Coordination

Consensus & Agreement

- It is generally important that the processes within a distributed system have some sort of agreement
- Coordination among multiple parties involves agreement among those parties
- Agreement $\iff$ Consensus $\iff$ Consistency
- Agreement is difficult in a dynamic asynchronous system in which processes may fail or join/leave

Impossibility Theorems

- Two fundamental theorems, FLP and CAP, influences the system design choices

- FLP theorem: asynchronicity vs synchronicity

  Consensus is impossible to implement in such a way that it both a) is always correct and b) always terminates if even one machine might fail in an asynchronous system with crash fault failures

- CAP theorem: what happens when network partitions are included in the failure model

  You can’t implement consistent storage and respond to all requests if you might drop messages between processes.

FLP

- **Impossibility of Distributed Consensus with One Faulty Process**, by Fischer, Lynch and Paterson (1985)

- **Consensus Problem**: we have a set of processes, each one with a private input; the processes communicate; the processes must agree on on some process's input.

Consensus is important

- With consensus we can implement anything we can imagine:
  - leader decision
  - mutual exclusion
  - transaction commitment
  - much more…
- In some models consensus is possible, in some other models, it is not
- The goal is to learn whether, for a given model, consensus is possible or not… and prove it!

(Wrong) Consensus Protocol

- Model:
  - n > 1 processes
  - shared memory (may be accessed simultaneously by multiple processes)
  - processors can atomically *read* and *write* (not both) a shared memory location
- Protocol:
  - There is a specific memory location *C*
  - Initially *C* is in a special state $\perp$
  - Processor *1* writes its value v_1 into *C*, then decides on v_1
  - Processors $j \neq 1$ read *C* until they read something else than $\perp$ and then decide on that
- Problems with this protocol?

Consensus Properties

1. Agreement: Every correct process must agree on the same value.

2. Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.

3. Termination: All correct processes eventually reach a decision.

4. Validity: If all correct processes propose the same value V, then all correct processes decide V.

FLP System Model

- Asynchronous communication model, i.e., no upper bound on the amount of time processors may take to receive, process and respond to an incoming message

- Communication links between processors are assumed to be reliable. It is well known that given arbitrarily unreliable links no solution for consensus could be found even in a synchronous model.

- Processors are allowed to fail according to the crash fault model – this simply means that processors that fail do so by ceasing to work correctly. There are more general failure models, such as byzantine failures where processors fail by deviating arbitrarily from the algorithm they are executing.

Notation (I)

- There are *N > 1* processors which communicate by sending messages.
- A message is a pair *(p,m)* where *p* is the processor the message is intended for, and *m* is the contents of the message.
- Messages are stored in an abstract data structure called the message buffer which is a multiset – simply a set where more than one of any element is allowed – which supports two operations, *send* and *receive*.
- *send(p,m)* simply places the message *(p,m)* in the message buffer.
- *receive(p)* either returns a (random) message for processor *p* (and removes it from the message buffer) or the special value ∅, which does nothing.

Notation (II)

- Configuration: the internal state of all of the processors – the current step in the algorithm that they are executing and the contents of their memory – together with the contents of the message buffer.
- Step: the system moves from one configuration to the next by a step which consists of a processor *p* performing *receive(p)* and moving to another configuration, i.e.:
    - based on *p* local state and *m*, send an arbitrary but finite number of messages
    - based on *p* local state and *m*, change *p* local state to some new state
- Event: each step is therefore uniquely defined by the message that is received (possibly ∅) and the process *p* that received it. That pair is called an *event* (equivalent to a message)
    - Configurations move from one to another through events.
    - An event *e* can be applied to a configuration *C* if either *m* is ∅ or *(p,m)* is in the message buffer
    - *C' = e(C)* means that if we apply event *e* to configuration *C* we move to configuration *C'*
- Execution: a possibly infinite sequence of events from a specific initial configuration.
    - Since the receive operation is non-deterministic, there are many different possible executions for a given initial configuration.

