Impossibility Results:

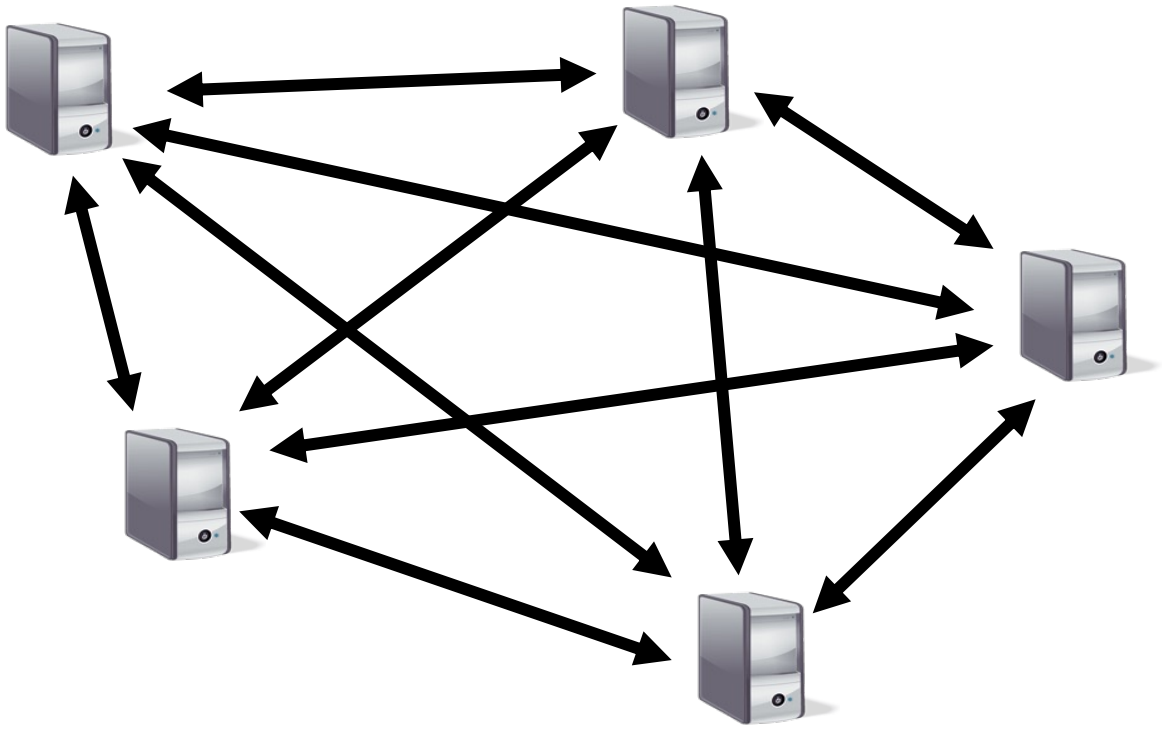
CAP, PRAM & FLP

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

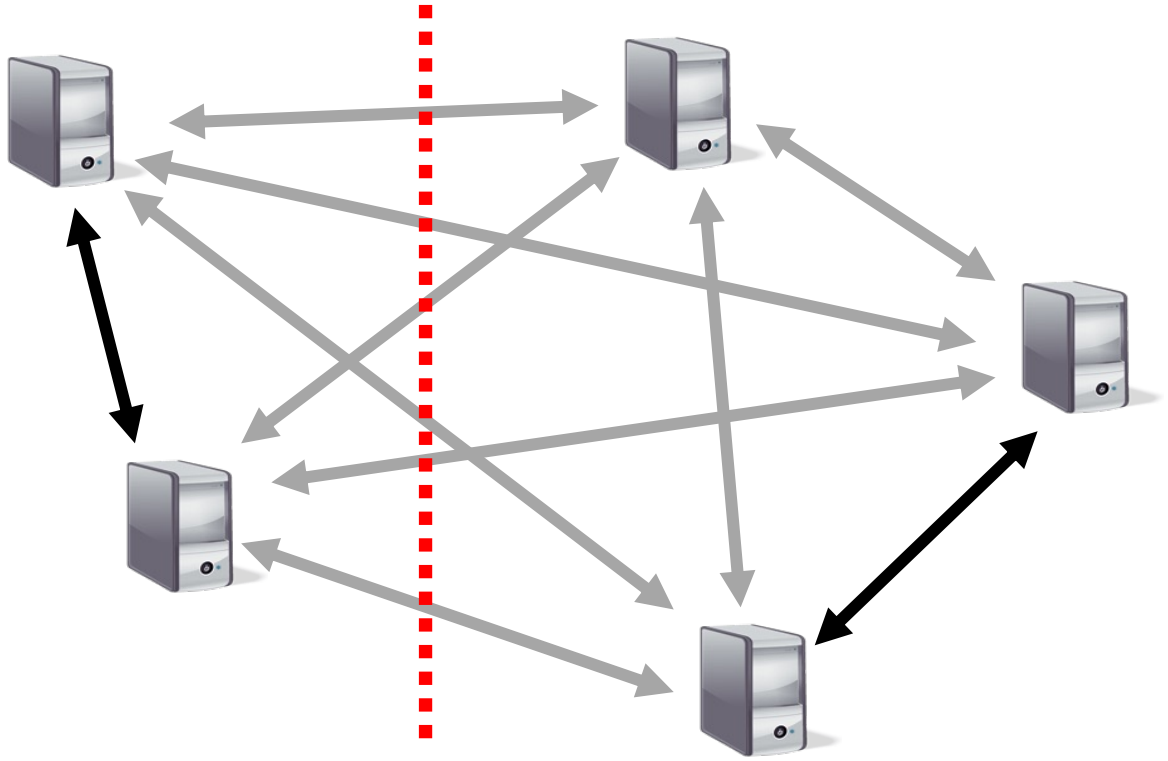
CS 240: Computing Systems and Concurrency
Lecture 16

