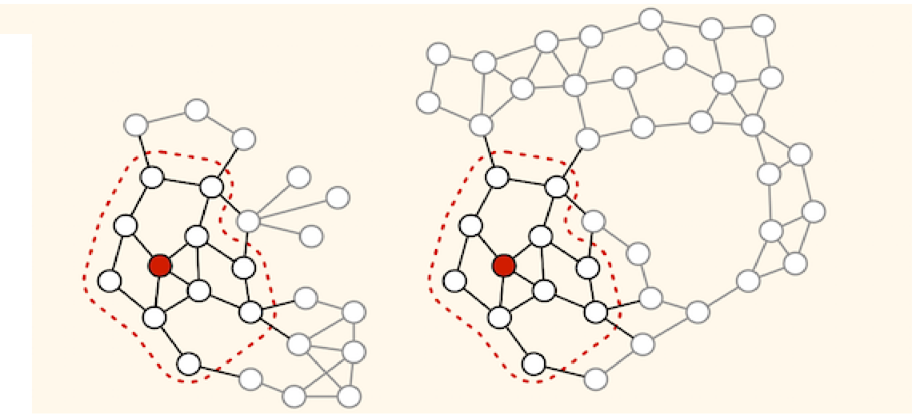
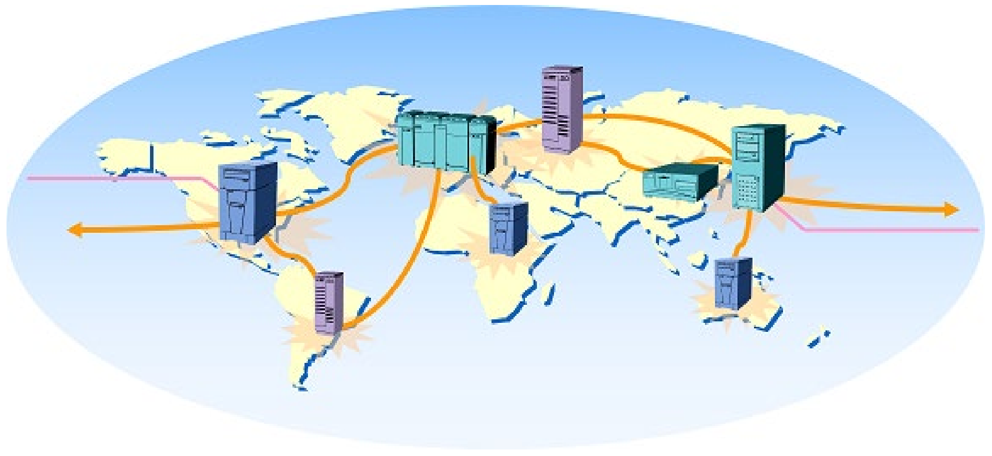
A THEORETICAL VIEW OF DISTRIBUTED SYSTEMS

Nancy Lynch, MIT EECS, CSAIL

National Science Foundation, April 1, 2021



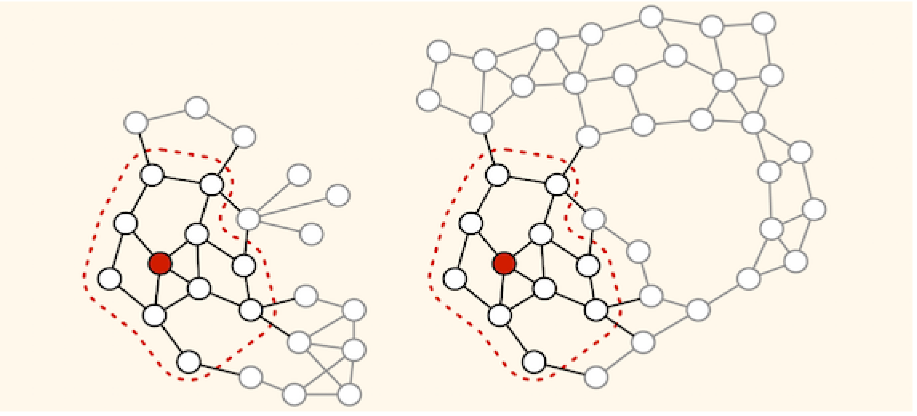


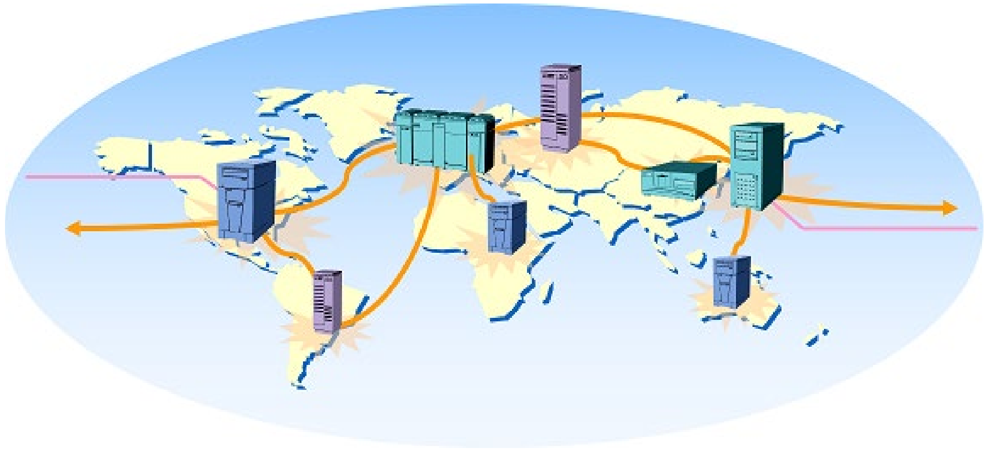
Theory for Distributed Systems

- We have worked on theory for distributed systems, trying to understand (mathematically) their capabilities and limitations.
- This has included:
  - Defining abstract, mathematical models for problems solved by distributed systems, and for the algorithms used to solve them.
  - Developing new algorithms.
  - Producing rigorous proofs, of correctness, performance, fault-tolerance.
  - Proving impossibility results and lower bounds, expressing inherent limitations of distributed systems for solving problems.
  - Developing foundations for modeling, analyzing distributed systems.
- Kinds of systems:
  - Distributed data-management systems.
  - Wired, wireless communication systems.
  - Biological systems: Insect colonies, developmental biology, brains.

This talk:

1. Algorithms for Traditional Distributed Systems
2. Impossibility Results
3. Foundations
4. Algorithms for New Distributed Systems





1. Algorithms for Traditional Distributed Systems

- Mutual exclusion in shared-memory systems, resource allocation: Fischer, Burns,…late 70s and early 80s.

- Dolev, Lynch, Pinter, Stark, Weihl. *Reaching approximate agreement in the presence of faults.* JACM,1986.
- Lundelius, Lynch. *A new fault-tolerant algorithm for clock synchronization.* Information and Computation,1988.

- **Dwork, Lynch, Stockmeyer. *Consensus in the presence of partial synchrony.* JACM,1988. Dijkstra Prize, 2007.**

1A. Dwork, Lynch, Stockmeyer [DLS]

"This paper introduces a number of practically motivated partial synchrony models that lie between the completely synchronous and the completely asynchronous models and in which consensus is solvable.

