Tema 3. Diseño de Software Crítico

Replicación en Sistemas Distribuidos

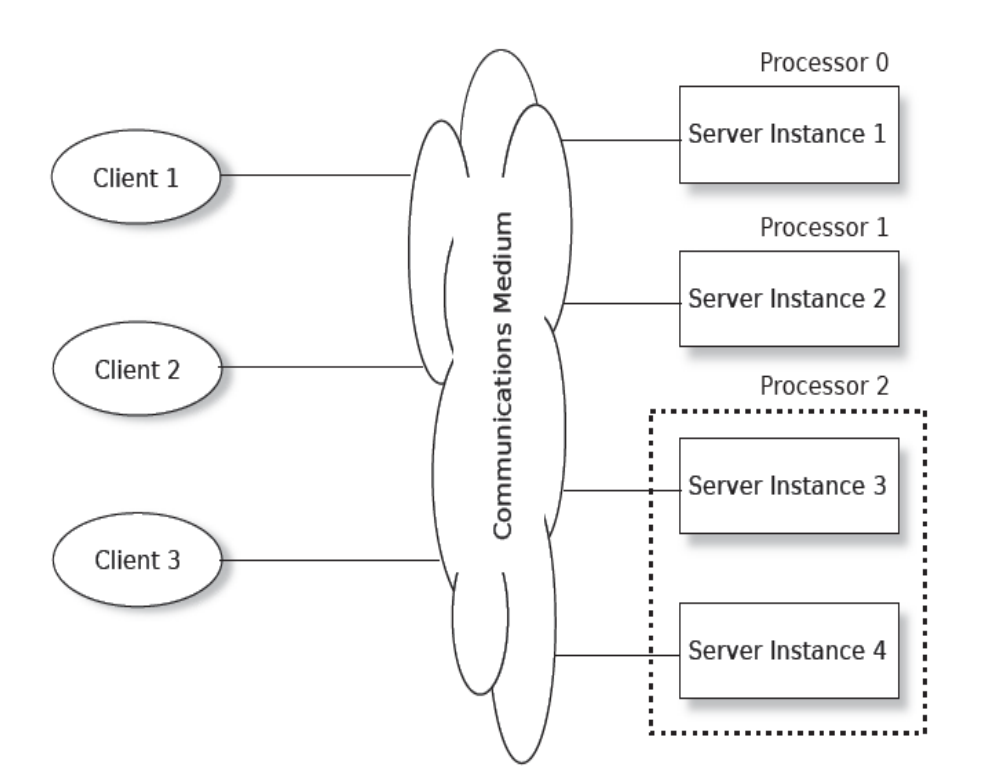
Manuel Díaz mdiaz@uma.es

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Replication in Distributed Systems

- Introduction to group services
  - Multicast: Time and Ordering
  - Failure detection
- Time and ordering
  - Implementing message ordering: FIFO, Causal, Total
  - Multicast
  - Consensus
  - Virtual Synchrony
- Failure Detection in distributed systems
  - Heartbeat algorithms
  - Gossiping

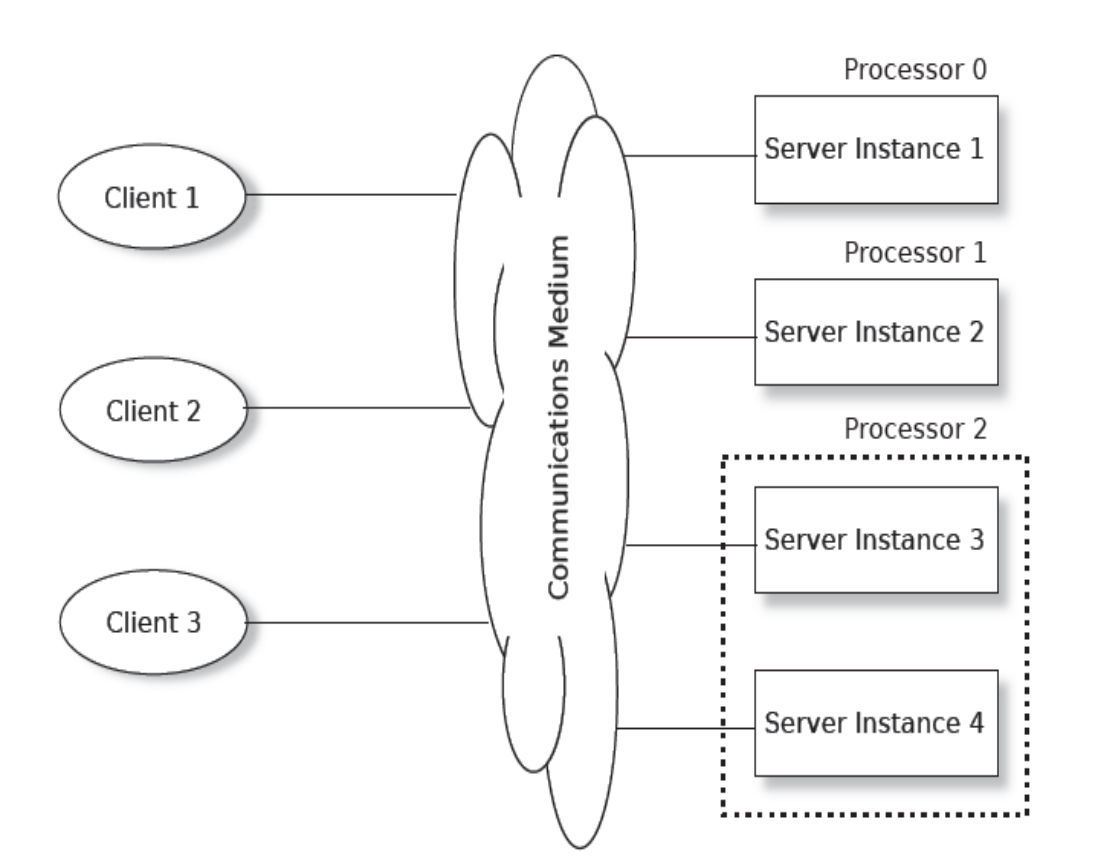
Active Replication



The underlying idea is:

- The servers all start in the same state (and the updating of a new group member's state is also part of the protocol)
- Each server receives the same messages in exactly the same order
- All the servers must eventually arrive at the same state

Active Replication



For some systems, the requirement for common arrival order of all events at each server in the group can be relaxed

Sender order: All messages from each client are delivered to each group member in the order they were sent. There is no guarantee that messages from two different clients will be delivered in the same order to the group

Causal order: If a message s1 causes a program to send message s2 (i.e., s1 causes s2), then s1 will be delivered before s2 to all group members

Total or agreed order: If any group member receives s1 before s2, then all group members will

Active Replication

- The communication mechanism between the clients and the servers in the figure is indicated by a cloud.
- In principle, this can be any medium supporting messages (WAN, LAN a backplane in a rack,…)
  - The closer the connection, the better active replication works
- There are several technical solutions and implementations of this approach in the form of middleware or virtualization platforms

State Machine Replication

- Fundamental approach to fault-tolerance
  - Google Spanner
  - Apache Zookeeper
  - Windows Azure Storage
  - MySQL Group Replication
  - Galera Cluster,…
- Simple execution model
  - Repliclas order all commands
  - Replicas execute commands deterministically and in the same order

But, What do we replicate?
Single big service performs poorly… why?

- Until 2005 “one server” was able to scale and keep up, like for Amazon’s shopping cart. A 2005 server often ran on a small cluster with, perhaps, 2-16 machines in the cluster.

- This worked well.

- But suddenly, as the cloud grew, this form of scaling broke. Companies threw unlimited money at the issue but critical services like databases still became hopelessly overloaded and crashed or fell far behind.

Partially based on matrial from Ken Birman – University of Cornell

The dangers of replication

- At Microsoft, Jim Gray anticipated this as early as 1996.
- He and colleagues wrote a foundational paper from their insights:

The dangers of replication and a solution. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. In Proceedings of the 1996 ACM SIGMOD Conference. pp 173-182. DOI=http://dx.doi.org/10.1145/233269.233330

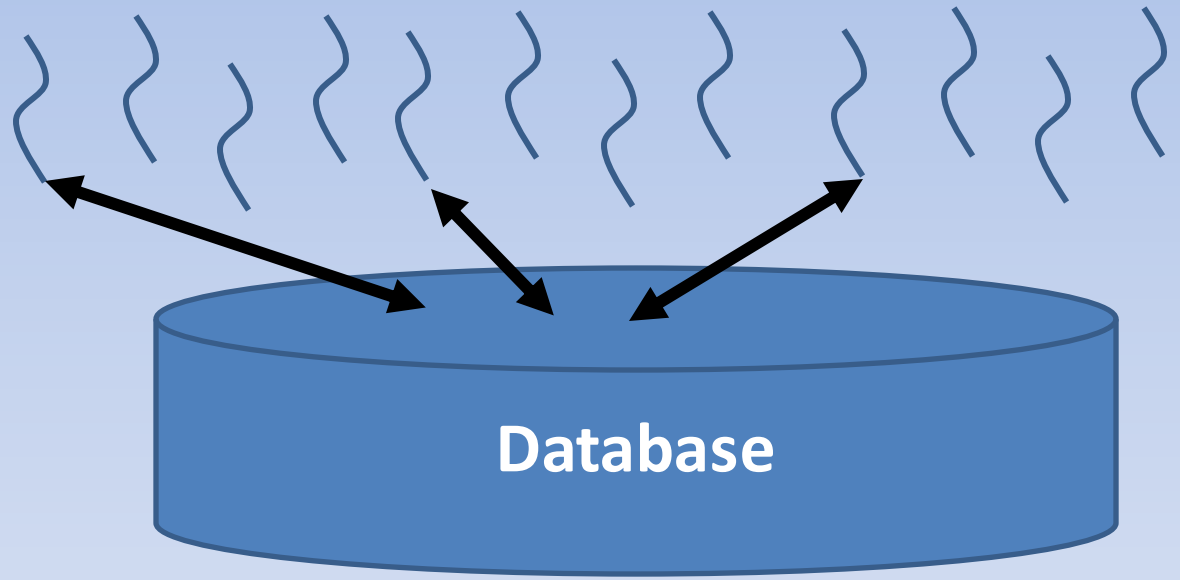
Basic message: divide and conquer is really the *only* option.