Marco Canini

Network partitions divide systems



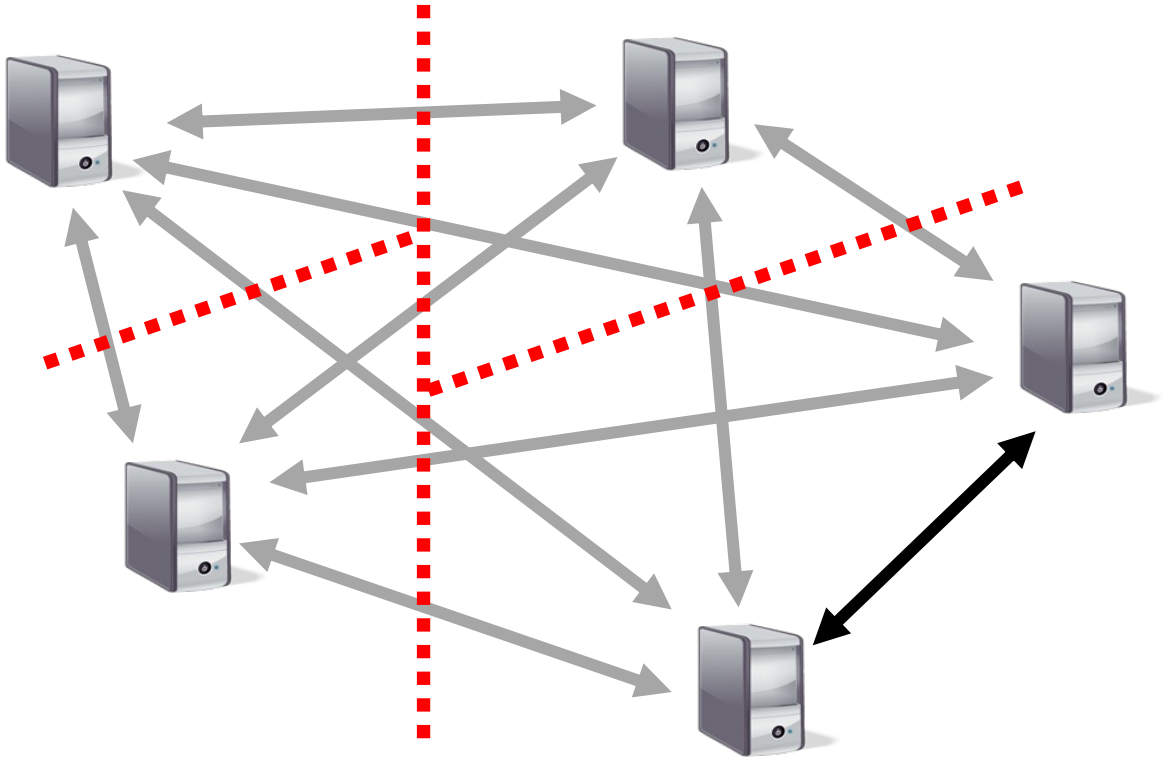
Network partitions divide systems



How can we handle partitions?

- Totally-ordered Multicast?
- Bayou?
- Dynamo?
- Chord?
- Paxos?
- RAFT?

How about this set of partitions?



Fundamental trade-off?

- Replicas appear to be a single machine, but lose availability during a network partition

OR

- All replicas remain available during a network partition but do not appear to be a single machine

CAP theorem preview

- You cannot achieve all three of:
  1. Consistency
  2. Availability
  3. Partition-Tolerance
- Partition Tolerance => Partitions Can Happen
- Availability => All Sides of Partition Continue
- Consistency => Replicas Act Like Single Machine