Notation (III)

- Schedule & Run: a particular execution σ, defined by a possibly infinite sequence of events from a starting configuration, is called a *schedule* and the sequence of steps taken to realize the schedule is a *run*.
    - Non-faulty processes take infinitely many steps in a run (presumably eventually just receiving $\varnothing$ once the algorithm has finished its work) – otherwise a process is considered faulty.
    - σ can be applied to configuration C if the events in σ can be applied to C in order
    - $C' = \sigma(C)$ means that if we apply schedule σ to configuration C we move to configuration C'
- An admissible run is one where at most one process is faulty (capturing the failure requirements of the system model) and every message is eventually delivered (this means that every processor eventually gets chosen to receive infinitely many times).
- We say that a run is a deciding run provided that some process eventually decides according to the properties of consensus, and that a consensus protocol is totally correct if every admissible run is a deciding run.

Proof Sketch

- FLP Theorem [1985]. No totally correct consensus algorithm exists (for the given system model).

- The idea behind it is to show that there is *some admissible run* – i.e., one with only one processor failure and eventual delivery of every message – *that is not a deciding run* – i.e., in which no processor eventually decides and the result is a protocol which runs for ever (because no processor decides).

- Two processors and binary consensus values

PAXOS

- Leslie Lamport. **The part-time parliament**. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.
- Leslie Lamport described in 1990 the algorithm as the solution to a problem of the parliament on a fictitious Greek island called Paxos (not Italy)
- Many readers were so distracted by the description of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected.
- Leslie Lamport refused to rewrite the paper. He later wrote that he “was quite annoyed at how humorless everyone working in the field seemed to be”
- After a few years, some people started to understand the importance of the algorithm
- After eight years, Leslie Lamport submitted the paper again, basically unaltered. It got accepted!

Leslie Lamport
ACM Turing Award 2014

Correctness vs. Termination

- In asynchronous systems, we cannot guarantee termination and correctness at the same time
- PAXOS is correct, so termination is not guaranteed
- PAXOS cannot guarantee that a consensus is reached in a finite number of steps
- In practice, PAXOS can be optimized to reduce probability of no termination
- For example, the acceptors could send NAK if they do not accept a prepare message or a proposal (this optimization increases the message complexity)
- PAXOS is used in Apache's Zookeeper and Google's Chubby