It gave practitioners the right tool for building fault-tolerant systems and contributed to the understanding that safety can be maintained at all times, despite the impossibility of consensus, and progress is facilitated during periods of stability.

These are the pillars on which every fault-tolerant system has been built for two decades. This includes academic projects, as well as real-life data centers, such as the Google file system."

Distributed consensus

- Processors in a distributed network must agree on a value in some set V.
- Each processor starts with an initial value in V, and they must agree on a value in V.
- But some of the processors might be faulty (stopping, or Byzantine).
- Agreement: All non-faulty processors agree.
- Validity: If all processors have the same initial value v, then v is the only allowed decision for a non-faulty processor.
- Problem arose originally as:
  - The Database Commit problem [Gray 78].
  - The Byzantine Agreement problem (for aircraft altimeter readings) [Pease, Shostak, Lamport 80].

[DLS] contributions

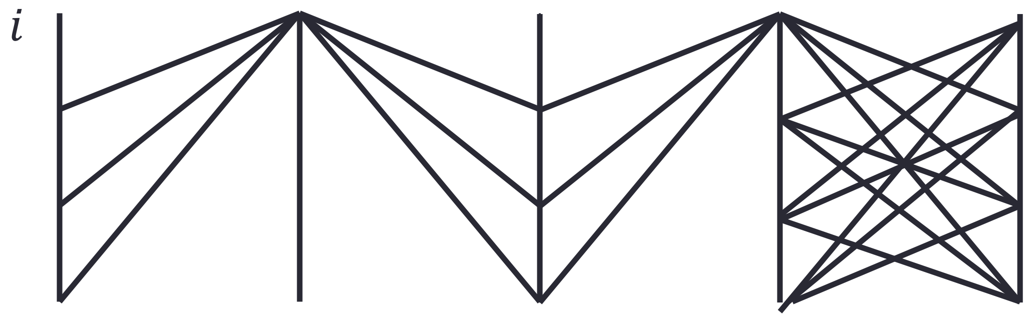
- Considers a variety of partial synchrony models, with different processor rate and message-delay assumptions.
- Considers a variety of failure models: stopping failures, Byzantine failures, Byzantine failure with authentication, sending and receiving omission failures,…
- Gives algorithms to reach agreement in all cases, guaranteeing agreement and validity always, and termination if/when the system's behavior stabilizes.
- Key algorithmic ideas:
  - Different processors try to take charge of reaching agreement.
  - Rotating coordinator.
  - Must reconcile to avoid contradictory decisions.

[DLS] contributions

- E.g., consider synchronous rounds, stopping failures.
- Messages may be lost, but after some Global Stabilization Time, all messages between non-faulty processors are delivered.
- 4-round phases, coordinator $p_i,\ i = k \bmod n$, at phase k.
- A processor may lock a value v with phase number k, meaning that it thinks that the coordinator might decide v at phase k.
- Phase k, coordinator p_i:
  - Round 1: Each processor p_j sends "acceptable" decision values (known to be someone's initial value, p_j doesn't hold a lock on a different value) to p_i; p_i tries to find a value v to propose, acceptable to a majority of processors.
  - Round 2: p_i broadcasts proposed value v, recipients lock (v, k).
  - Round 3: Those who locked (v, k) send acks to p_i; if p_i receives a majority of acks, decides v.
  - Round 4: Cleanup, exchange lock information, release older locks.

[DLS] contributions

- Phase k, coordinator p_i:
  - Round 1: Send acceptable decision values to p_i; p_i tries to pick a value v to propose, acceptable to a majority of processors.
  - Round 2: p_i broadcasts proposed value v, recipients lock (v, k).
  - Round 3: Those who locked (v, k) send acks to p_i; if p_i receives majority of acks, decides v.
  - Round 4: Cleanup, exchange lock information, release older locks.
- Some ideas inspired by [Skeen 3-phase commit].
- [Paxos] consensus protocol uses similar ideas.



Part 1: Algorithms for Traditional Distributed Systems

1A: Consensus with partial synchrony [DLS]

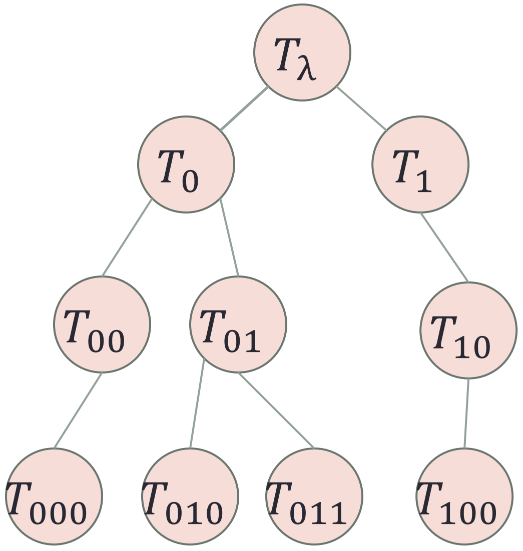
1B: Concurrency control algorithms for nested transactions

1C: Distributed shared memory

1D: Reconfigurable atomic memory

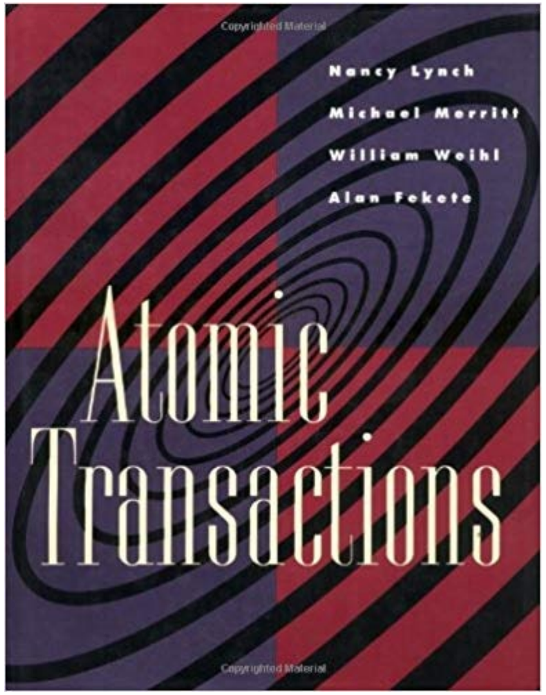
1B. Concurrency Control Algorithms for Nested Transactions

- **Lynch, Merritt, Weihl, Fekete. *Atomic transactions in concurrent/distributed Systems.* Morgan Kaufmann, 1993.**
- Background:
  - Transactions, concurrency control: [Gray], [Bernstein, Goodman].
  - Extensions to nested transactions: [Liskov]
  - Systems papers, implementations, little theory.
- Our contributions:
  - Modeled nested transaction requirements and algorithms mathematically.
  - Generalized existing algorithms.
  - Proved correctness.
- Many papers, book.