Partially based on matrial from Ken Birman – University of Cornell

The core issue

- The paper assumes that your goal is some form of lock-based consistency, which they model as database serializability or “state machine replication”, as in our case
- So applications will be using read locks, and write locks, and because we want to accommodate more and more load by adding more servers, the work spreads over the pool of servers.
- This is a very common way to think about servers of all kinds.

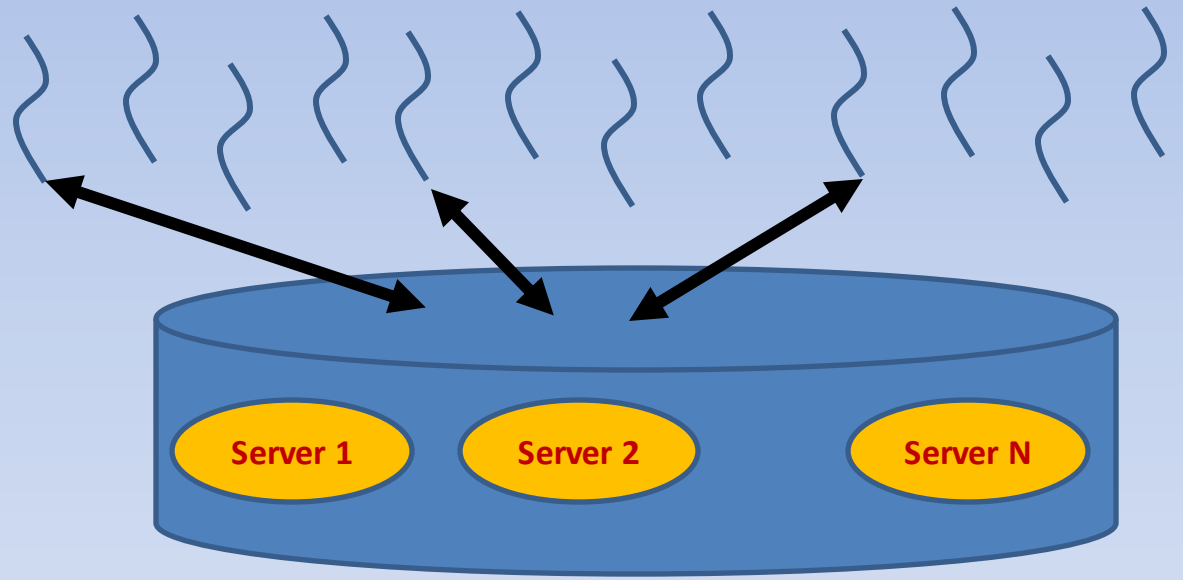
Their setup, as a picture



Applications using the database: *client processes*

During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.

Their Basic setup, as a picture



Applications using the database: *client processes*

During the run, T concurrent requests are issued. Here, 3 are running right now, but T could be much larger.

What should be the goals?

- A scalable system needs to be able to handle "more T's" by adding to N
- Instead, they found that the work the servers must do will increase as T^5

  Worse, with an even split of work, deadlocks occur as $N^{3,}$ causing feedback (because the reissued transactions get done more than once).

- Example: if 3 servers (N=3) could do 1000 TPS, with 5 servers the rate might drop to 300 TPS, purely because of deadlocks forcing abort/retry.

Why do services slow down at scale?

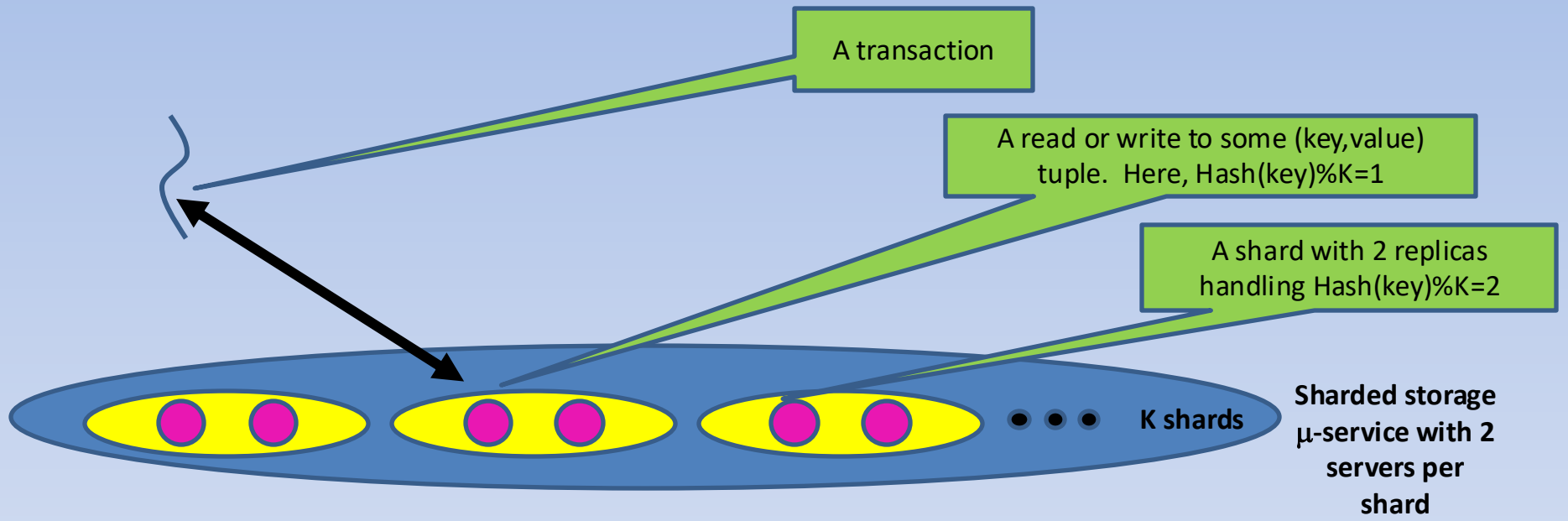
- The paper pointed to several main issues:
  - **Lock contention**. The more concurrent tasks, the more likely that they will try to access the same object (birthday paradox!) and wait for locks.
  - **Abort**. Many consistency mechanisms have some form of optimistic behavior built in. Now and then, they must back out and retry.
  - **Deadlock** also causes abort/retry sequences.

- The paper actually explores multiple options for structuring the data, but ends up with similar negative conclusions *except in one specific situation.*

What was that one good option?

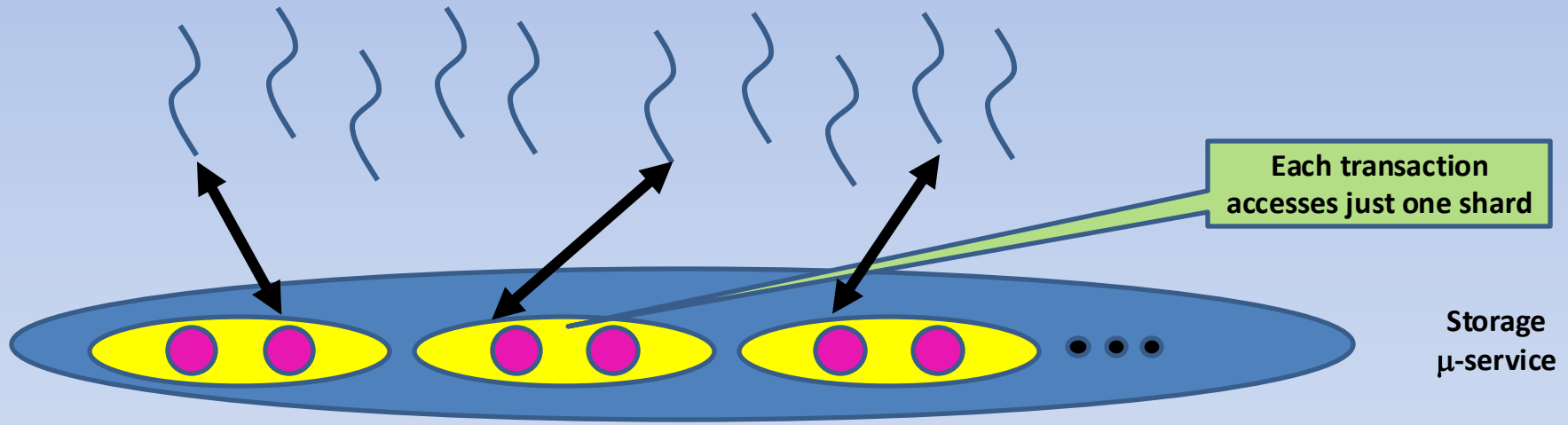
- Back in 1996, they concluded that you are forced to shard the database into a large set of much smaller databases, with distinct data in each.
- The later showed that for every problem he ran into, it was possible to devise a sharded solution in which transactions only touched a single shard at a time.
- In 1996, it wasn't clear that every important service could be sharded. By the 2007 it was clear that in fact, this is feasible!

Sharding: This works… but carefully



We will often see this kind of picture. Cloud systems make very heavy use of key-based sharding. A (key,value) store holds data in the shards.

Sharding: This works… but carefully



If a transaction does all its work at just one shard, never needing to accesst wo or more, we can use *state machine replication* to do the work.

No locks or 2-phase commit are required. This scales very well.