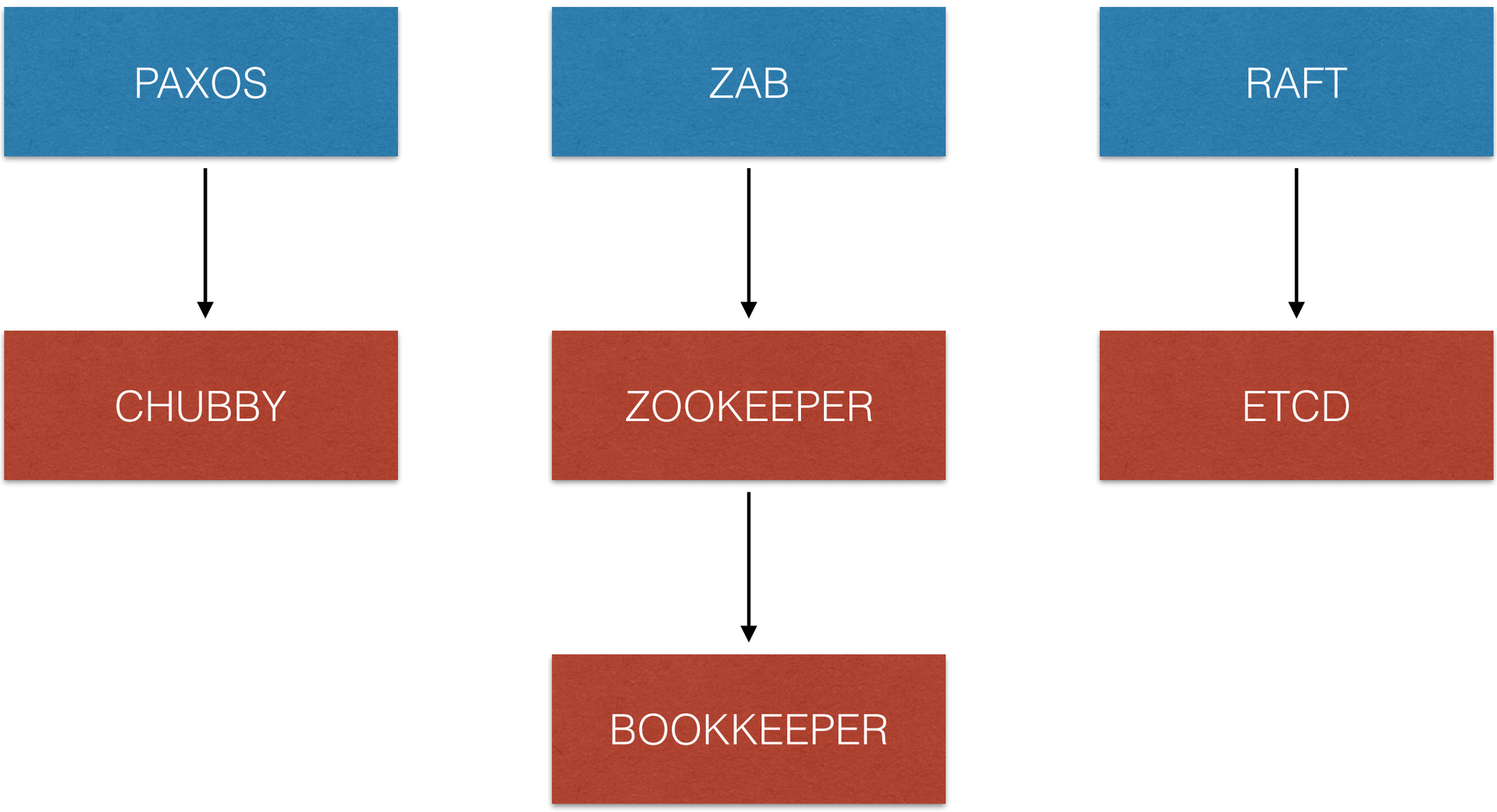
PAXOS Protocols

- PAXOS employs consensus to serialize operations at a leader and apply the operations at each replica in this exact serialized order dictated by the leader.

- ZAB (ZooKeeper Atomic Broadcast) is referred to as an *atomic broadcast protocol* because it enables the nodes to deliver the same set of transactions (state updates) in the same order. Atomic broadcast or total order broadcast and consensus are equivalent problems.

- RAFT is a recent consensus protocol that was designed to enhance understandability of the Paxos protocol

Paxos Protocols vs. Systems



PAXOS Use Criteria

1. Paxos system should not be in the performance critical path of the application.
2. Frequency of write operations to the Paxos system should be kept low.
3. Amount of data maintained in the Paxos system should be kept small.
   Preferably only small metadata should be stored/maintained in the Paxos system.
4. Application adopting the Paxos system should really require strong consistency.
5. Application adopting the Paxos system should not be distributed over the Wide Area Network (WAN).
6. The API abstraction should be fit the goal.

PAXOS Use Patterns (I)

- **Server Replication**. Server replication via the state machine replication (SMR) approach is a canonical application for Paxos protocol. The SMR requires a state machine to be deterministic: multiple copies of the state machine begin in the start state and receive the same inputs in the same order and each replica will arrive at the same state having generated the same outputs. Paxos is used for serializing and replicating the operations to all nodes in order to ensure that states of the machines are identical and the same sequence of operations is applied.

- **Log Replication**. The objective of log replication is different than that of server replication. Log replication is applied in data integration systems that use the log abstraction to duplicate data across different nodes, while server replication is used in SMR to make copies of the server state. Since Paxos systems such as ZooKeeper have limited storage, they are not typically suitable for the data- centric/intensive task of log replication.

PAXOS Use Patterns (II)

- **Synchronization Service**. An important application of consensus is to provide synchronization. Traditionally, concurrent access to the shared data is controlled by some form of mutual exclusion through locks. However such approach requires applications to build their own failure detection and recovery mechanism, and a slow or blocked process can harm the overall performance. When the consensus protocol/ system is decoupled from the application, the application not only gains fault tolerance of the shared data, but also achieves wait-free concurrent data access with guaranteed consistency.

- **Barrier Orchestration**. Large-scale graph processing systems based on BSP (Bulk Synchronous Parallel) model like Google Pregel, Apache Giraph and Apache Hama use Paxos systems for coordination between computing processes.