  - Specifically, Linearizability

Impossibility Results Useful!!!!

- Fundamental tradeoff in design space
  - **Must** make a choice
- Avoids wasting effort trying to achieve the impossible
- Tells us the best-possible systems we can build!

CAP conjecture [Brewer 00]

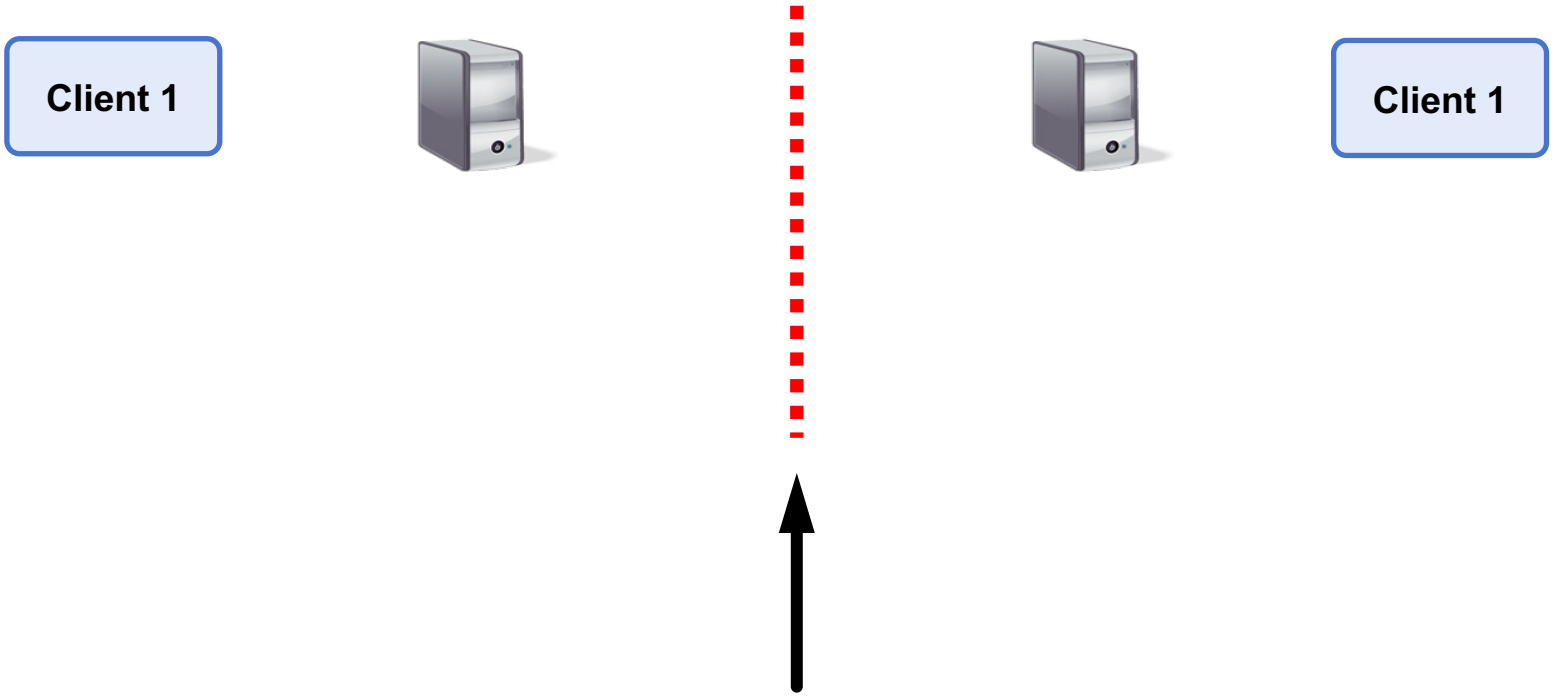
- From keynote lecture by Eric Brewer (2000)
  - History: Eric started Inktomi, early Internet search site based around "commodity" clusters of computers
  - Using CAP to justify "BASE" model: Basically Available, Soft-state services with Eventual consistency
- Popular interpretation: 2-out-of-3
  - Consistency (Linearizability)
  - Availability
  - Partition Tolerance: Arbitrary crash/network failures