CAP Theorem

- Conjectured by Prof. Eric Brewer at PODC (Principle of Distributed Computing) 2000 keynote talk
- Described the *trade-offs involved in distributed system*
- It is impossible for a web service to provide following *three guarantees at the same time*

Availability

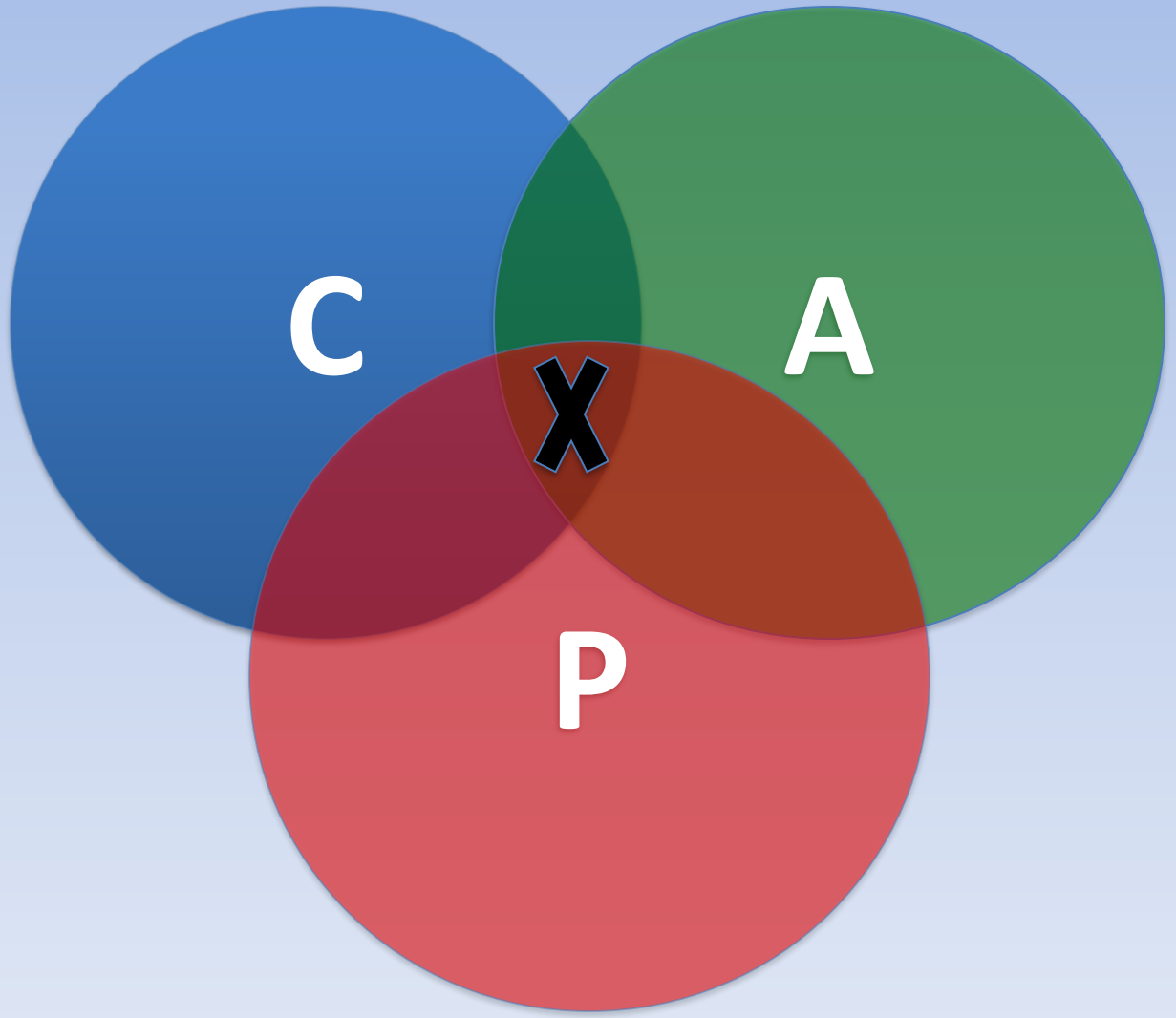
Consistency

Partition-tolerance

CAP Theorem

- Consistency:
  - All nodes should see the same data at the same time
- Availability:
  - Node failures do not prevent survivors from continuing to operate
- Partition-tolerance:
  - The system continues to operate despite network partitions
- A distributed system can satisfy any two of these guarantees at the same time **but not all three**

CAP Theorem



CAP Theorem: Proof

2002: Proven by research conducted by Nancy Lynch and Seth Gilbert at MIT

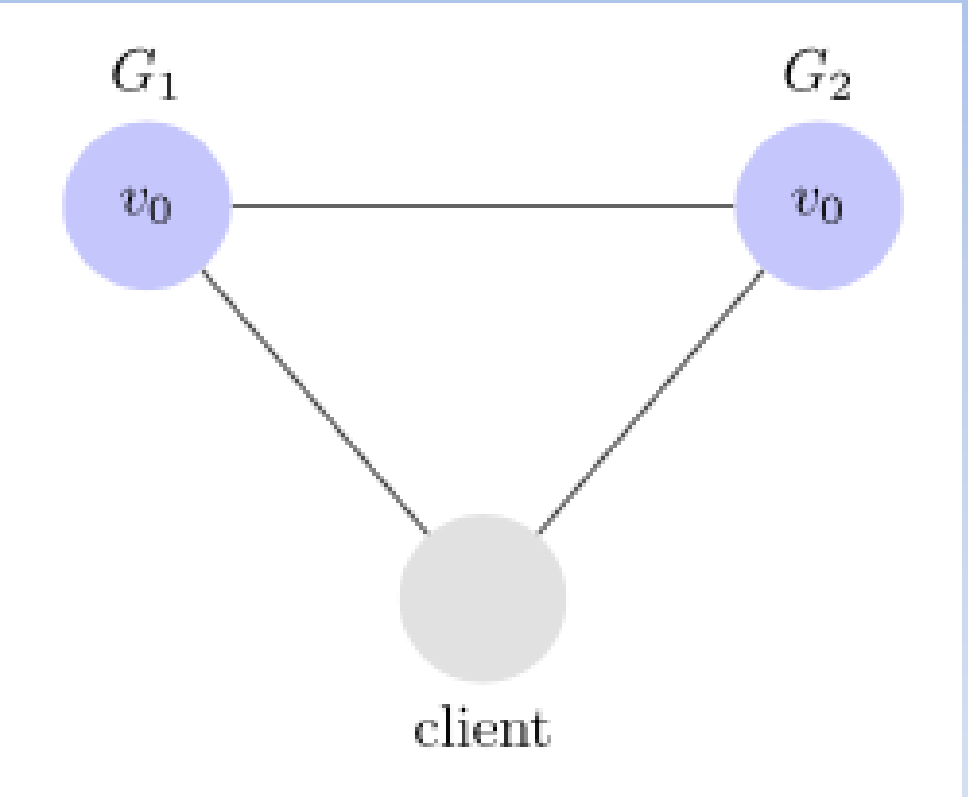
Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

CAP Theorem

Let's consider a simple system is composed of two servers, G1 and G2.

Both servers are keeping track of the same variable, v, whose value is initially v0

G1 and G2 can communicate with each other and can also communicate with external clients.

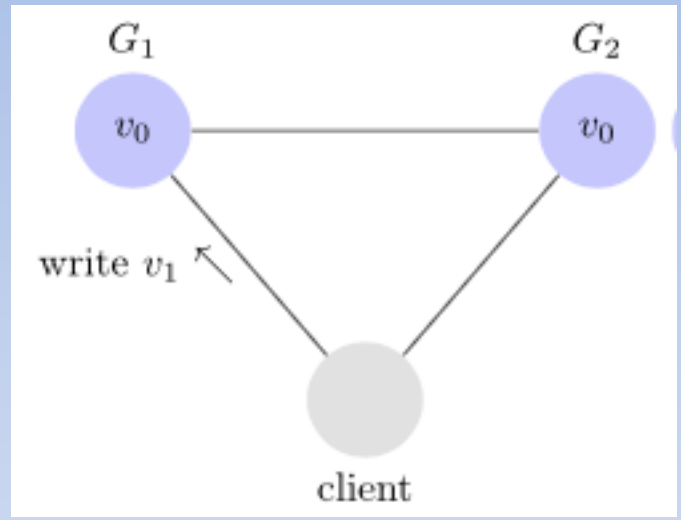


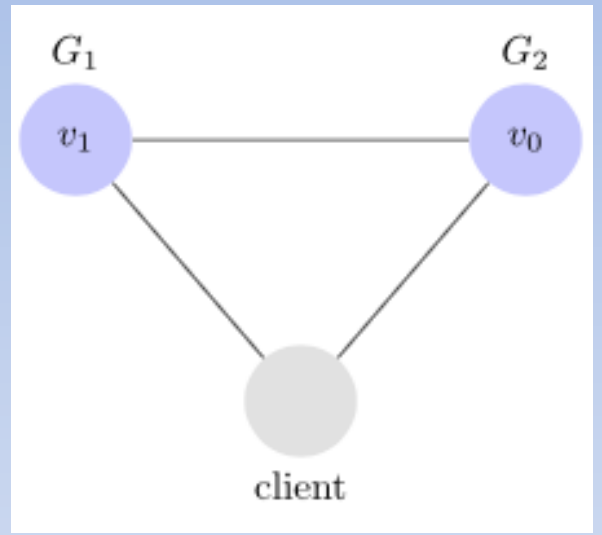
CAP Theorem

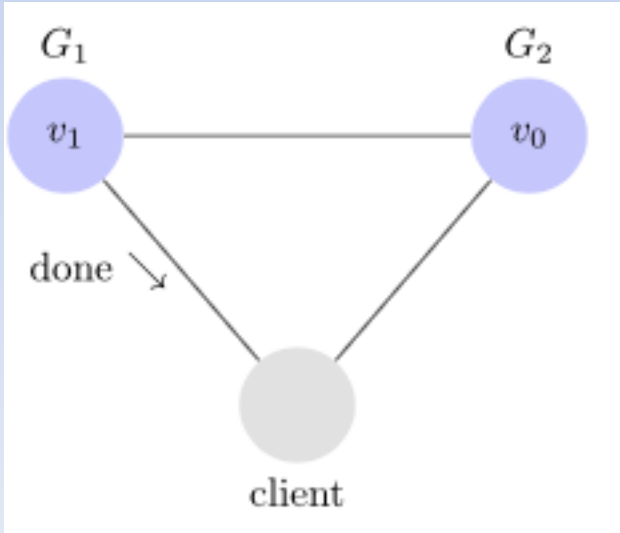
A client can request to write and read from any server

When a server receives a request, it performs any computations it wants and then responds to the client

For example, here is what a write looks like.