PAXOS Use Patterns (III)

- **Configuration Management**. Most Paxos systems provide the ability to store arbitrary data by exposing a filesystem or key-value abstraction to the systems. This gives the applications access to durable and consistent storage for small data items that can be used to maintain configuration metadata like connection details or feature flags. These metadata can be watched for changes, allowing applications to reconfigure themselves when configuration parameters are modified. Leader election (LE), group membership (GM), service discovery (SD), and metadata management (MM) are main use cases under configuration management, as they are important for cluster management in cloud computing systems.

- **Message Queues**. A common misuse pattern is to use the Paxos system to maintain a distributed queue, such as a publisher-subscriber message queue or a producer-consumer queue. Unfortunately, queues in production can contain many thousands of messages resulting in a large volume of write operations and potentially huge amounts of data going through the Paxos system. Moreover, in this case the Paxos system stands in the critical path of every queue operation.

Consequences of FLP

- There is no way to solve the consensus problem under a very minimal system model in a way that cannot be delayed forever
- Complete correctness if not possible in asynchronous models
- In practice, we may live with very low probability of disagreement (give up safety)
- In practice, we may live with very low probability of blocking (give up liveness)
- Two-phase commit or even three-phase commit can block forever
- This result is particularly relevant to people designing algorithms

CAP

- Presented as Brewer's Conjecture in 2000

- Formalized and proved in **Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**, by Lynch and Gilbert (2002)

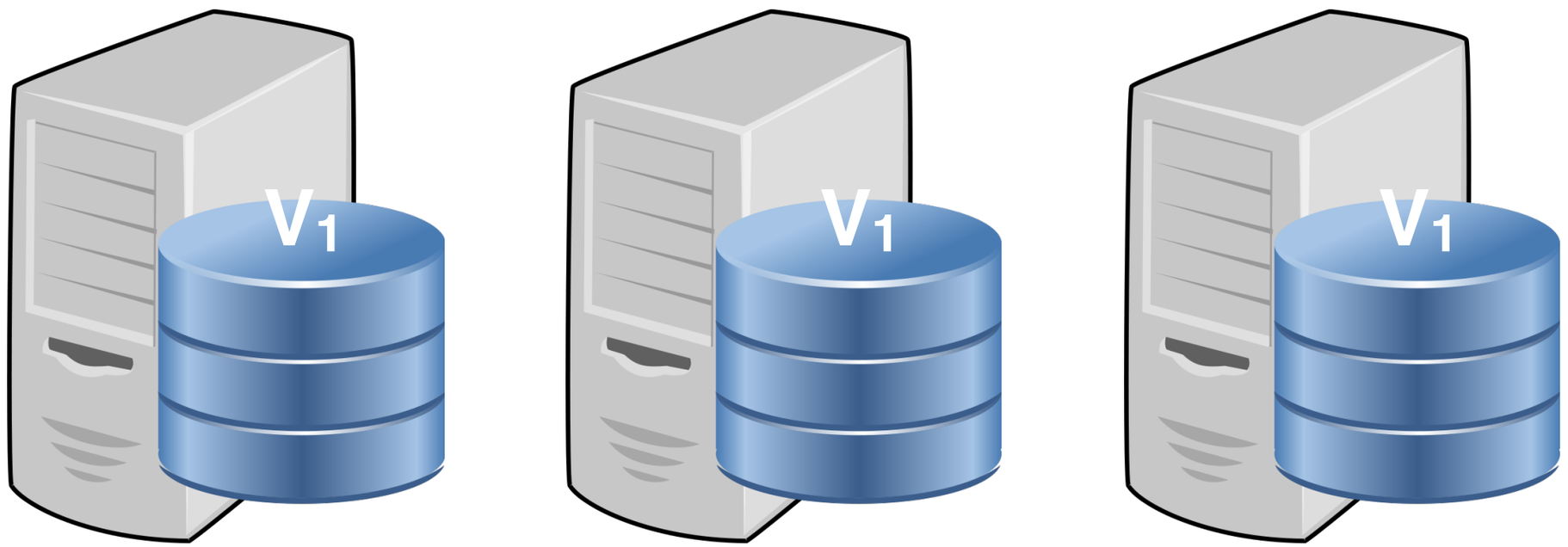
- **Consistency**, **availability** and **partition-tolerance** cannot be achieved all at the same time in a distributed system.