CAP theorem [Gilbert Lynch 02]

Assume to contradict that Algorithm *A* provides all of CAP

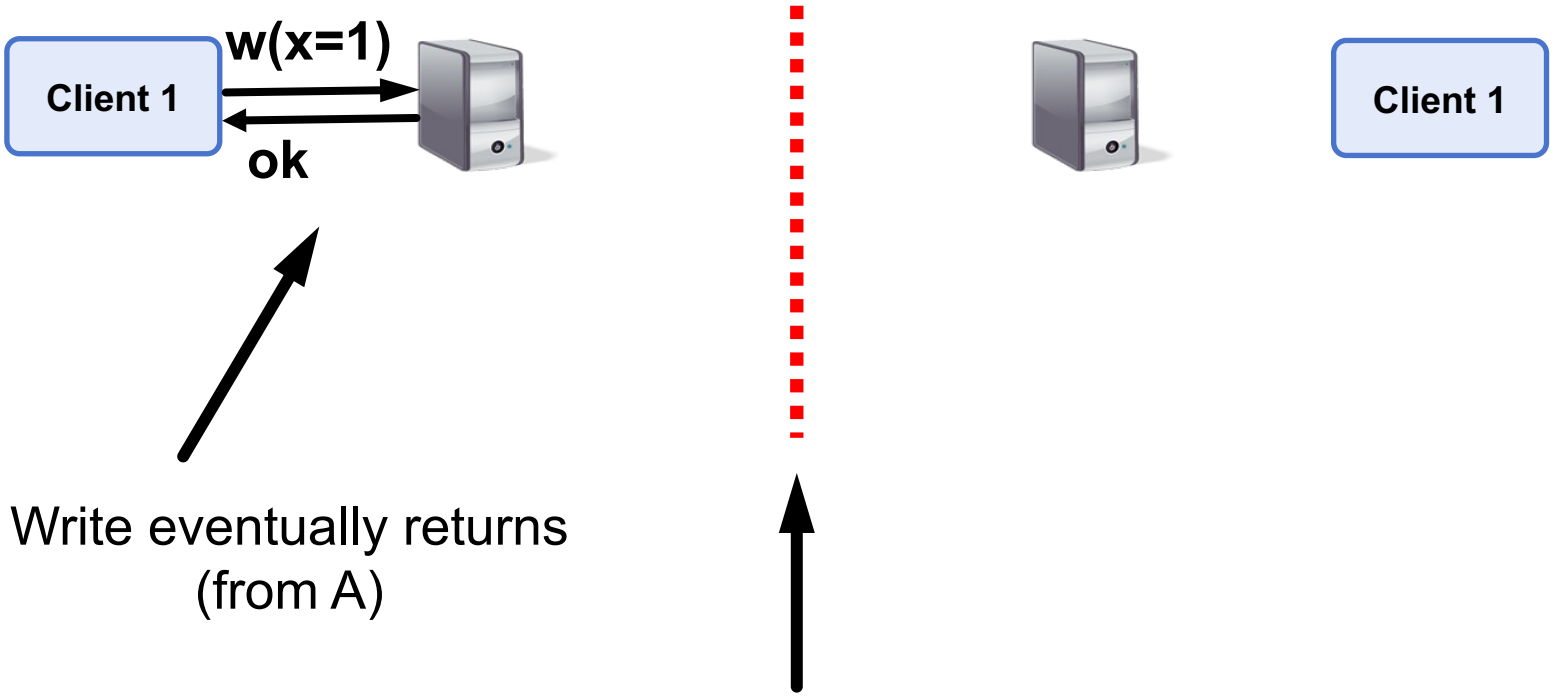
CAP theorem [Gilbert Lynch 02]