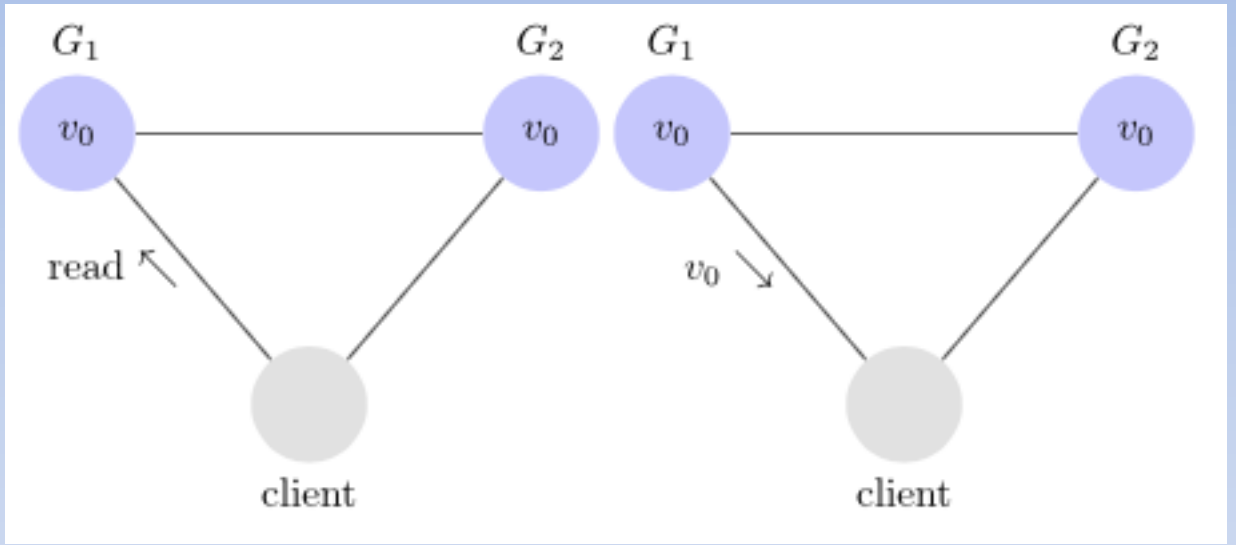






CAP Theorem

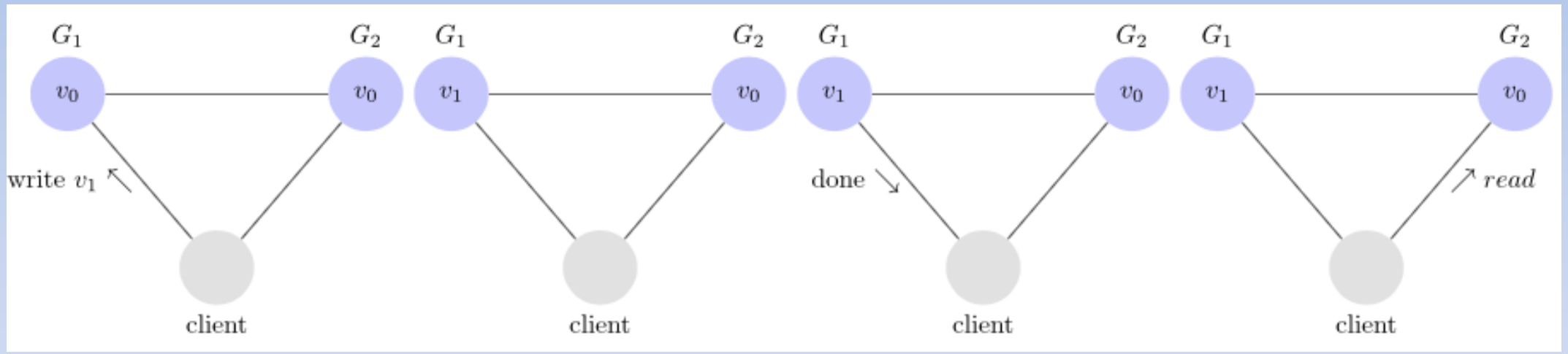
Here is what a write looks like.

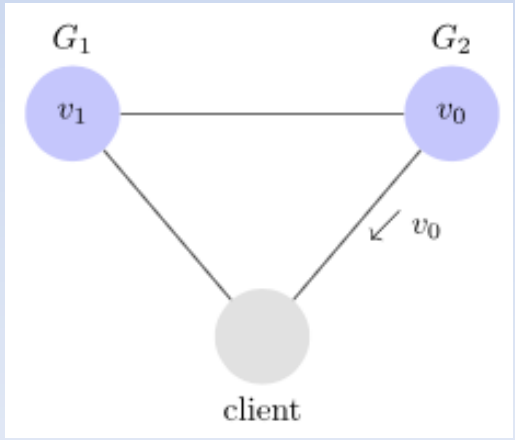


CAP Theorem

Consistency

Any read operation that begins after a write operation completes must return that value, or the result of a later write operation





Inconsistency example

CAP Theorem

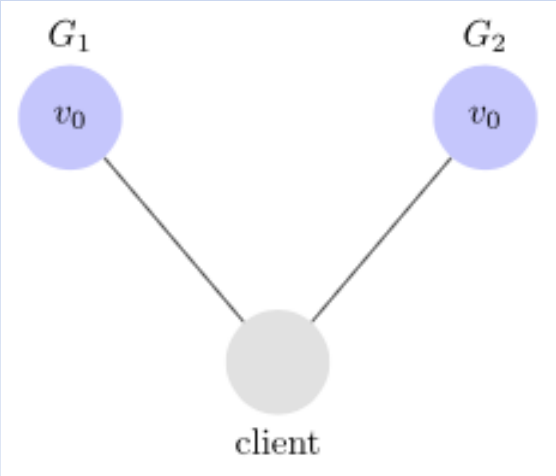
Availability

Every request received by a non-failing node in the system must result in a response

If our client sends a request to a server and the server has not crashed, then the server must eventually respond to the client. The server is not allowed to ignore the client's requests

Partition Tolerance

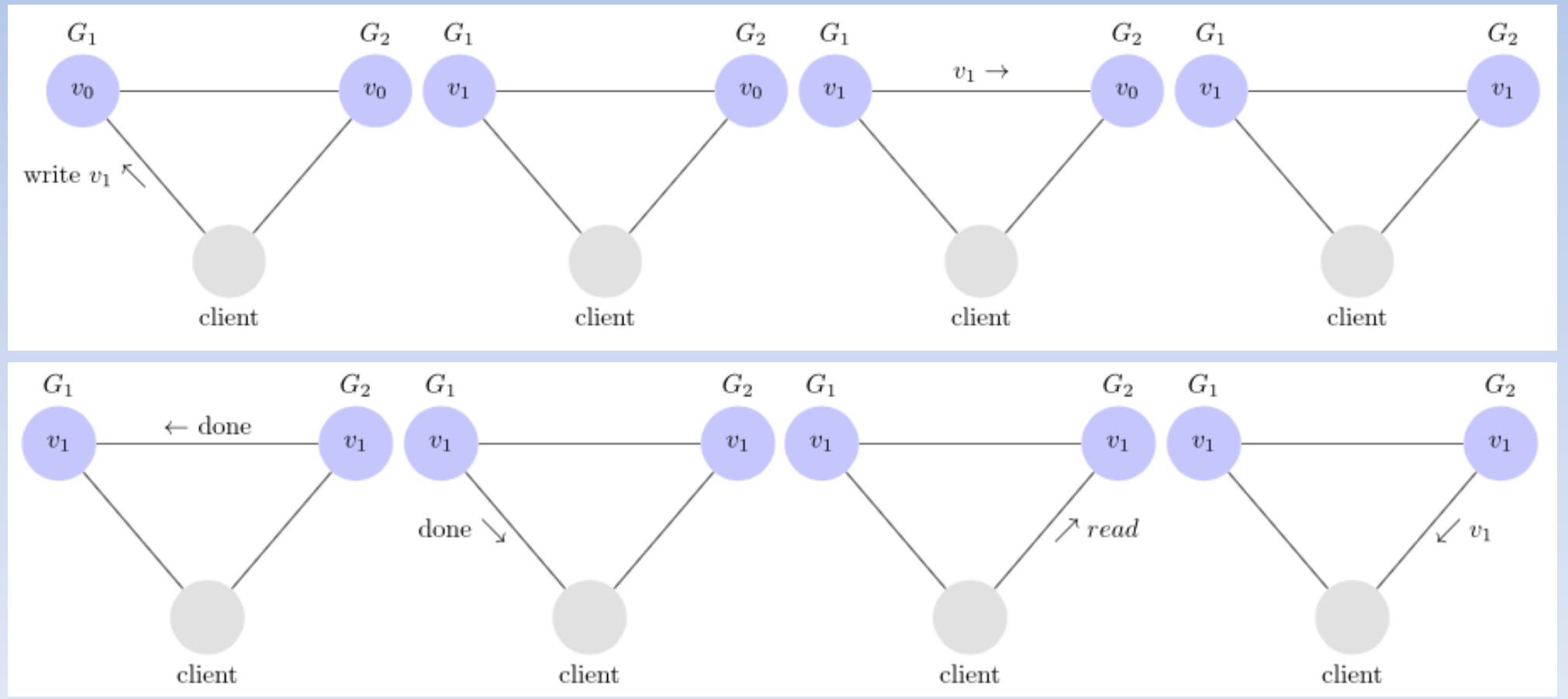
The network will be allowed to lose arbitrarily many messages sent from one node to another. This means that any messages G1 and G2 send to one another can be dropped.



CAP Theorem

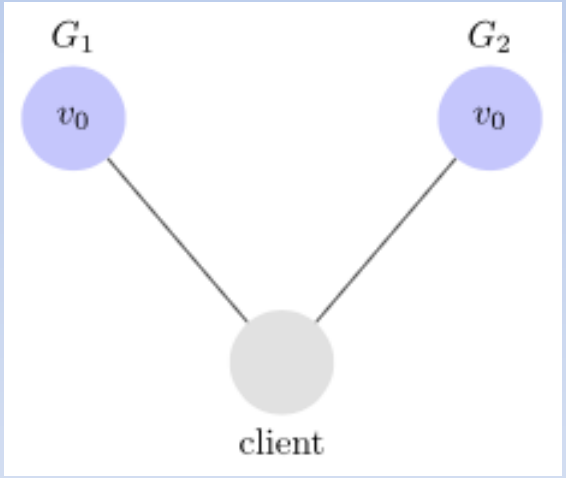
Consistency

In this system, G1 replicates its value to G2 before sending an acknowledgement to the client. Thus, when the client reads from G2, it gets the most up to date value of v: v1.