- Simply, in an asynchronous network that performs as expected, where messages may be lost (partition-tolerance), it is impossible to implement a service providing correct data (consistency) and eventually responding to every request (availability) under every pattern of message loss.

- Slides from http://cs-wwwarchiv.cs.unibas.ch/lehre/hs10/cs341/_Downloads/Workshop/Talks/2010-HS-DIS-I_Giangreco-CAP_Theorem-Talk.pdf

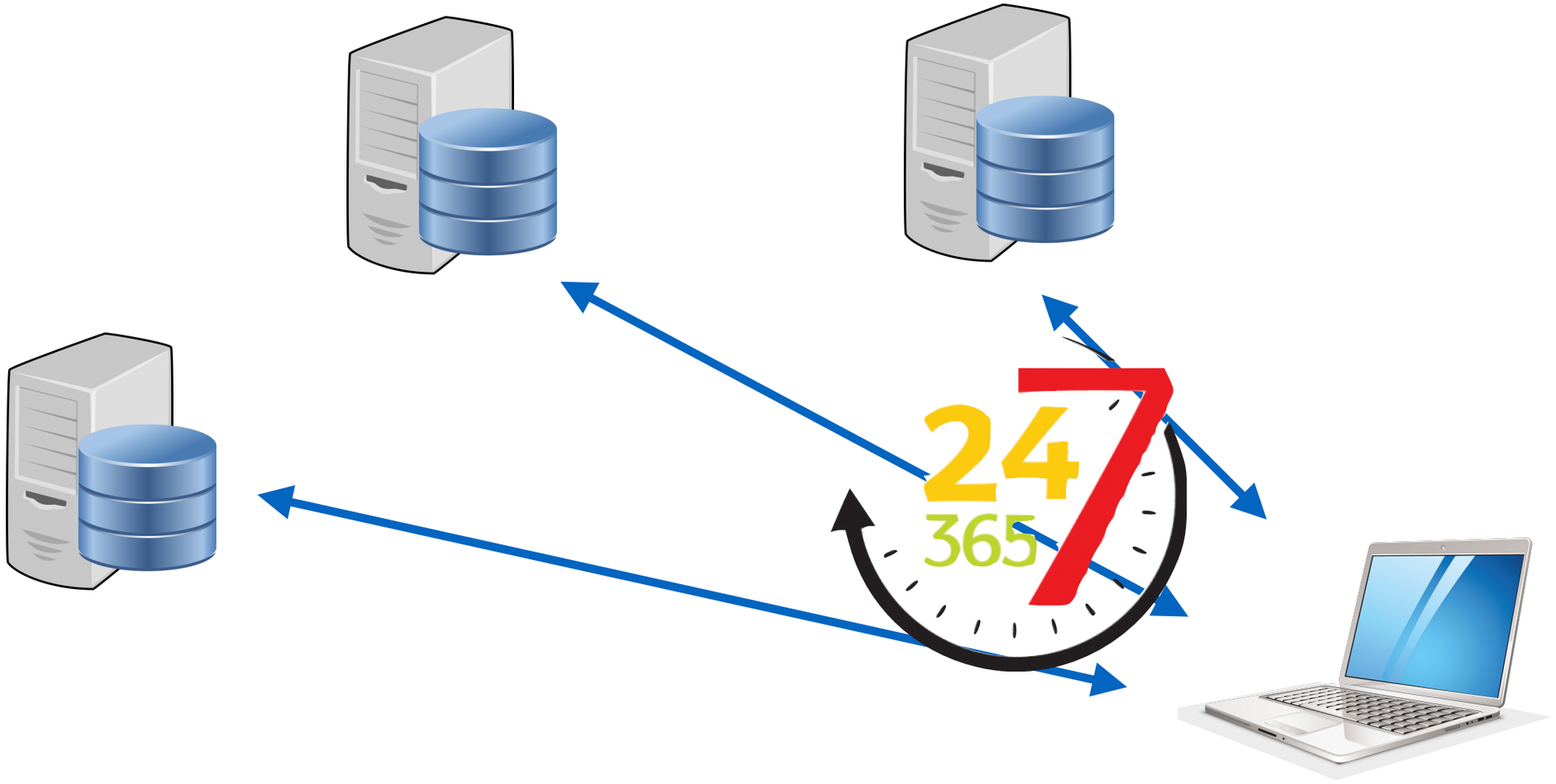
Consistency

- All the nodes in the system see the same state of the data
- Formally, we speak of *atomic* or *linearizable consistency*
- There exists a sequential order on all operations which is consistent with the order of invocations and responses, such that each operation looks as if it were completed at a single instant.



