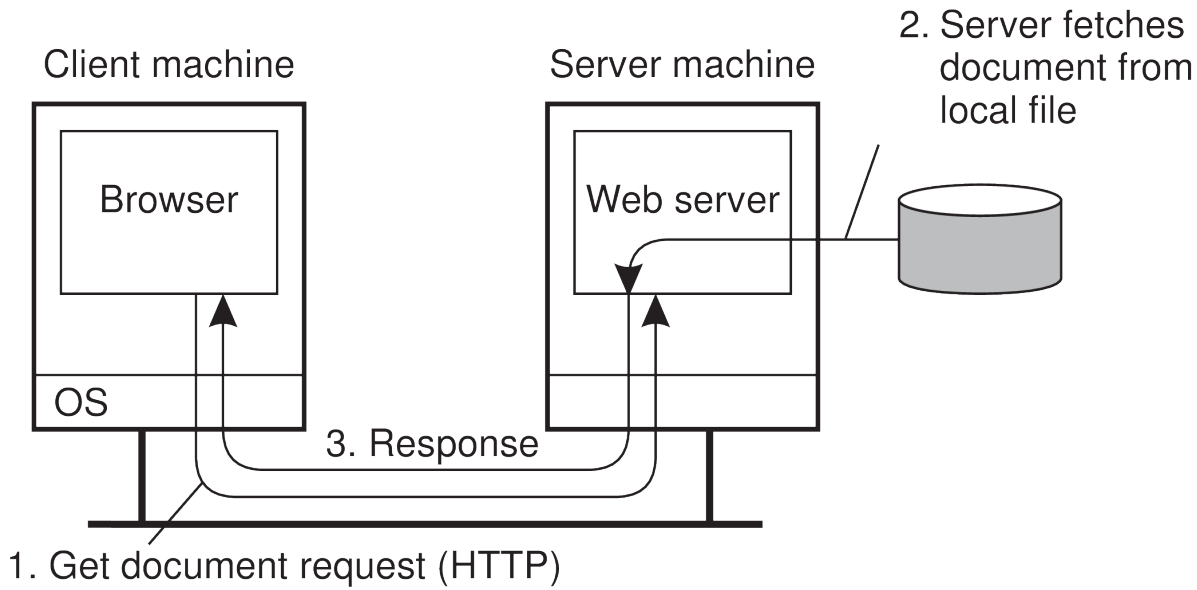
A web architecture can be client-server, two-tiered or multitiered (there is no limit to the number of tiers):



The client communicates with an application server (web server) which communicates with a database server.

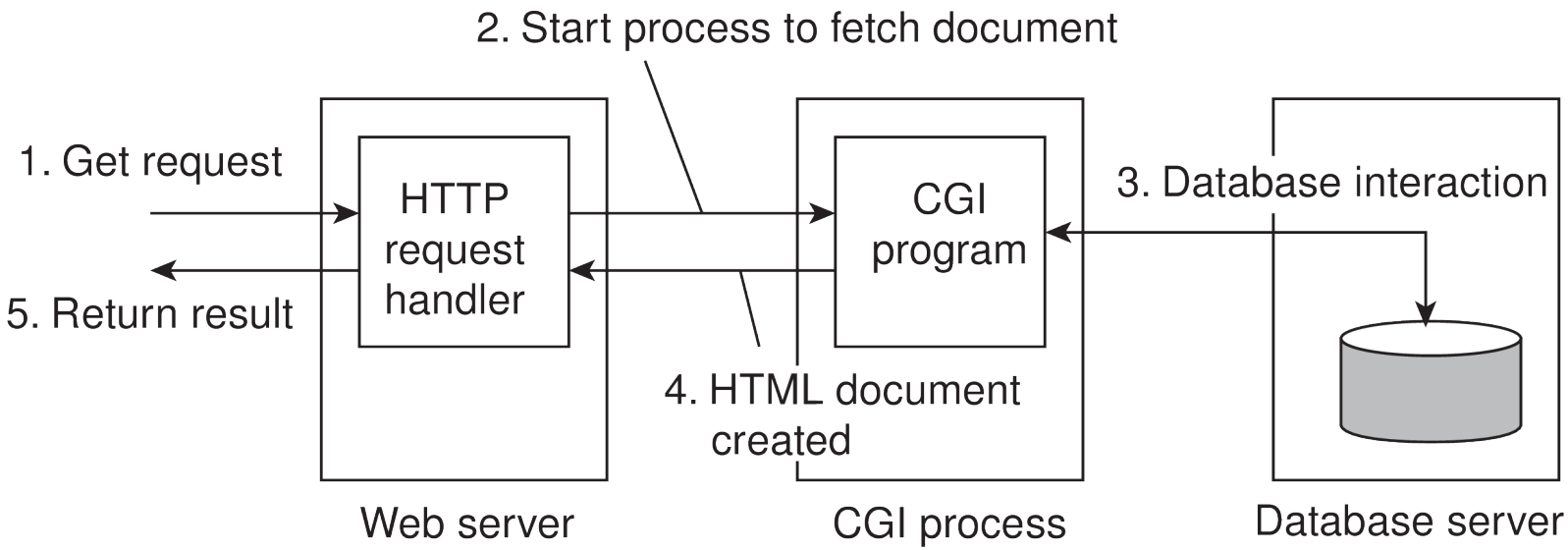
Multitiered / *n*-Tiered (contd.)