Assume to contradict that Algorithm *A* provides all of CAP



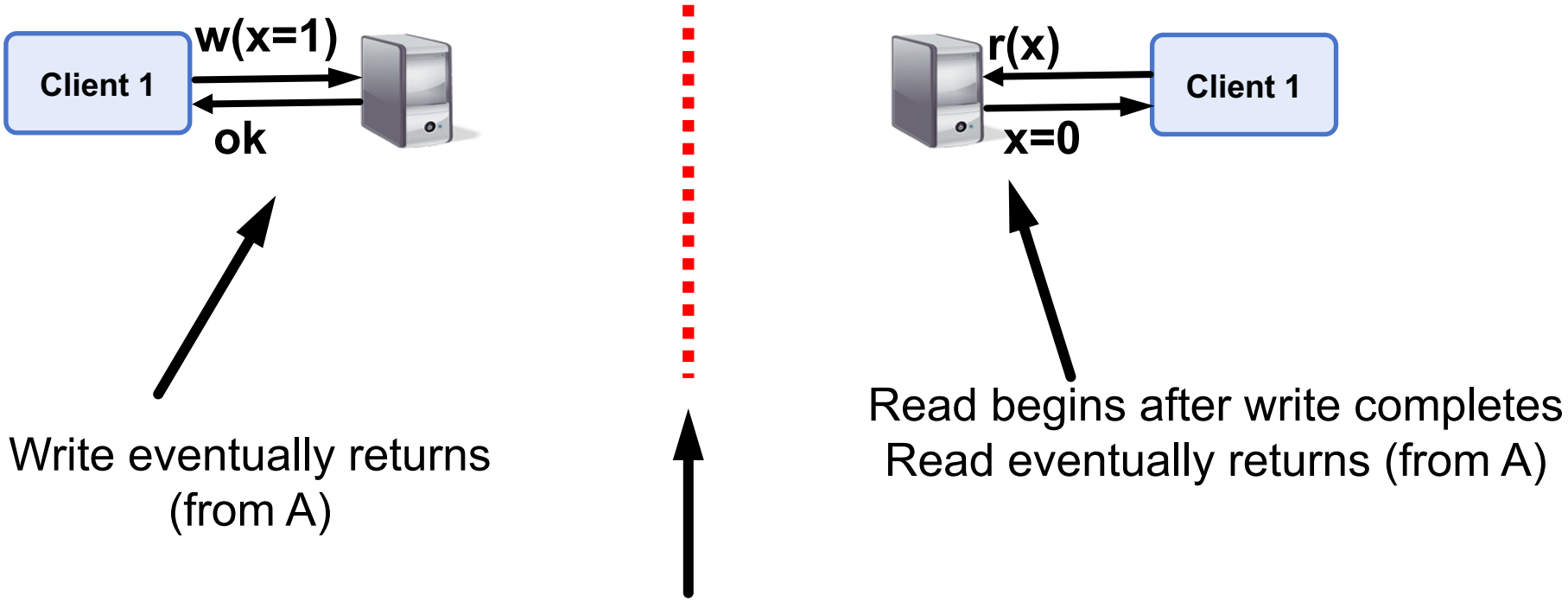
CAP theorem [Gilbert Lynch 02]