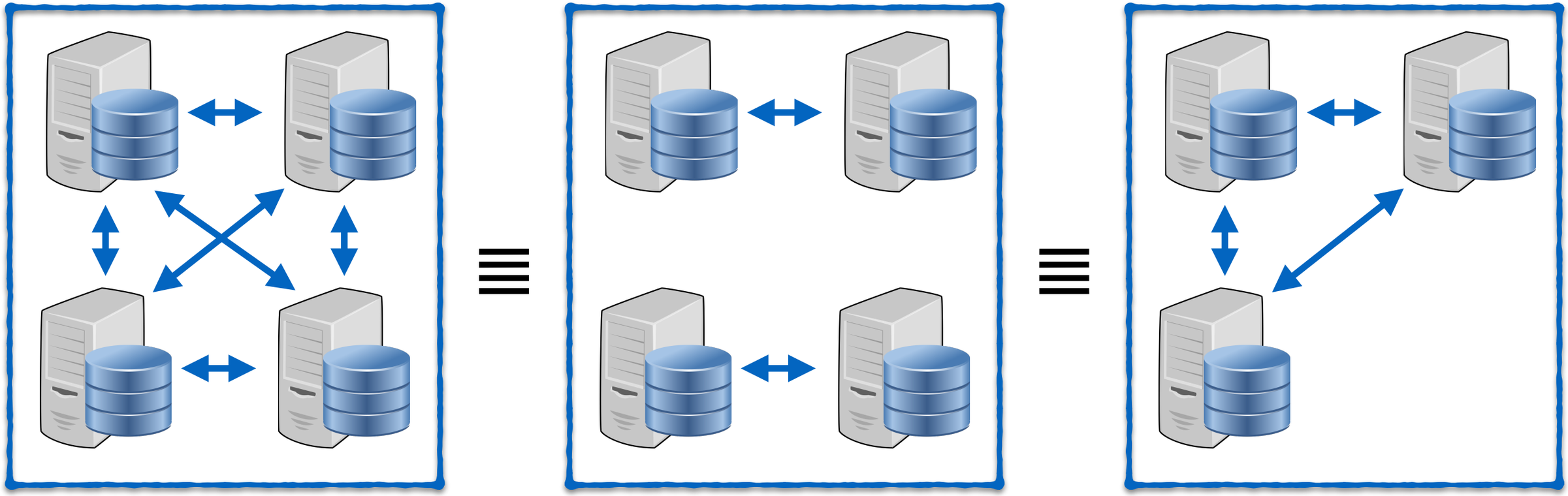
Availability

- Every request received by a non-failing node should be processed and must result in a response



Partition Tolerance

- If some nodes crash and/or some communications fail, system still performs as expected



CAP Theorem 1

It is impossible in the **asynchronous network model** to implement a read/write data object that guarantees the following properties:

- Availability
- Atomic consistency

in all fair executions (including those in which messages are lost).

Asynchronous, i. e. there is no clock, nodes make decisions based only on the messages received and local computation.