Equivalently:



Multitiered / *n*-Tiered (contd.)

Or it can be four-tiered. I think you get the idea:



Peer-to-Peer Architecture

Peer-to-Peer is very different and is *decentralized*.

In theory, every node in a P2P architecture has the same role. In practice, this is not true.

Structured Peer-to-Peer

Some P2Ps are organized into ring, binary tree and grid structures.

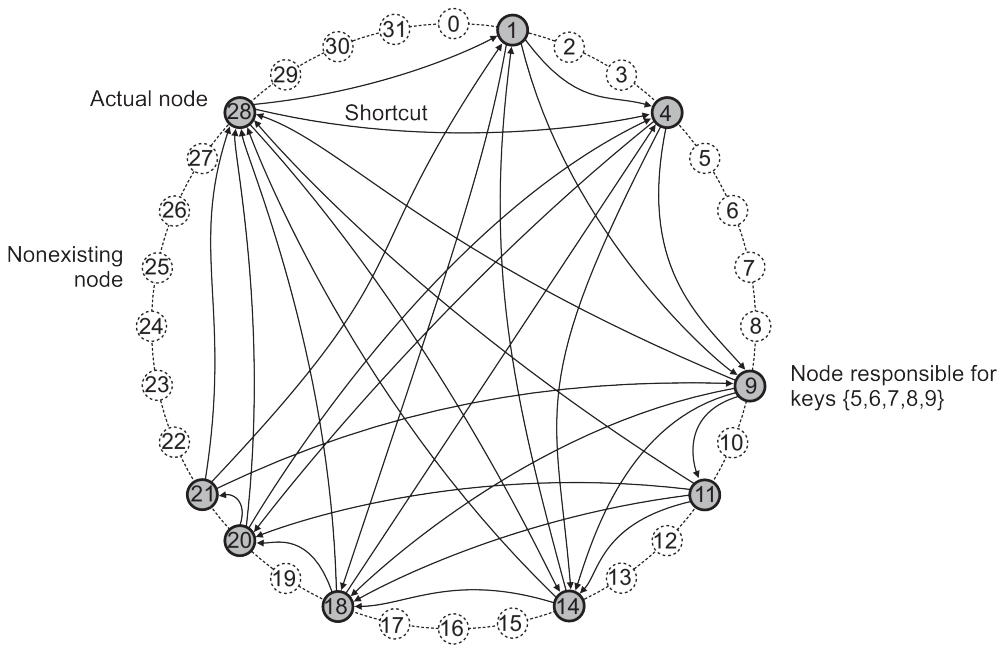


Figure: Chord system

These are used for indexing data!

Structured Peer-to-Peer (contd.)

In such a P2P system, we store key-value pairs. The **key** is just the hash of the **value**, which can be any kind of hashable data.

Each node is assigned an **identifier** that comes from the set of keys stored on that node.

This is a form of a **distributed hash table (DHT)**.

Routing lookups when nodes join and leave the network gets complicated, so we will discuss it later. Routing may not follow the ring structure, and can also follow *shortcuts*.

Note that this is kinda-sorta reminiscent of a B+ tree and/or hash indices from CS 143.

Unstructured Peer-to-Peer

You are probably more familiar with unstructured P2P systems.

Each node maintains a dynamic list of neighbors. This creates what is called a **random graph**, each edge in the network exists with a probability $P(u, v)$ where u and v are nodes.

Random graphs are a fascinating research area in mathematics, statistics/probability and computer science.

Unstructured Peer-to-Peer (contd.)

When a node **joins** the system, it contacts a **well known node** to get an *initial* list of peers (neighbors).

This list is then used to find other peers, or it can remove peers that are non-responsive. This process is continuous.

Unstructured Peer-to-Peer (contd.)

Retrieving data is not as straightforward though. We have to *search* for it – possibly brute force. Two extremes:

1. **Flooding**. The query is sent from a node to all of its neighbors.
   - If it does not contain the data, it forwards the message/query to its neighbors.
   - If the neighbor contains the data, it either responds directly to the client, or it can pass it back to the *forwarder*.
   - If the neighbor has already seen the request (cycle), it can ignore it.
   - Very expensive. Can use a TTL in hops.

Unstructured Peer-to-Peer (contd.)

2. **Random Walks**. The query is sent to a *random* neighbor at each step.
   - Hits are processed the same as in the flood.
   - Misses require the node passing the message/query to a random neighbor and so on.
   - Less network traffic, but takes longer.
   - One solution: Start n random walks in parallel. Decreases time by a factor of n. Can also add a TTL in hops.