Assume to contradict that Algorithm *A* provides all of CAP



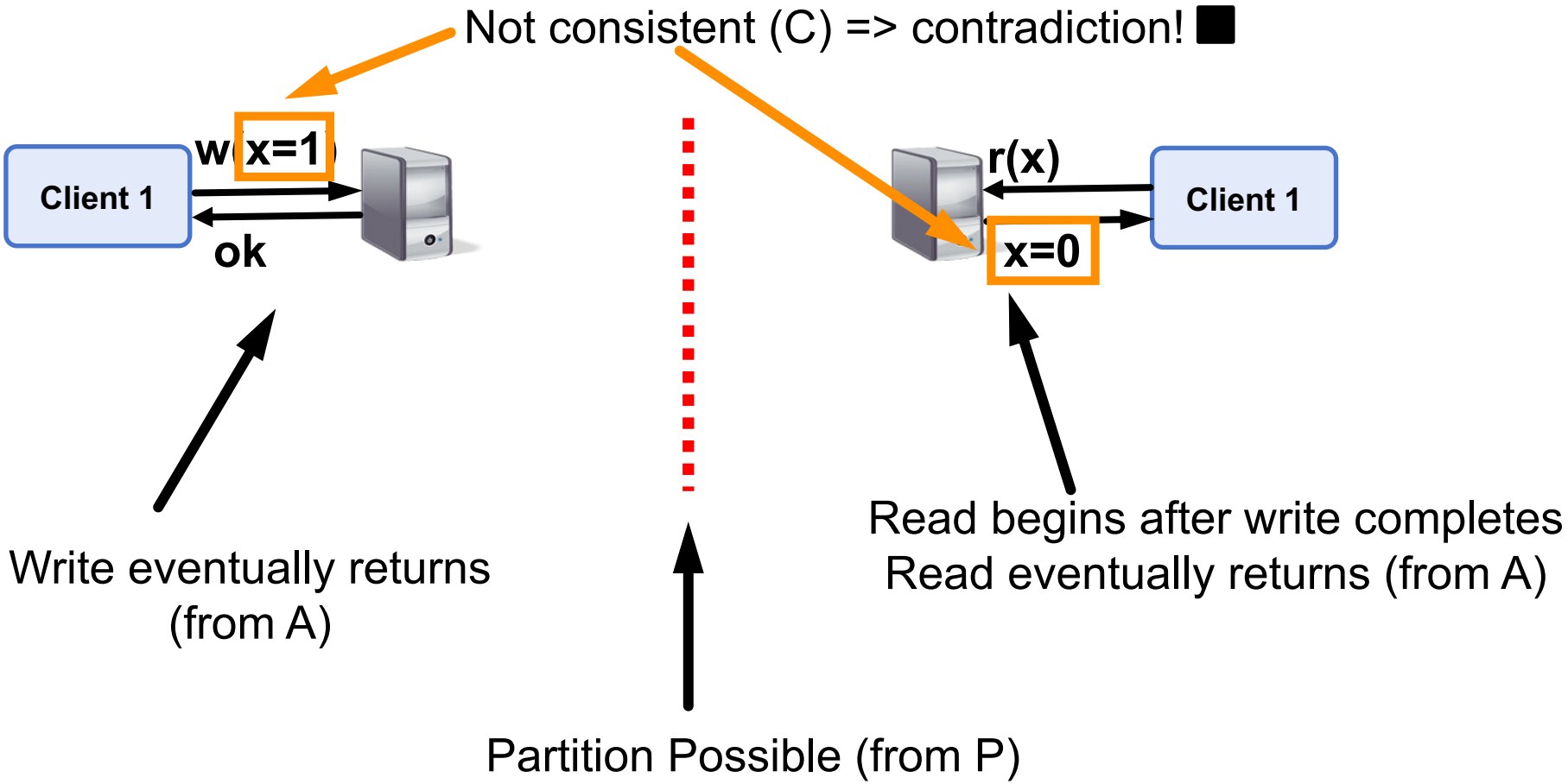
CAP theorem [Gilbert Lynch 02]

Assume to contradict that Algorithm *A* provides all of CAP



CAP theorem [Gilbert Lynch 02]

Assume to contradict that Algorithm *A* provides all of CAP



CAP Interpretation Part 1

- Cannot “choose” no partitions
  - 2-out-of-3 interpretation doesn’t make sense
  - Instead, availability OR consistency?