CAP Theorem 2

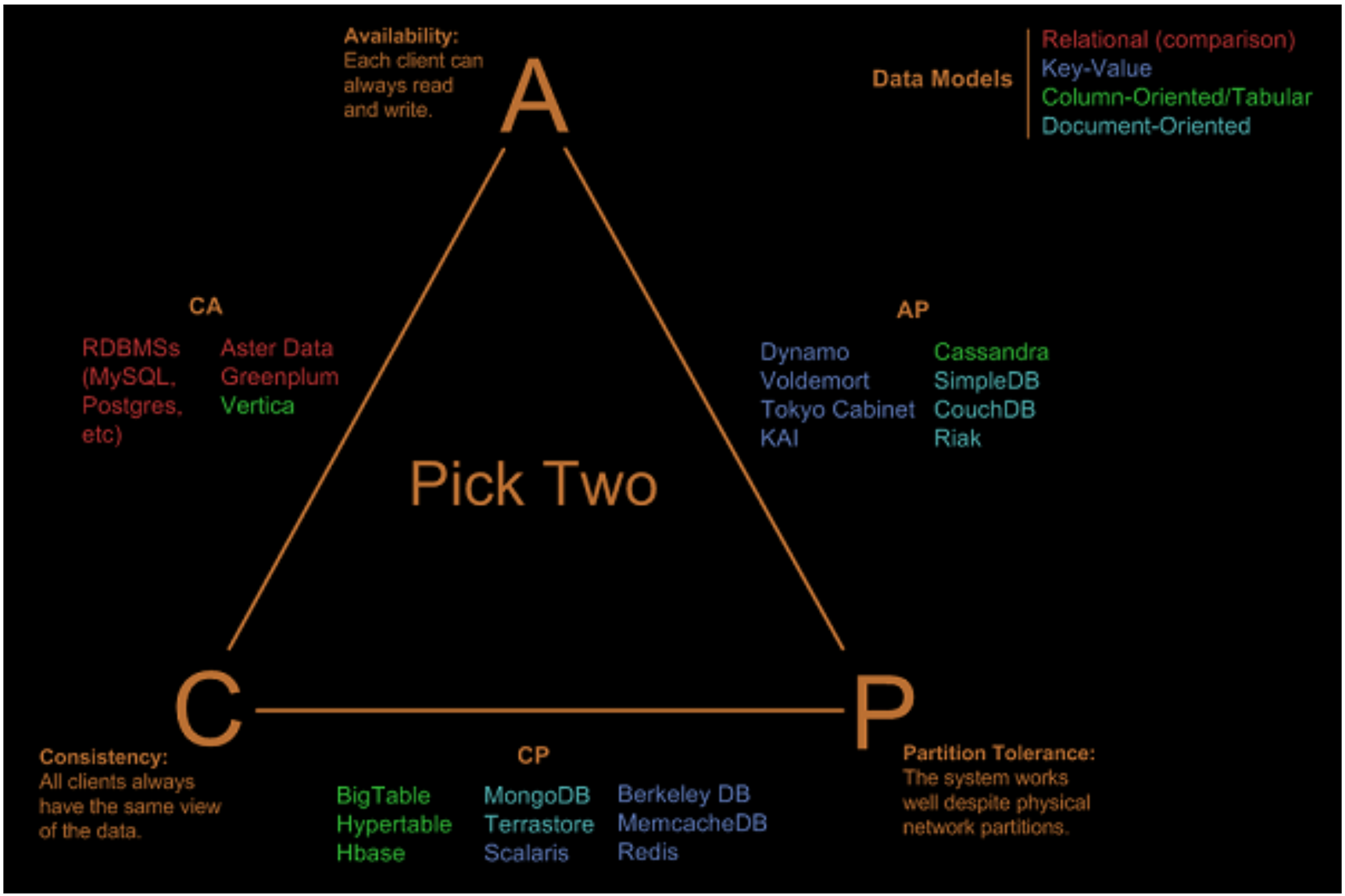
It is impossible in the **partially synchronous network model** to implement a read/write data object that guarantees the following properties:

- Availability
- Atomic consistency

in all fair executions (including those in which messages are lost).

Partially synchronous, i. e. every node has a clock, and all clocks increase at the same rate. However, they are not synchronized.

Consequences of CAP



Consequences of CAP

- When partitions are rare (e.g., parallel systems), CAP should allow perfect C and A most of the time

- In distributed systems it is not possible to avoid network partitions.

- There is not a need to choose between either C or A, instead, it is more an act of balancing between the two properties.

Practical Consequences of CAP

- Many system designs used in early distributed relational database systems did not take into account partition tolerance (e.g. they were CA designs).

- There is a tension between strong consistency and high availability during network partitions. A distributed system consisting of independent nodes connected by an unpredictable network cannot behave in a way that is indistinguishable from a non-distributed system.

- There is a tension between strong consistency and performance in normal operation. Strong consistency requires that nodes communicate and agree on every operation. This results in high latency during normal operation.

- If we do not want to give up availability during a network partition, then we need to explore whether consistency models other than strong consistency are workable for our purposes.