Concurrency Control Algorithms for Nested Transactions

- General theory for nested transactions, including a general Atomicity Theorem that provides a compositional method for proving correctness of concurrency control algorithms
- Lock-based algorithms.
- Timestamp-based algorithms.
- Hybrid locking/timestamp algorithms.
- Optimistic concurrency control algorithms.
- Orphan management algorithms.
- Replicated data algorithms.
- All rigorously, in terms of our I/O automata modeling framework.

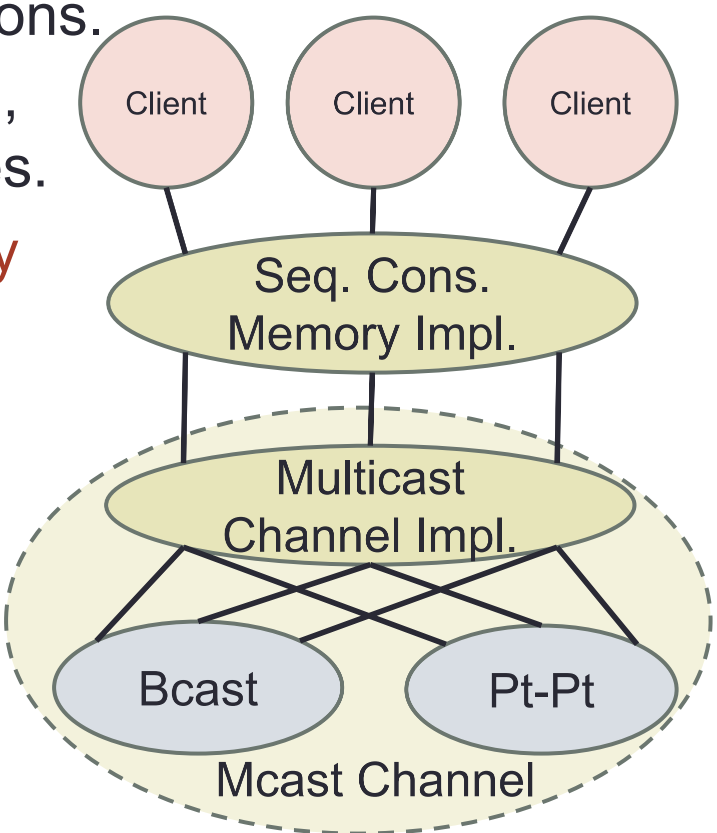


1C. Distributed Shared Memory

- **Fekete, Kaashoek, Lynch.** ***Implementing sequentially consistent shared objects using broadcast and point-to-point communication.*** **ICDCS, 1995. JACM, 1998.**

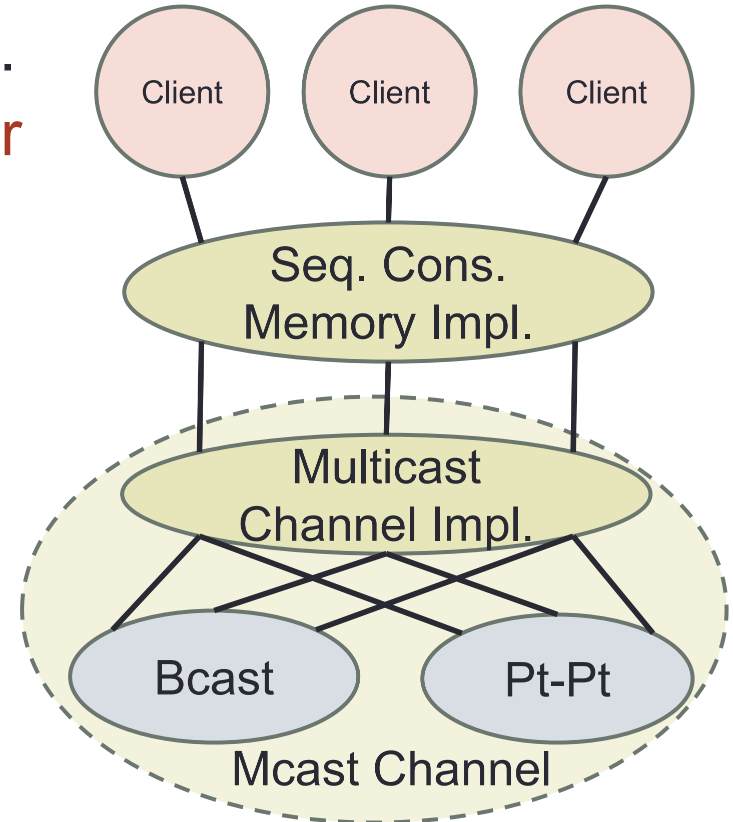
[Fekete, Kaashoek, Lynch, 1988]

- Considered an algorithm to implement sequentially consistent read/update shared memory, using basic broadcast and point-to-point communication services, as in Amoeba.
- Based on the Orca system [Bal, Kaashoek,Tanenbaum 93], for writing applications for clusters of workstations.
- Orca defines an abstract Multicast Channel, with strong ordering and causality properties.
- Implements sequentially consistent memory over any Multicast Channel, using a partial replication strategy (read any copy, update all copies).
- Implements this Channel over basic bcast and point-to-point communication services, using a sequence-number-based protocol.



[FKL] contributions

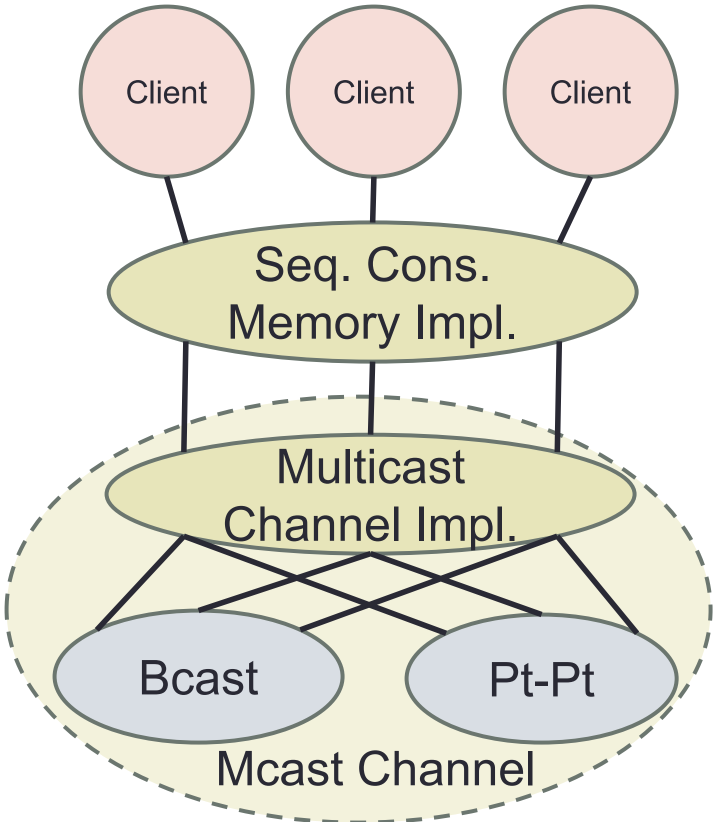
- We specified the message delivery and ordering requirements of the Multicast Channel mathematically.
- Defined a sequence-number-based algorithm to implement the Multicast Channel over the basic comm services.
- We tried to prove the algorithm correct.
- But, we discovered an algorithmic error in the Orca implementation:
  - Didn't piggyback certain needed sequence numbers on certain response messages!
- Error was fixed, in the algorithm and in the actual system.
- Then we completed the proof.