- i.e., fundamental trade-off between availability and consistency
  - When designing system must choose one or the other, both are not possible

CAP Interpretation Part 2

- Cannot “beat” CAP theorem

- Can engineer systems to make partitions extremely rare, however, and then just take the rare hit to availability (or consistency)

More trade-offs L vs. C

- Low-latency: Speak to fewer than quorum of nodes?
  - 2PC: write N, read 1
  - RAFT: write $\lfloor N/2 \rfloor + 1$, read $\lfloor N/2 \rfloor + 1$
  - General: $|W| + |R| > N$

- L and C are fundamentally at odds
  - "C" = linearizability, sequential, serializability (more later)

PACELC

- If there is a partition (P):
  - How does system tradeoff A and C?
- Else (no partition)
  - How does system tradeoff L and C?

- Is there a useful system that switches?
  - Dynamo: PA/EL
  - “ACID” dbs: PC/EC

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

PRAM [Lipton Sandberg 88] [Attiya Welch 94]

- *d* is the worst-case delay in the network over all pairs of processes [datacenters]
- Sequentially consistent system
- read time + write time ≥ *d*

PRAM Theorem:

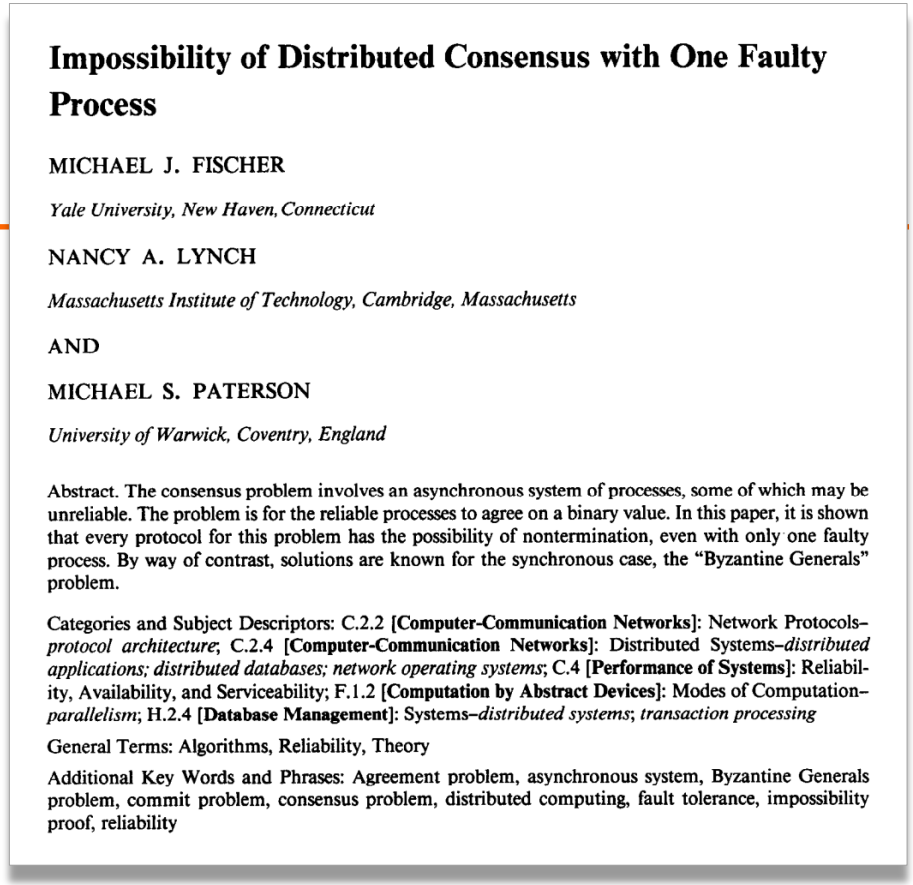
Impossible for sequentially consistent system to always provide low latency

PRAM [Lipton Sandberg 88] [Attiya Welch 94]