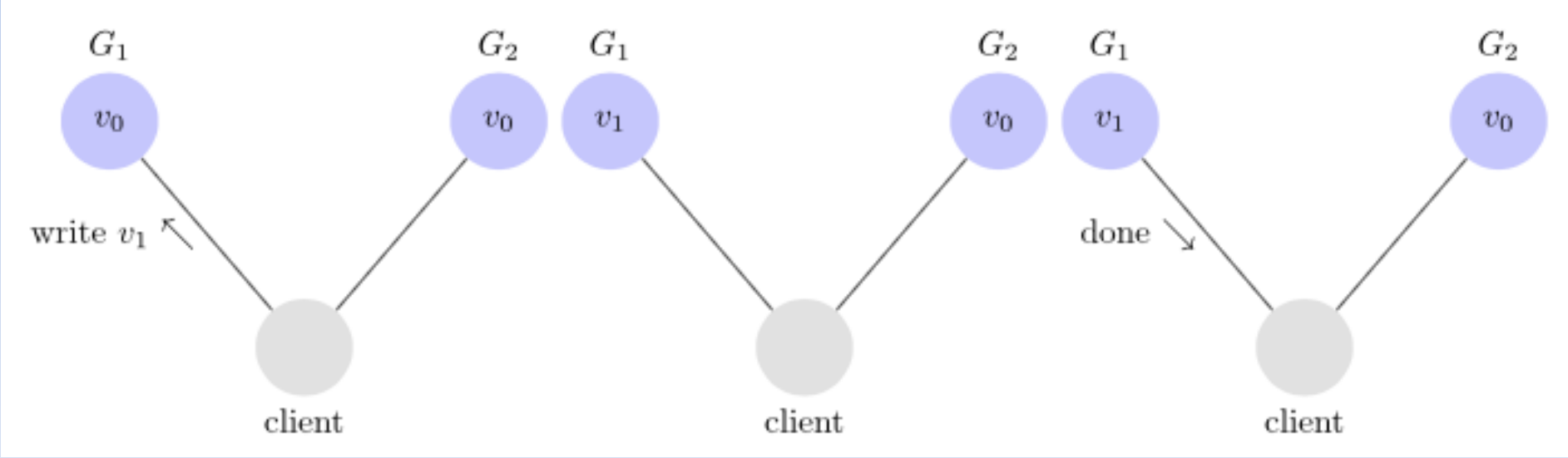
CAP Theorem: Proof

Assume for contradiction that there does exist a system that is consistent, available, and partition tolerant. The first thing we do is partition our system. It looks like this.



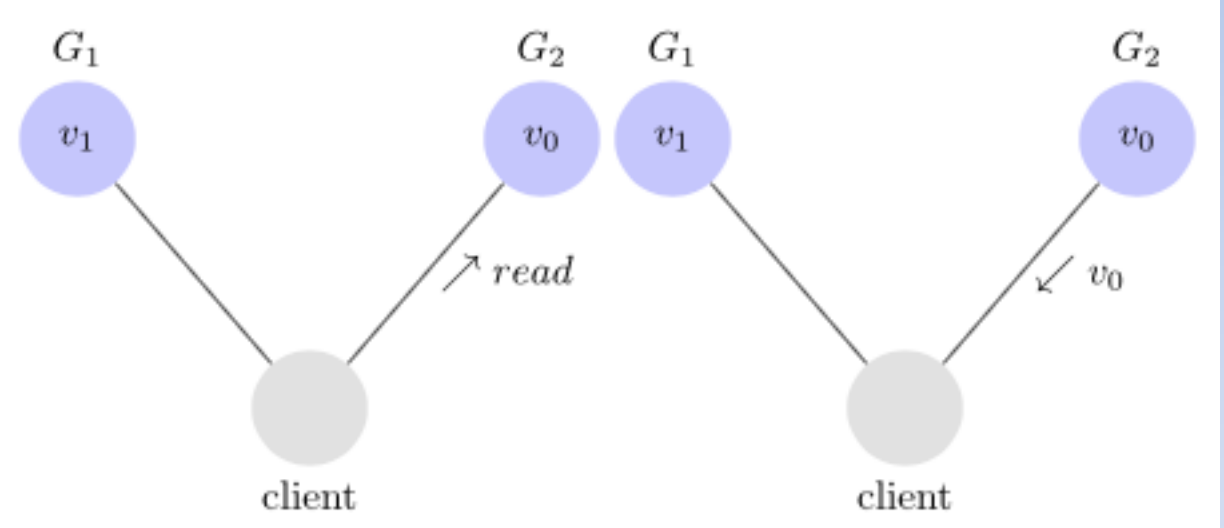
CAP Theorem: Proof

Next, we have our client request that v1 be written to G1. Since our system is available, G1 must respond. Since the network is partitioned, however, G1 cannot replicate its data to G2



CAP Theorem: Proof

Next, we have our client issue a read request to G2. Again, since our system is available, G2must respond. And since the network is partitioned, G2 cannot update its value from G1. It returns v0



G2 returns v0 to our client after the client had already written v1 to G1. This is inconsistent. This is a contradiction, so nop such system exists

Why this is important?

- The future of computer systems is **distributed** (IoT, Big Data Trend, etc.)
- CAP theorem describes the **trade-offs** involved in distributed systems
- A proper understanding of CAP theorem is essential to **making decisions** about the future of distributed system **design**
- Misunderstanding can lead to **erroneous or inappropriate** design choices

Problem for Relational Database to Scale

- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which unfortunately is **impossible** …

Revisit CAP Theorem

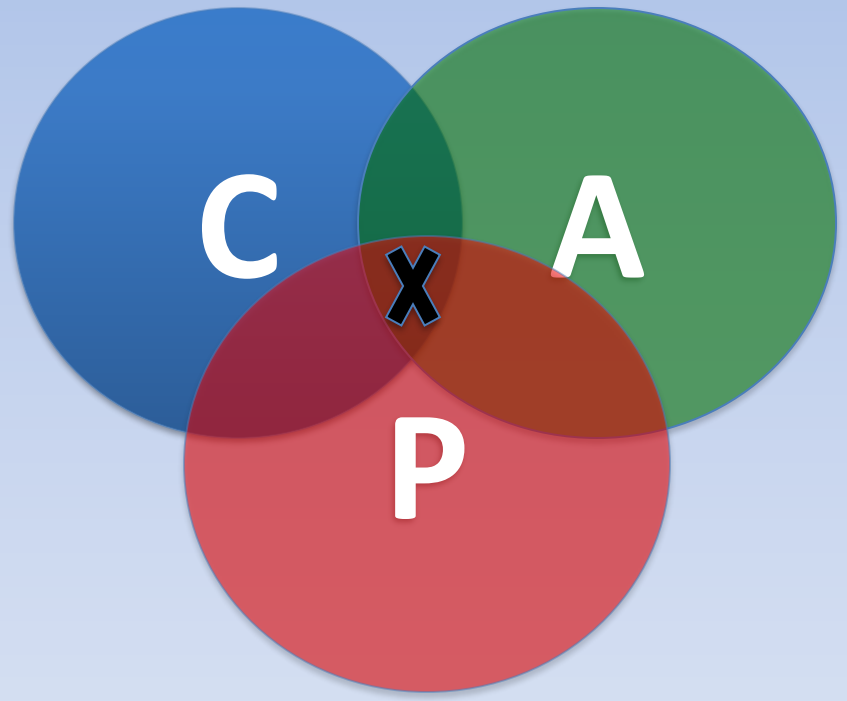
Of the following three guarantees potentially offered a by distributed systems:

- Consistency
- Availability
- Partition tolerance

Pick two

This suggests there are three kinds of distributed systems:

- CP
- AP
- CA

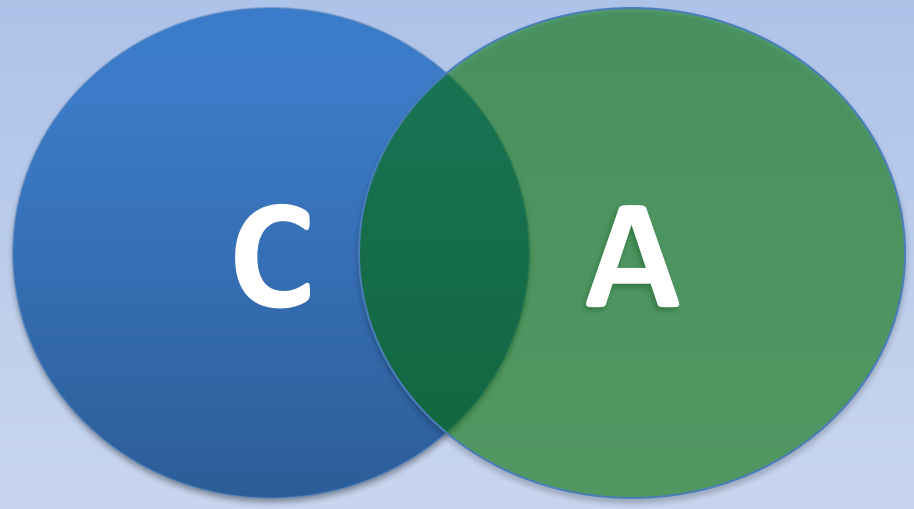


Any problems?

A popular misconception: 2 out 3

How about CA?

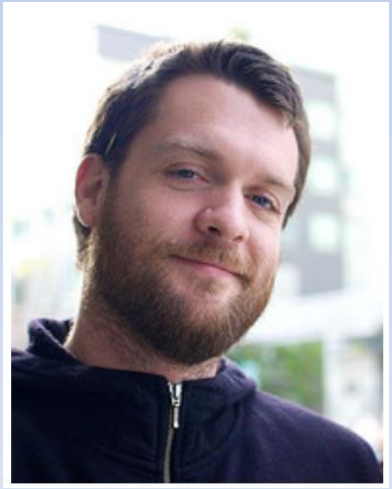
Can a distributed system (with unreliable network) really be not tolerant of partitions?



A few witnesses

Coda Hale, Yammer software engineer:

> "Of the CAP theorem's Consistency, Availability, and Partition Tolerance, **Partition Tolerance is mandatory in distributed systems**. You cannot not choose it."



http://codahale.com/you-cant-sacrifice-partition-tolerance/

A few witnesses

Werner Vogels, Amazon CTO

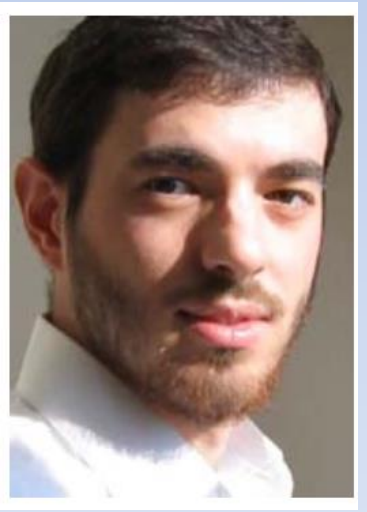
> "An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time**."