[FKL] contributions

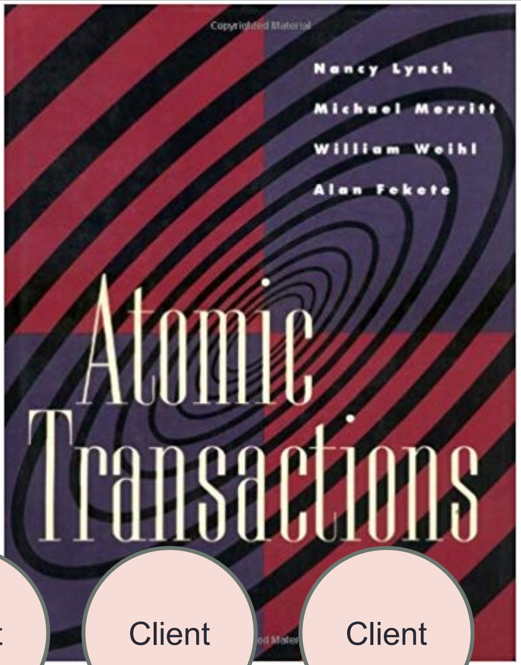
- We defined a sequence-number-based algorithm to implement the Multicast Channel over the basic communication services, proved it correct.

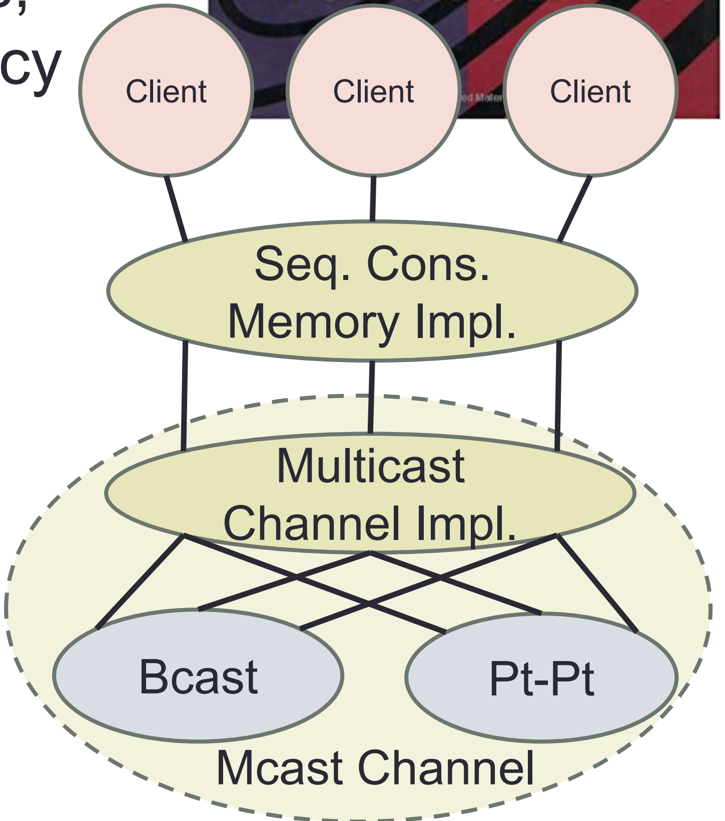
- We defined a partial-replication algorithm to implement sequentially consistent memory over the Multicast Channel, generalizing the Orca algorithm.
- Developed a new proof technique for proving sequential consistency.
- Using I/O automata, composition, abstraction.



Summary: System modeling and proofs

- Using I/O-automata-based formal methods, we modeled and verified many distributed data-management systems, especially those with strong consistency requirements.
- Specified required properties formally.
- Defined abstract versions of system algorithms.
- Clarified ambiguities.
- Proved the algorithms correct.
- Found and fixed some errors.





1D. Reconfigurable Atomic Memory

- Lynch, Shvartsman. *RAMBO: A reconfigurable atomic memory service for dynamic networks.* DISC, 2002.
- Gilbert, Lynch, Shvartsman. *RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks.* DSN, 2003.

- **Gilbert, Lynch, Shvartsman. *RAMBO: A robust, reconfigurable atomic memory service for dynamic networks.* Distributed Computing, 2010.**

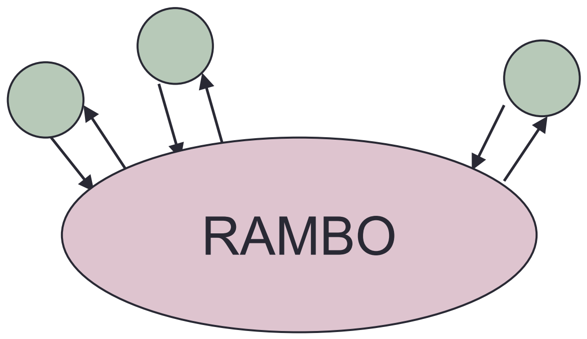
Goal

- Implement atomic Read/Write shared memory in a dynamic network setting.
  - Atomic memory “looks like” centralized shared memory.
  - Participants may join, leave, fail during computation.
  - Mobile networks, peer-to-peer networks.
- High availability, low latency.
- Atomicity in spite of asynchrony and change.
- Good performance under limits on asynchrony and change.
- Applications:
  - Soldiers in a military operation.
  - First responders in a natural disaster.

Atomic Memory in Static Networks [Attiya, Bar-Noy, Dolev 95]

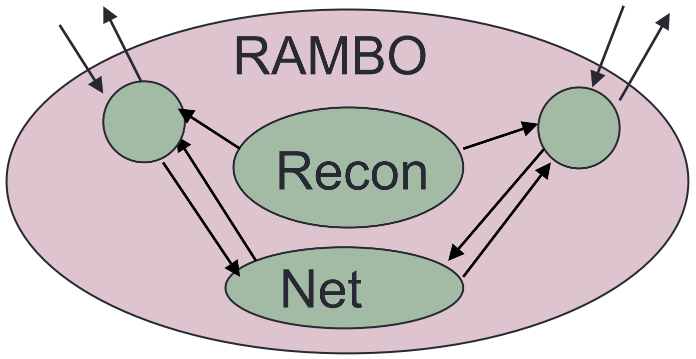
- Read-quorums, write-quorums of processors; every read-quorum intersects every write-quorum.
- Replicate objects everywhere, with version tags.
- To Read: Contact a read-quorum, determine the latest version, propagate it to a write-quorum, return it.
- To Write: Contact a read-quorum, determine the latest tag, choose a larger tag, write (tag,value) to a write-quorum.
- Operations proceed concurrently, interleaving at fine granularity; still guarantees atomicity.