- Fundamental tradeoff between consistency and latency!
- Proof intuition (see papers for details)
- Let P1 and P2 be the 2 **furthest away** processes; assume 2 objects: x, y
- Assume to contradict read time + write time < d
- Thus the following executions are possible because P1's Write can't be seen at P2 Read:
- P1: |--W(x=1)--| |--R(y)=0--|
- P2: |--W(y=1)--| |--R(x)=0--|
- But there is no total order of these operations, so does not provide sequential consistency

"FLP" result

- No deterministic 1-crash-robust consensus algorithm exists with asynchronous communication



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 **[Computer-Communication Networks]**: Network Protocols–*protocol architecture*; C.2.4 **[Computer-Communication Networks]**: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 **[Performance of Systems]**: Reliability, Availability, and Serviceability; F.1.2 **[Computation by Abstract Devices]**: Modes of Computation–*parallelism*; H.2.4 **[Database Management]**: Systems–*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

FLP is the original impossibility result for distributed systems!

- Useful interpretation: no consensus algorithm can always reach consensus with an asynchronous network
  - Do not believe such claims!

- Led to lots and lots of theoretical work
  - (Consensus is possible when the network is reasonably well-behaved)

FLP's weak assumptions

- Only 1 failure
  - Also impossible for more failures
- For "weak" consensus (only some process needs to decide)
  - Also impossible for real consensus
- For reliable communication
  - Also impossible for unreliable communication
- For only two states: 0 and 1
  - Also impossible for more failures
- For crash failures
  - Also impossible for Byzantine failures

FLP's strong assumptions

- Deterministic actions at each node
- Asynchronous network communication
- All "runs" must eventually achieve consensus

Main technical approach

- Initial state of system can end in decision “0” or “1”

- Consider 5 processes, each in some initial state

[1,1,1,1,1] → 1
[1,1,1,1,0] → ?
[1,1,1,0,0] → ?
[1,1,0,0,0] → ?
[1,0,0,0,0] → 0

Must exist two configurations here which differ in decision

Main technical approach

- Initial state of system can end in decision “0” or “1”

- Consider 5 processes, each in some initial state

[1,1,1,1,1] → 1
[1,1,1,1,0] → 1
[1,1,1 0,0] → 1
[1,1,0 0,0] → 0
[1,0,0,0,0] → 0

Assume decision differs between these two processes

Main technical approach

- Goal: Consensus holds in face of 1 failure

One of these configurations must be "bi-valent" (i.e., undecided): Both futures possible

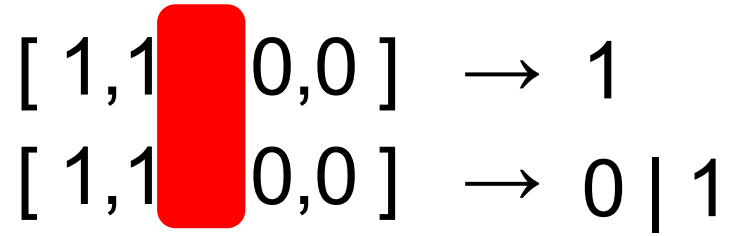
[1,1 0,0] → 1 | 0

[1,1 0,0] → 0

Main technical approach

- Goal: Consensus holds in face of 1 failure

One of these configurations must be "bi-valent" (i.e., undecided): Both futures possible



- Inherent non-determinism from asynchronous network
- Key result: All bi-valent states can remain in bi-valent states after performing some work

Staying bi-valent forever

1. System thinks process *p* failed, adapts to it…
2. But no, *p* was merely slow, not failed…
   (Can't tell the difference between slow and failed.)
3. System think process *q* failed, adapts to it…
4. But no, *q* was merely slow, not failed…
5. Repeat ad infinitum …

Consensus is impossible

But, we achieve consensus all the time…

FLP's strong assumptions

- Deterministic actions at each node
  - Randomized algorithms can achieve consensus
- Asynchronous network communication
  - Synchronous or even partial synchrony is sufficient
- All "runs" must eventually achieve consensus
  - In practice, many "runs" achieve consensus quickly
  - In practice, "runs" that never achieve consensus happen vanishingly rarely
    - Both are true with good system designs