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

A few witnesses

Daneil Abadi, Co-founder of Hadapt

> So in reality, there are only two types of systems ... I.e., if there is a partition, **does the system give up availability or consistency?**



http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

CAP Theorem Reformulation

Prof. Eric Brewer in 2012: father of CAP theorem

> "The "2 of 3" formulation was always **misleading** because it tended to oversimplify the tensions among properties. ...
>
> **CAP prohibits only a tiny part of the design space**: *perfect availability and consistency in the presence of partitions*, which are rare."

http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

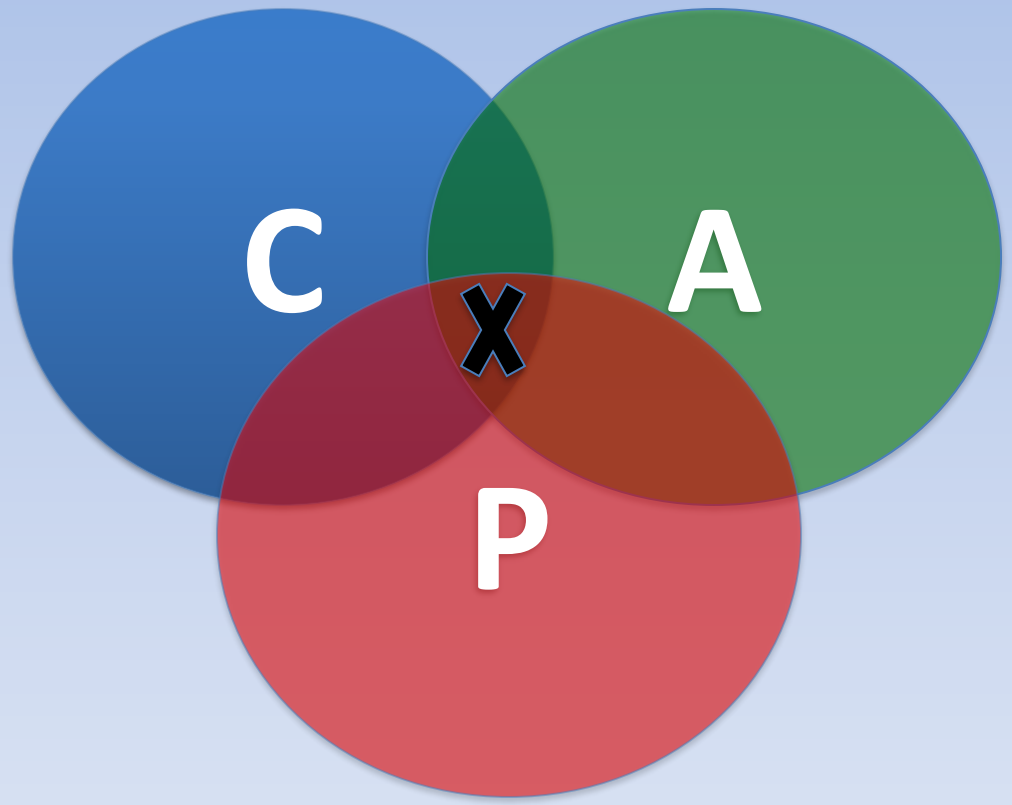
Consistency or Availability

Consistency and Availability is not "binary" decision

AP systems relax consistency in favor of availability – but are not inconsistent

CP systems sacrifice availability for consistency- but are not unavailable

This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



Types of Consistency

Strong Consistency

After the update completes, **any subsequent access** will return the **same** updated value.

Weak Consistency

It is **not guaranteed** that subsequent accesses will return the updated value.

Eventual Consistency

Specific form of weak consistency

It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)

Eventual Consistency Variations

Causal consistency

Processes that have causal relationship will see consistent data

Read-your-write consistency

A process always accesses the data item after it's update operation and never sees an older value

Session consistency

As long as session exists, system guarantees read-your-write consistency

Guarantees do not overlap sessions

Eventual Consistency Variations

Monotonic read consistency

If a process has seen a particular value of data item, any subsequent processes will never return any previous values

Monotonic write consistency

The system guarantees to serialize the writes by the *same* process

In practice

A number of these properties can be combined

Monotonic reads and read-your-writes are most desirable

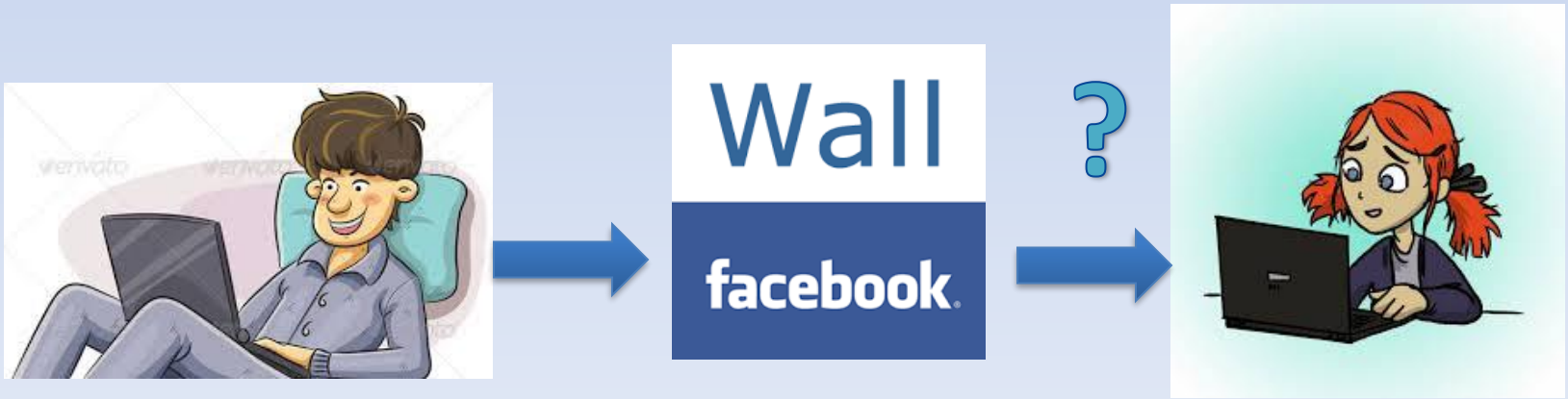
Eventual Consistency
- A Facebook Example

Bob finds an interesting story and shares with Alice by posting on her Facebook wall

Bob asks Alice to check it out

Alice logs in her account, checks her Facebook wall but finds:

- **Nothing is there!**



Eventual Consistency
- A Facebook Example

Bob tells Alice to wait a bit and check out later

Alice waits for a minute or so and checks back:

- **She finds the story Bob shared with her!**

Eventual Consistency - A Facebook Example

Reason: it is possible because Facebook uses an **eventual consistent model**

Why Facebook chooses eventual consistent model over the strong consistent one?

- Facebook has more than 1 billion active users
- It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
- Eventual consistent model offers the option to **reduce the load and improve availability**

Eventual Consistency
- A Dropbox Example

Dropbox enabled immediate consistency via synchronization in many cases.

However, what happens in case of a network partition?

Eventual Consistency
- A Dropbox Example

Let’s do a simple experiment here:

- Open a file in your drop box
- Disable your network connection (e.g., WiFi, 4G)
- Try to edit the file in the drop box: can you do that?
- Re-enable your network connection: what happens to your dropbox folder?

Eventual Consistency - A Dropbox Example

Dropbox embraces eventual consistency:

- Immediate consistency is impossible in case of a network partition
- Users will feel bad if their word documents freeze each time they hit Ctrl+S , simply due to the large latency to update all devices across WAN
- Dropbox is oriented to **personal syncing**, not on collaboration, so it is not a real limitation.

Eventual Consistency - An ATM Example

In design of automated teller machine (ATM):

- Strong consistency appear to be a nature choice
- However, in practice, **A beats C**
- Higher availability means **higher revenue**
- ATM will allow you to withdraw money *even if the machine is partitioned from the network*
- However, it puts **a limit** on the amount of withdraw (e.g., 100 EUR)
- The bank might also charge you a fee when a overdraft happens

Dynamic Tradeoff between **C** and **A**

An airline reservation system:

- When most of seats are available: it is ok to rely on somewhat out-of-date data, availability is more critical
- When the plane is close to be filled: it needs more accurate data to ensure the plane is not overbooked, consistency is more critical

Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption

Heterogeneity: Segmenting **C** and **A**

No single uniform requirement

- Some aspects require strong consistency
- Others require high availability

Segment the system into different components

- Each provides different types of guarantees

Overall guarantees neither consistency nor availability

- Each part of the service gets exactly what it needs

Can be partitioned along different dimensions

Partitioning Examples

- Data Partitioning
- Operational Partitioning
- Functional Partitioning
- User Partitioning
- Hierarchical Partitioning

Partitioning Examples

Data Partitioning

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent

Partitioning Examples

Operational Partitioning

- Each operation may require different balance between consistency and availability
- Example:
  - Reads: high availability; e.g.., “query”
  - Writes: high consistency, lock when writing; e.g., “purchase”

Partitioning Examples

Functional Partitioning

- System consists of sub-services
- Different sub-services provide different balances
- Example: A comprehensive distributed system
  - Distributed lock service (e.g., Chubby, Zookeeper) :
    - Strong consistency
  - DNS service:
    - High availability

Partitioning Examples

User Partitioning

- Try to keep related data close together to assure better performance
- Example: Craglist
  - Might want to divide its service into several data centers, e.g., east coast and west coast
    - Users get high performance (e.g., high availability and good consistency) if they query servers closet to them
    - Poorer performance if a New York user query Craglist in San Francisco

Partitioning Examples

Hierarchical Partitioning

- Large global service with local “extensions”
- Different location in hierarchy may use different consistency
- Example:
  - Local servers (better connected) guarantee more consistency and availability
  - Global servers has more partition and relax one of the requirement

What if there are no partitions?