RAMBO algorithm



- = Reconfigurable Atomic Memory for Basic Objects
- Uses configurations, each with:
  - members, a set of processors,
  - read-quorums, write-quorums
- Objects are replicated at all members of C.
- Reads and Writes access quorums of C, as in ABD; handles small, transient changes.
- To handle larger, more permanent changes, reconfigure to a new configuration C', moving object copies to members of C'.

Algorithm structure



- Main algorithm + Reconfiguration service
- Recon service:
  - Supplies a consistent sequence of configurations.
  - Triggered by external reconfiguration requests.
- Main algorithm:
  - Handles reading and writing of objects.
  - Removes old configurations, in the background.
  - Reads/Writes use all currently-active configurations.

Reads and Writes

- 2-phase protocol:
- Phase 1: Collect object info (values, tags) from read-quorums of all known active configurations.
- Phase 2: Propagate latest object info to write-quorums of all known active configurations.

- Many Read/Write operations may execute concurrently.
- Quorum intersection properties guarantee atomicity.
- Each phase terminates by a fixed-point condition, involving a quorum from each known active configuration.

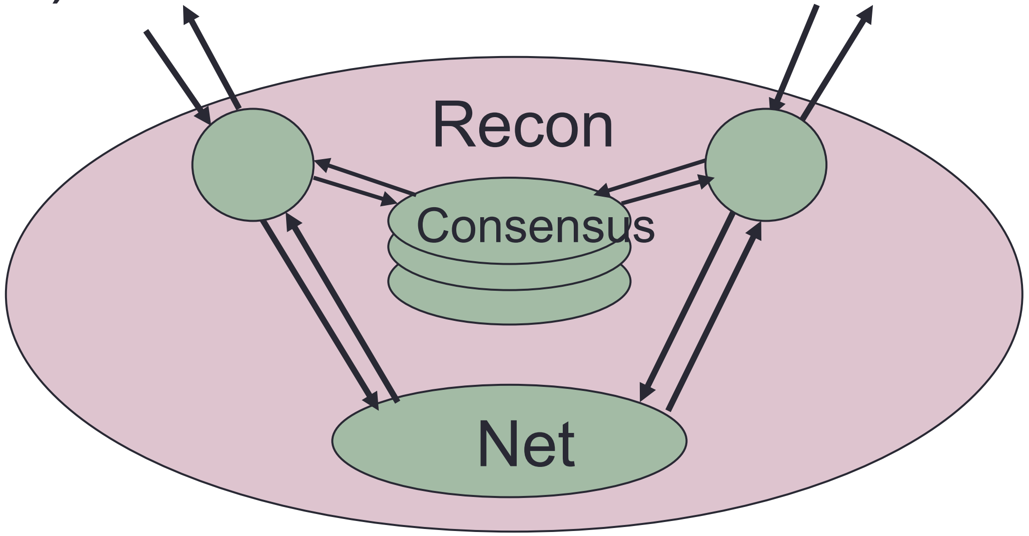
Removing old configurations

- "Garbage-collect" them in the background.
- Another two-phase protocol:
- Phase 1: For each old configuration C:
  - Inform a write-quorum of C about the new configuration.
  - Collect object information from a read-quorum of C.
- Phase 2:
  - Propagate latest object information to a write-quorum of the new configuration.

- GC proceeds concurrently with Reads and Writes, interleaving at fine granularity; still guarantees atomicity.

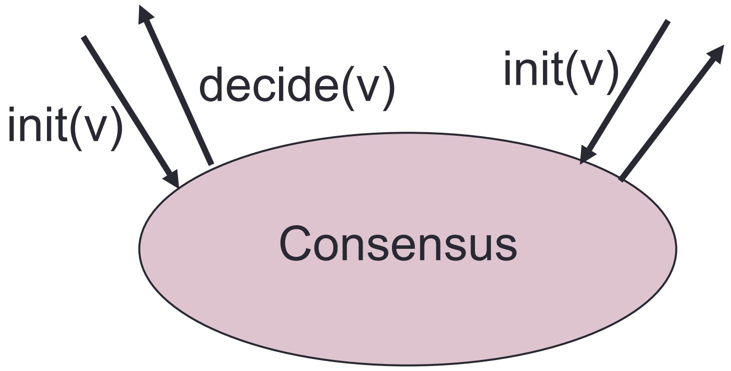
Implementing Recon

- Uses consensus to determine successive configurations.
- Members of old configuration can propose new configuration.
- Proposals reconciled using consensus.
- Consensus is heavyweight, but:
  - Used only for (infrequent) reconfigurations.
  - Does not delay Read/Write operations.



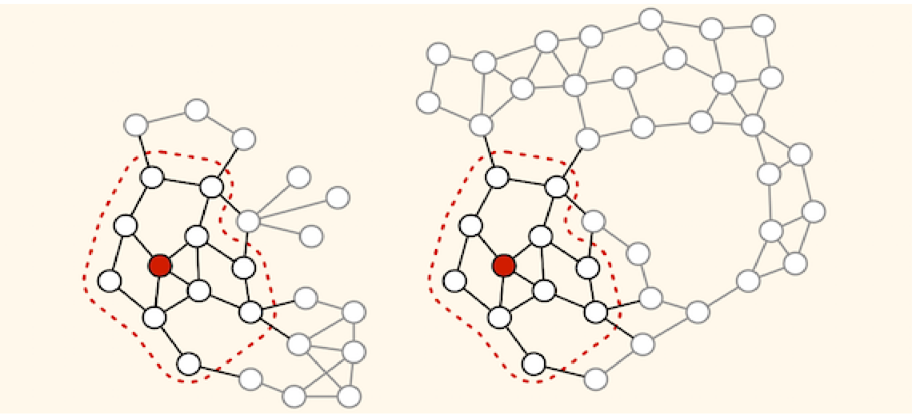
Implementing consensus

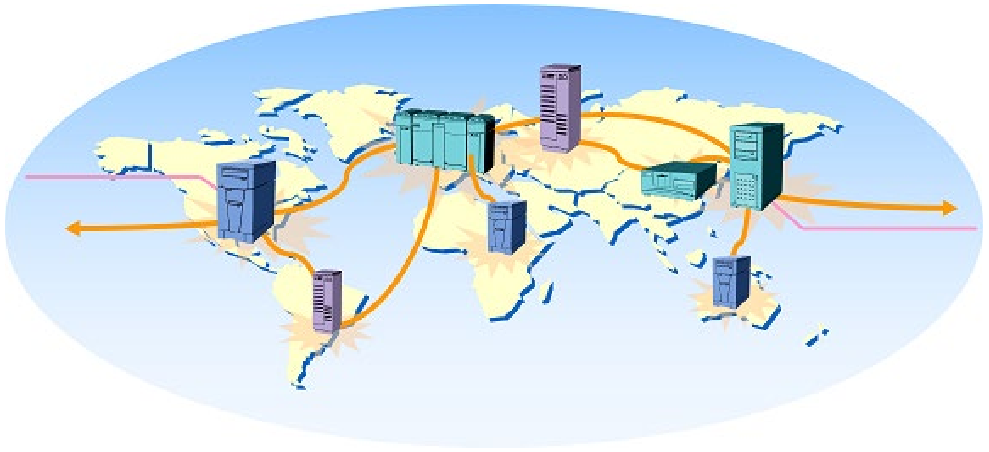
- Use DLS, or Paxos.
- Agreement, validity guaranteed always.
- Termination guaranteed when underlying system stabilizes.
- Models and proofs:
  - Using Timed I/O Automata.
  - Partial-order method for proving atomicity.