The concept of random walks is extremely important in statistical computing and machine learning. Random walks form the backbone of the Metropolis-Hastings algorithm used in Markov Chain Monte Carlo (MCMC), used for drawing samples from complex probability distributions. (For more info, take Stats 102C)

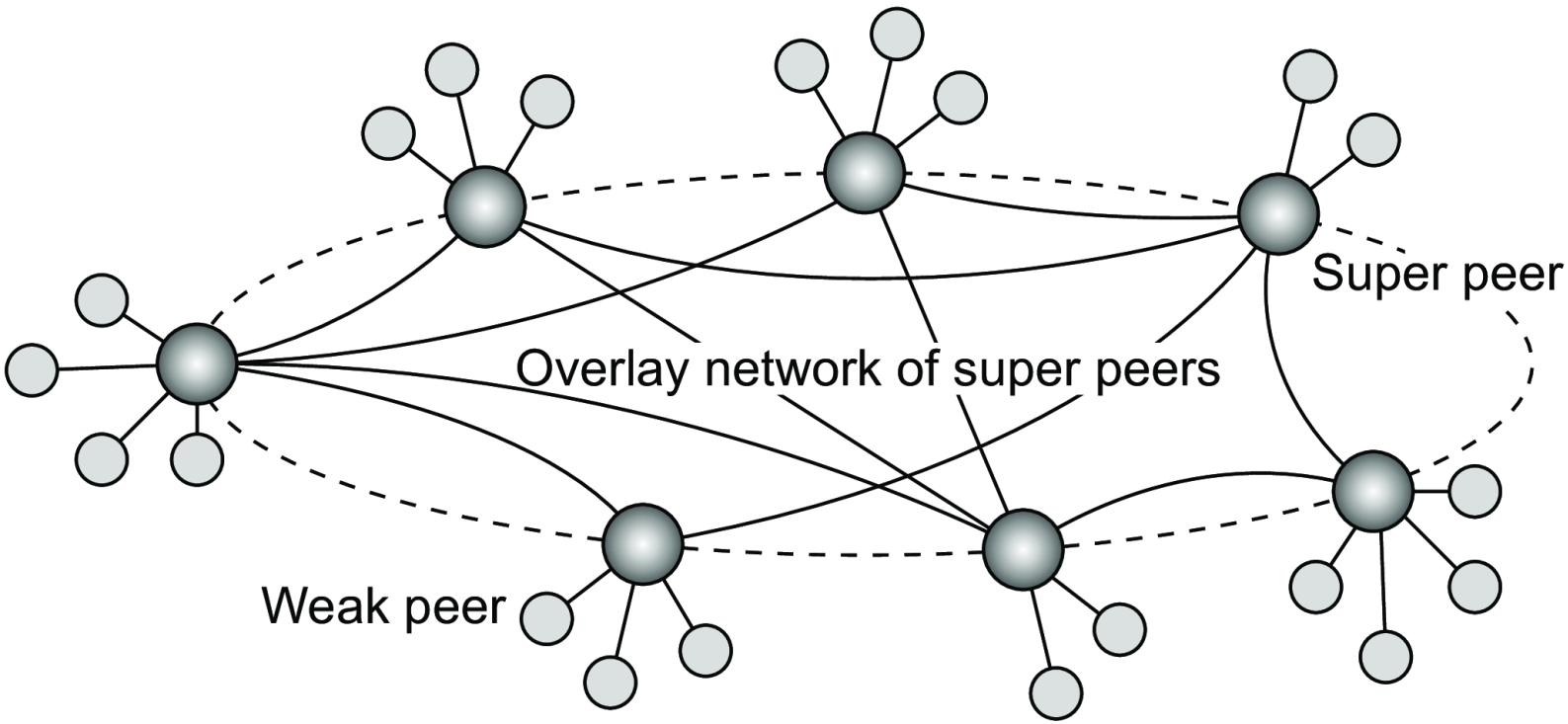
Unstructured Peer-to-Peer (contd.)

In Cassandra, it is said that all nodes play the same role. There are no masters, or special nodes. Other P2P systems include a hierarchy of nodes.

"Special" nodes may be used to store an index of data. A client can issue a query to an overlay network of special nodes to localize a search. Then, use flood or random walk. **Note that this index is dynamic though, a complication we will discuss later.**

These special nodes are called `super peers` and the others are **weak peers**.

Unstructured Peer-to-Peer (contd.)



Unstructured Peer-to-Peer (contd.)

But... what happens when a super peer becomes unavailable?

We may need to replace it. How? Which node? That involves **leader election** which we will cover later in the course.

This Friday

Tasks for you:

- Please try to find a partner by EOD Friday so we can start creating your repos.
- Discussion section will go over the dev environment setup for this class as well as introductory Go.
- There will be some Go exercises.