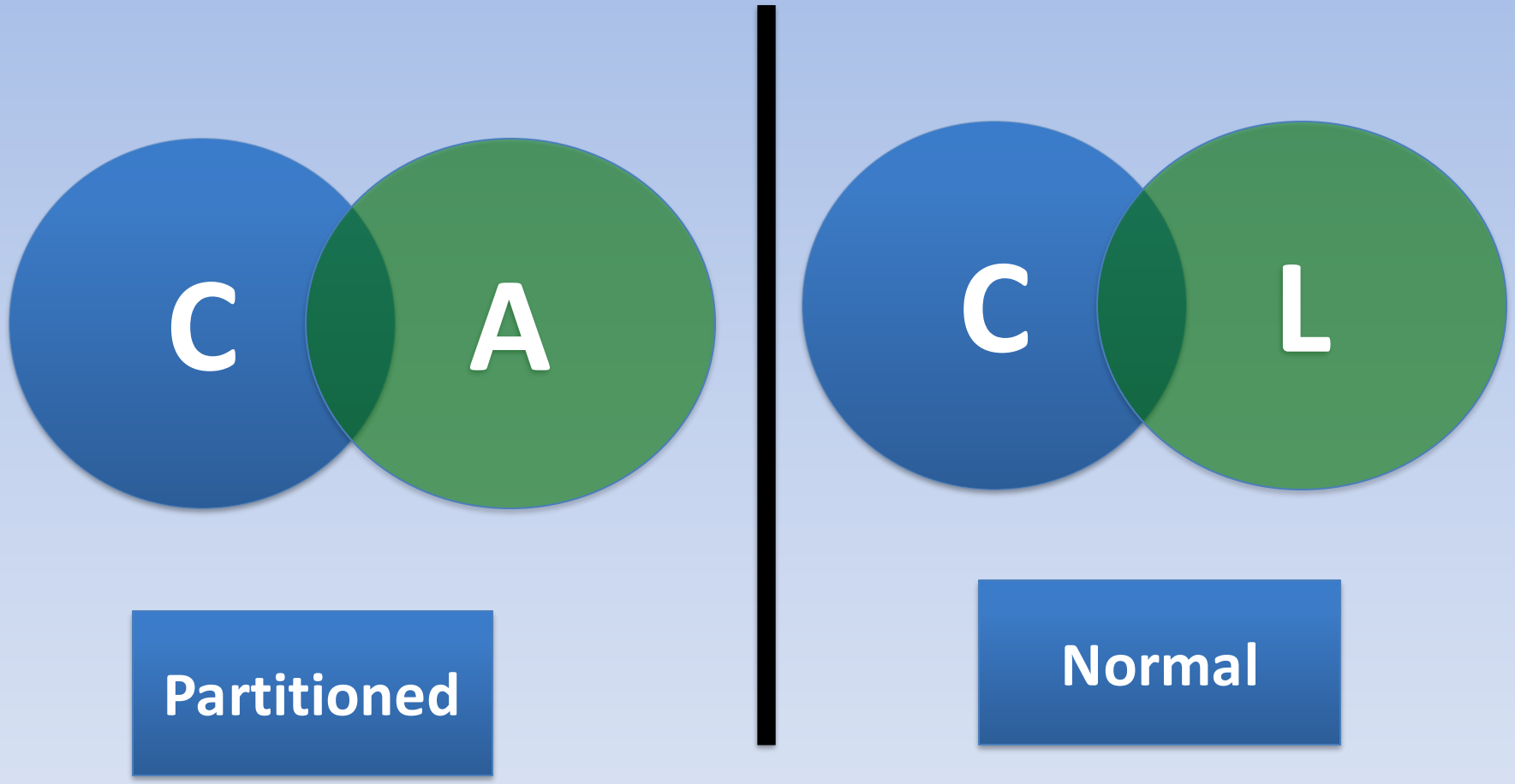
- Tradeoff between **Consistency** and **Latency**:
- Caused by the **possibility of failure** in distributed systems
  - High availability -> replicate data -> consistency problem
- Basic idea:
  - Availability and latency are arguably **the same thing**: unavailable -> extreme high latency
  - Achieving different levels of consistency/availability takes different amount of time

CAP -> PACELC

A more complete description of the space of potential tradeoffs for distributed system:

> If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?
>
> > Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

PACELC



Examples

- **PA/EL Systems:** Give up both Cs for availability and lower latency
  - Dynamo, Cassandra, Riak
- **PC/EC Systems:** Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store
- **PA/EC Systems:** Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL System:** Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Yahoo! PNUTS

State Machine Replication

- This is a model in which we can read from any replica.
- To do an update, we use an *atomic multicast* or a *durable Paxos write* (multicast if the state is kept in-memory, and durable if on disk).
- The replicas see the same updates in the same sequence and so we can keep them in a consistent state. We package the transaction into a message so "delivery of the message" performs the transactional action.

Replication in Distributed Systems

- Introduction to group services
  - Multicast: Time and Ordering
  - Failure detection
- Time and ordering
  - Implementing message ordering: FIFO, Causal, Total
  - Multicast
  - Consensus
  - Virtual Synchrony
- Failure Detection in distributed systems
  - Heartbeat algorithms
  - Gossiping

Failures are the Norm

… not the exception

Say, the rate of failure of one machine (OS/disk/motherboard/network, etc.) is once every 10 years (120 months) on average.

When you have 120 servers in the DC, the mean time to failure (MTTF) of the next machine is 1 month.

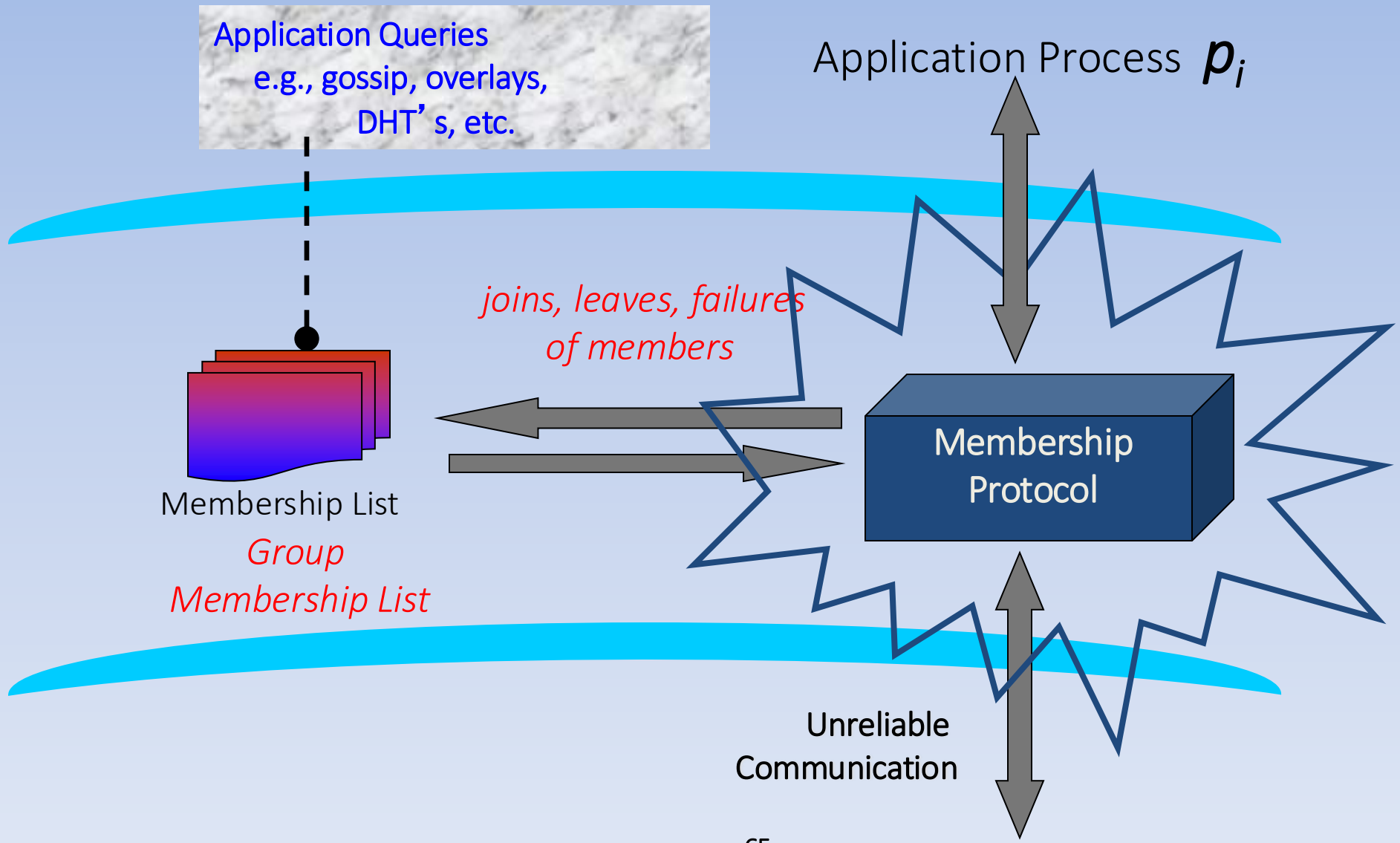
When you have 12,000 servers in the DC, the MTTF is about once every 7.2 hours!

Soft crashes and failures are even more frequent!