This talk:

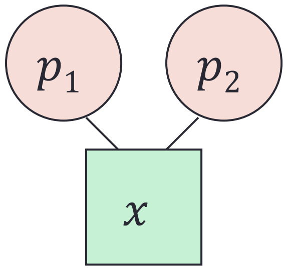
1. Algorithms for Traditional Distributed Systems
2. Impossibility Results
3. Foundations
4. Algorithms for New Distributed Systems





Part 2: Impossibility Results

- Distributed algorithms have strong inherent limitations, because they must work in difficult settings:
    - Local knowledge only.
    - Uncertainties, about remote inputs, timing, failures.
- Theoretical models enable actual proofs of such limitations.
- [Cremers, Hibbard 76]:
    - Shared-memory, Boolean shared variables, arbitrary operations.
    - Fair Mutual Exclusion: Every process(or) that requests the resource eventually gets it.
    - Unsolvable for two processes with one Boolean shared variable.



Part 2: Impossibility Results

2A: Lower bound on number of shared variables for mutual exclusion

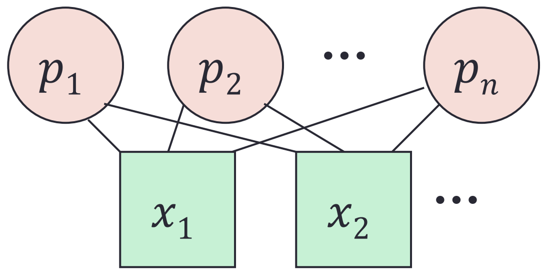
2B: Lower bound on the time to reach consensus, in synchronous systems

2C: Impossibility of distributed consensus with one faulty process, in asynchronous systems

2D: The CAP theorem

Impossibility Results for Mutual Exclusion

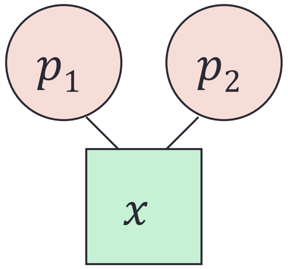
- Burns, Jackson, Lynch, Fischer, Peterson. *Data requirements for implementation of n-process mutual exclusion using a single shared variable.* JACM, 1982.



- **Burns, Lynch. *Bounds on shared memory for mutual exclusion.* Information and Computation, 1993 (actually proved in ~1980).**

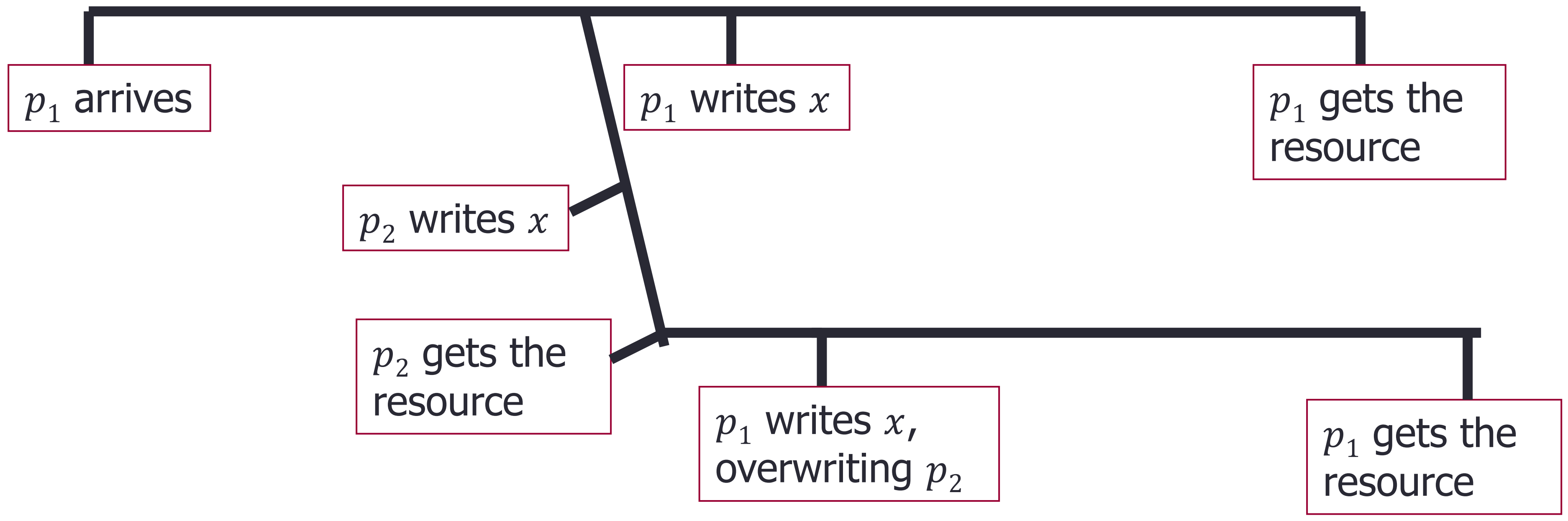
2A. Number of Shared Variables for Mutual Exclusion [Burns, Lynch 93]

- Theorem: Mutual exclusion for n processes, using read/write shared memory, requires $\geq n$ shared variables.
- Even if:
  - Fairness is not required, just progress.
  - Every process can read and write all variables.
  - Variables are of unbounded size.



- Example: $n = 2$
  - Suppose p_1 and p_2 solve mutual exclusion with progress, using one read/write shared variable x.
  - Suppose p_1 arrives, wants the resource. By the progress requirement, it must be able to get it.
  - Along the way, p_1 must write to x: If not, p_2 wouldn't know p_1 was there, so it could get the resource too, contradicting mutual exclusion.

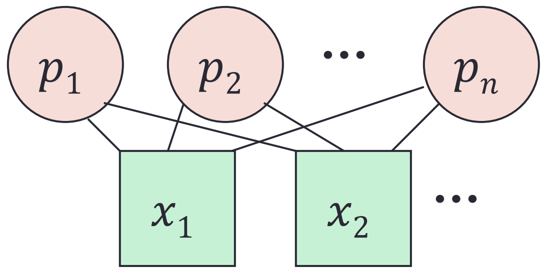
[Burns, Lynch] lower bound, $n = 2$



Contradicts mutual exclusion!

With $n > 2$ processes…

- Mutual exclusion with n processes, using read/write shared memory, requires n shared variables:



- Argument is more intricate, but uses the same key ideas:
  - Writing to a shared variable overwrites previous contents.
  - Locality: Process sees only its own state and the values it reads from shared variables.

Impossibility Results: Consensus

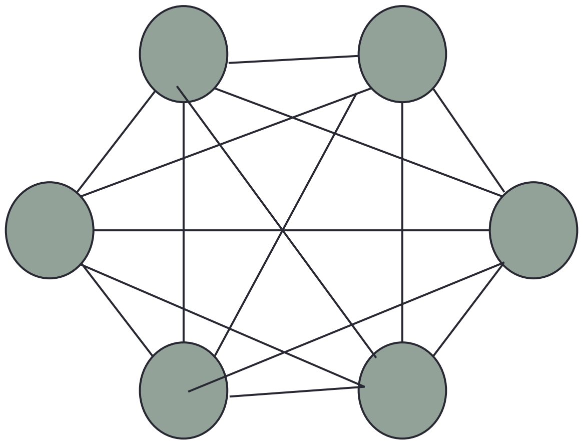
- **Fischer, Lynch. *A lower bound for the time to assure interactive consistency.* IPL, 1982.**
- Chaudhuri, Herlihy, Lynch, Tuttle. *Tight bounds for k-set agreement.* JACM, 2000.
- Fischer, Lynch, Merritt. *Easy impossibility proofs for distributed consensus problems.* Dist. Comp., 1986.
- **Fischer, Lynch, Paterson: *Impossibility of distributed consensus with one faulty process.* PODS, 1983; JACM, 1985.**

Distributed consensus

- Processes in a distributed network want to agree on a value in some set V.
- Each process starts with an initial value in V, and they want to agree on a value in V.
- Some processes might be faulty (stopping, or Byzantine).
- Agreement: All non-faulty processes agree.
- Validity: If all processes have the same initial value v, then v is the only allowed decision for a non-faulty process.

2B. Time to Reach Consensus in Synchronous Systems [Fischer, Lynch 82]

- All known algorithms had used $\geq f + 1$ rounds to reach consensus in the presence of up to f faulty processes.
- This is inherent: $f + 1$ rounds are needed in the worst case, even for stopping failures.
- Proof idea: Assume an f-round agreement algorithm tolerating f faults, get a contradiction.
- Assume:
  - n-node complete graph:
  - Binary decisions, $V = \{0,1\}$
  - Decisions right after round f.
  - All-to-all communication at every round.

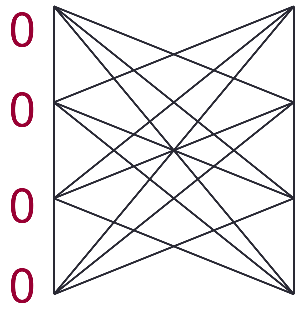


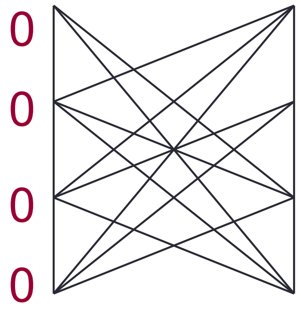
Special case: $f = 1$

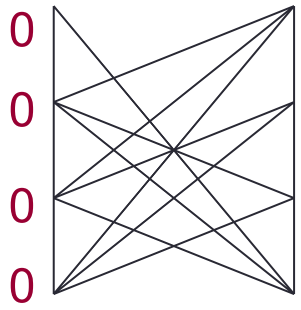
- Theorem 1: There is no n-process 1-fault stopping agreement algorithm in which non-faulty processes always decide at the end of round 1.
- Proof:
- By contradiction. Suppose there is such an algorithm.
- Construct a chain of executions, each with ≤ 1 failure, where:
  - First execution must have (unique) decision value 0.
  - Last must have decision value 1.
  - Any two consecutive executions are indistinguishable to some process i that is non-faulty in both. So i must decide the same in both executions, and the two executions must have the same decision values.
- So the decision values in the first and last executions must be the same, contradiction.

Lower bound proof, $f = 1$

- α_0: All inputs 0, no failures.
- …
- α_k: All inputs 1, no failures.
- Start chain from α_0.
- Execution α_1 removes message $1 \rightarrow 2$.
  - α_0 and α_1 indistinguishable to all except p_1 and p_2, hence to some non-faulty process.
- Execution α_2, removes message $1 \rightarrow 3$.
  - α_1 and α_2 indistinguishable to all except p_1 and p_3, hence to some non-faulty process.
- Remove message $1 \rightarrow 4$.
  - Indistinguishable to some non-faulty process.
- …





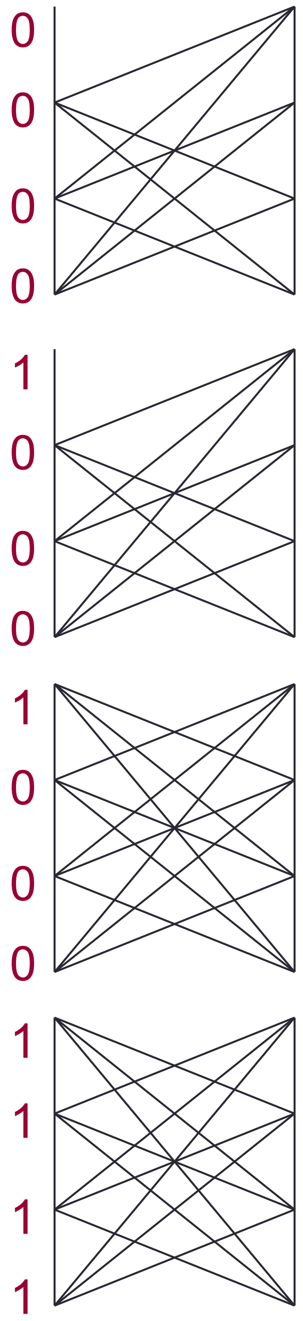


Continuing…

- Having removed all of p_1's messages, change p_1's input from 0 to 1.
  - Indistinguishable to everyone else.

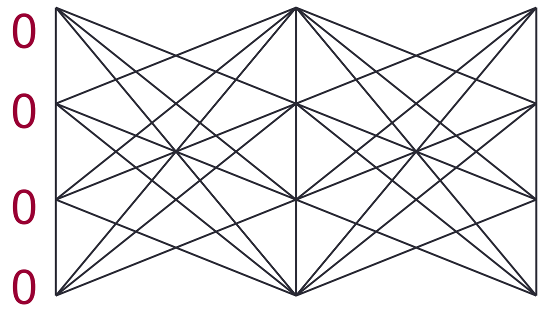
- We can't just keep removing messages, since we are allowed ≤ 1 failure in each execution.
- So, we first replace missing messages (one at a time), until p_1 is no longer faulty.

- Repeat with p_2, p_3, ..., eventually reach execution with all inputs 1, no failures.
- Yields the needed chain.