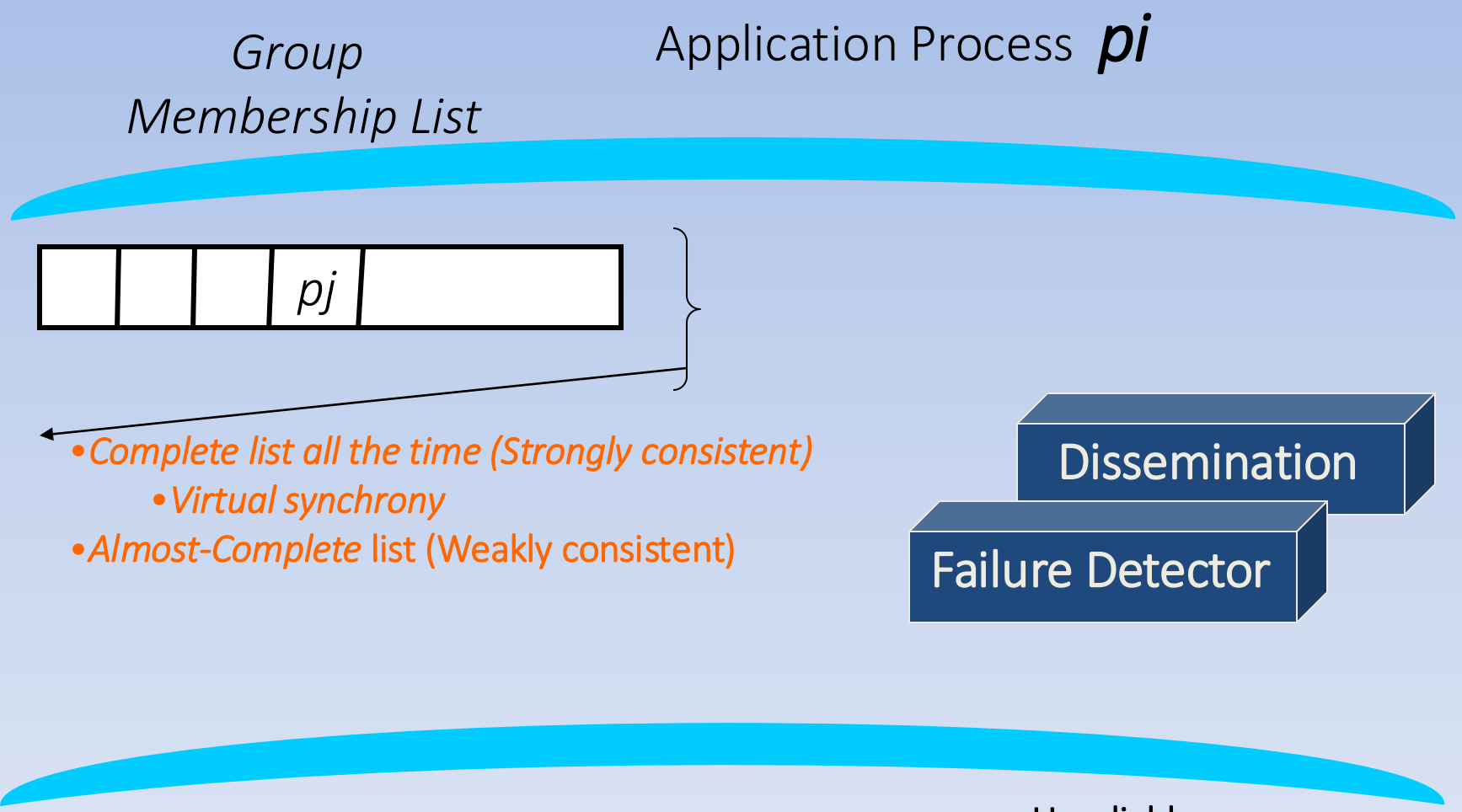
Target Settings

- Process ‘group’-based systems
  - Clouds/Datacenters
  - Replicated servers
- Fail-stop (crash) process failures

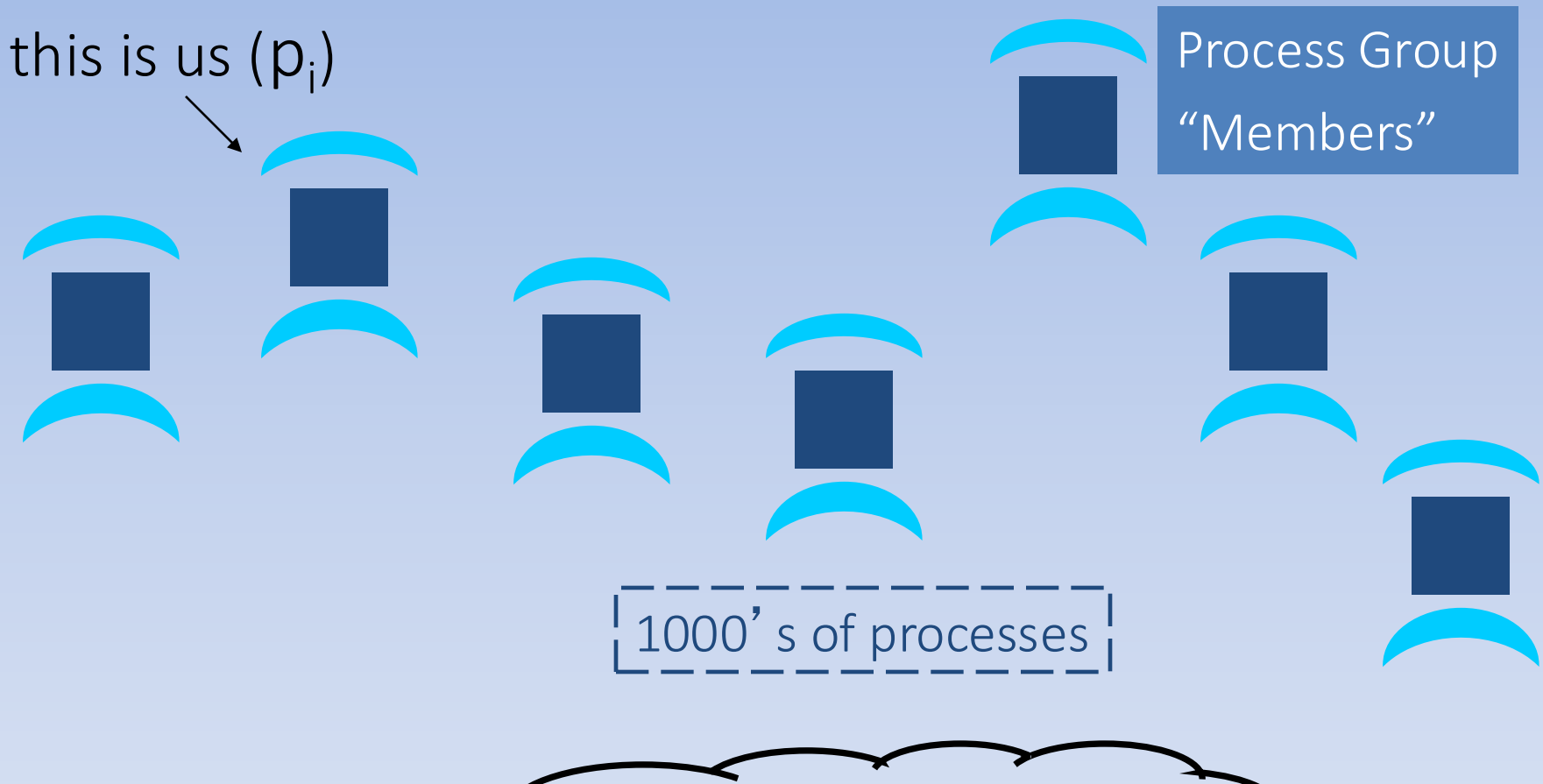
Group Membership Service



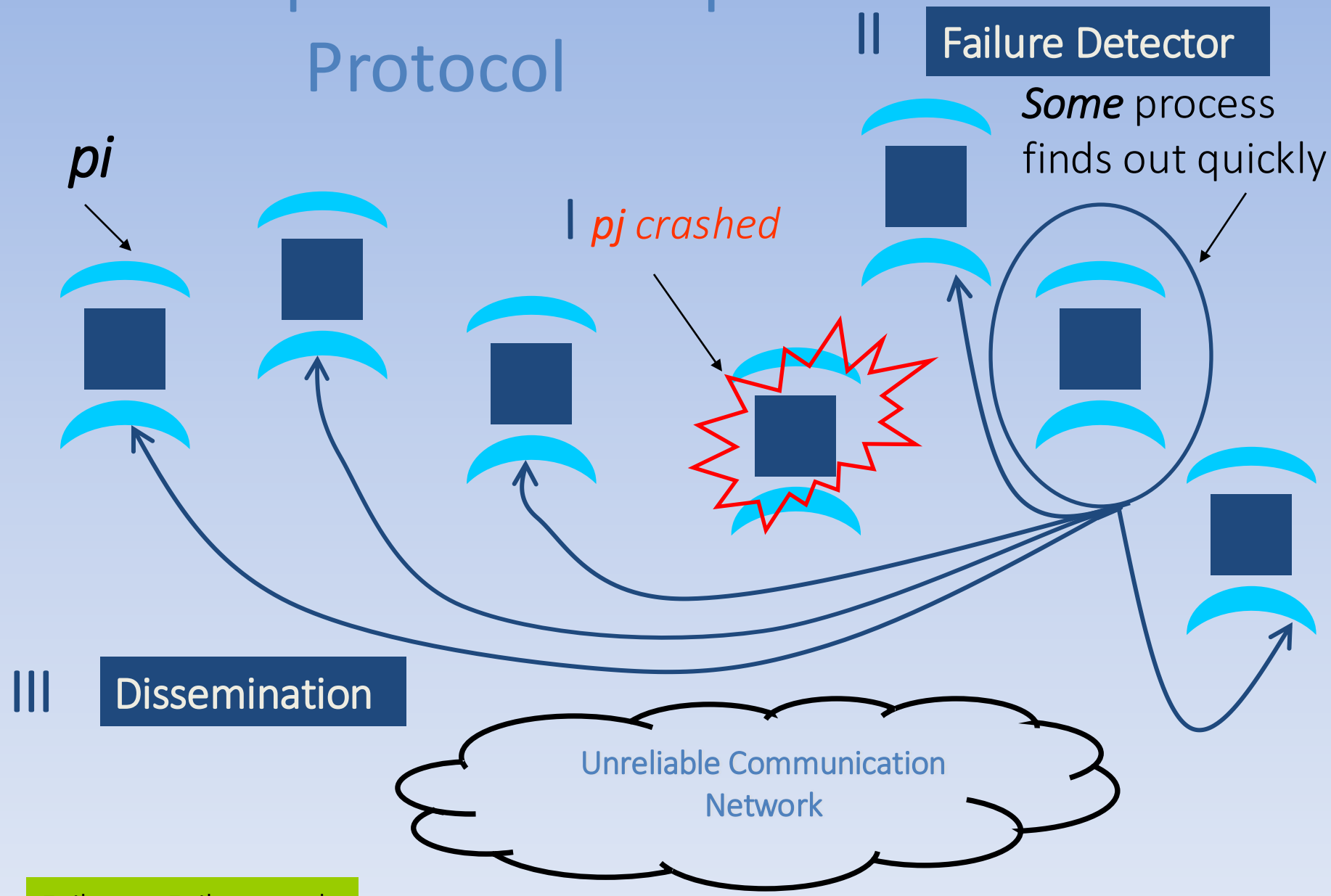
Two sub-protocols



Large Group: Scalability A Goal



Group Membership Protocol



Fail-stop Failures only