Lower bound proof, $f = 2$

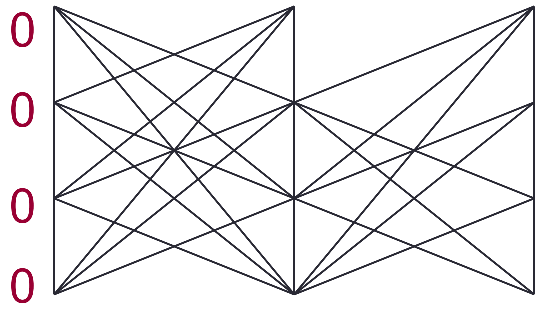
- Theorem 2: There is no n-process 2-fault stopping agreement algorithm in which non-faulty processes always decide at the end of round 2.
- Proof: Suppose there is.
- Construct a chain of executions, each with ≤ 2 failures.
- α_0: All inputs 0, no failures.
- α_k: All inputs 1, no failures.
- Each consecutive pair indistinguishable to some non-faulty process.



- E.g., consider how to change p_1's initial value from 0 to 1.

Lower bound proof, $f = 2$

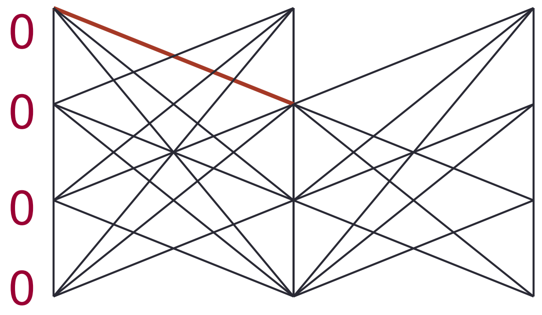
- Start with α_0, work toward killing p_1 at the start, to change its initial value, by removing its messages, one by one.
- Then work toward replacing the messages, one by one.
- Start by removing p_1's round 2 messages, one by one.



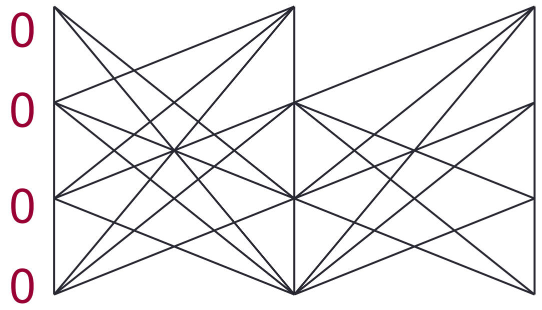
- Can't continue by removing p_1's round 1 messages, since then consecutive executions would not look the same to anyone, e.g., removing $1 \rightarrow 2$ at round 1 allows p_2 to tell everyone about the failure, at round 2.

Lower bound proof, $f = 2$

- Removing $1 \rightarrow 2$ at round 1 lets p_2 tell everyone about the failure:



vs.



- So, use several steps to remove the round 1 message $1 \rightarrow 2$
- In these steps, both p_1 and p_2 are faulty.
- Remove all of p_2's round 2 messages, one by one, replace them one by one.

- Repeat for all of p_1's round 1 messages.
- Then change p_1's initial value from 0 to 1, as needed.

2C. Consensus in Asynchronous Systems [Fischer, Lynch, Paterson 83 (FLP)]

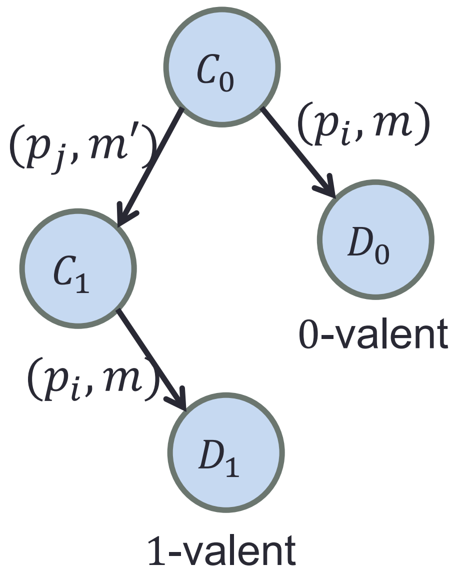
- Theorem: In an asynchronous distributed system in which at most one process may stop without warning, it is impossible for the non-faulty processes to reach agreement reliably.
- Impossibility holds even for very limited failures:
  - At most one process ever fails.
  - Failed process simply stops.
- Result may seem counter-intuitive:
  - If there are many processes, and at most one can fail, then all but one should be able to agree, and later tell the remaining one.
  - But this doesn't work!

[FLP] impossibility proof

- By contradiction: Assume a 1-fault-tolerant asynchronous algorithm that solves consensus, argue based on just the problem requirements that this cannot work.
- Assume $V = \{0,1\}$.
- Execution: A sequence of steps; in one step, one process receives one message, updates its state, and sends a finite number of messages.
- Assume every sent message eventually gets delivered.
- Execution produces a sequence of (global) configurations.
- Notice that:
  - In an execution in which all processes start with 0, the only allowed decision is 0.
  - If all processes start with 1, the only allowed decision is 1.
  - For “mixed inputs”, either decision is OK.

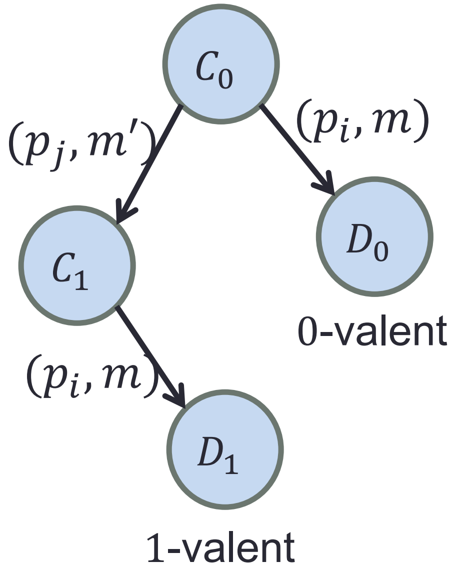
[FLP] impossibility proof

- Prove that the algorithm must yield a pattern of four configurations, C_0, C_1, D_0, D_1, where:
  - D_0 follows from C_0 in one step, in which a particular process p_i receives a particular message m.
  - D_1 follows from C_1 in one step, in which the same process p_i receives the same message m.
  - C_1 follows from C_0 in one step, in which a process p_j receives a message m'.
  - From D_0, only decision 0 is possible: D_0 is 0-valent.
  - From D_1, only decision 1 is possible: D_1 is 1-valent.
- Thus, we can "localize" a decision to a particular pattern of configurations.
- For if not, then we could make the algorithm execute forever, with all processes continuing to take steps, and no one ever deciding.
- Which contradicts the termination requirement.



[FLP] impossibility proof

- Now get a contradiction by considering two cases:
- Case 1: $i \neq j$
  - Then consider delivering (p_j, m') after D_0; still 0-valent.
  - So delivering (p_i, m) then (p_j, m') yields 0-valence, but delivering (p_j, m') then (p_i, m) yields 1-valence.
  - But the two steps occur at different processes, so their relative order can't matter.
  - Contradiction.
- Case 2: $i = j$
  - Then consider any deciding execution from C_0 in which p_i fails, but everyone else keeps taking steps.
  - Applying the same execution from D_0 must lead to a decision of 0.
  - Applying the same execution from D_1 must lead to a decision of 1.
  - But the other processes can't distinguish these cases!
  - Contradiction.

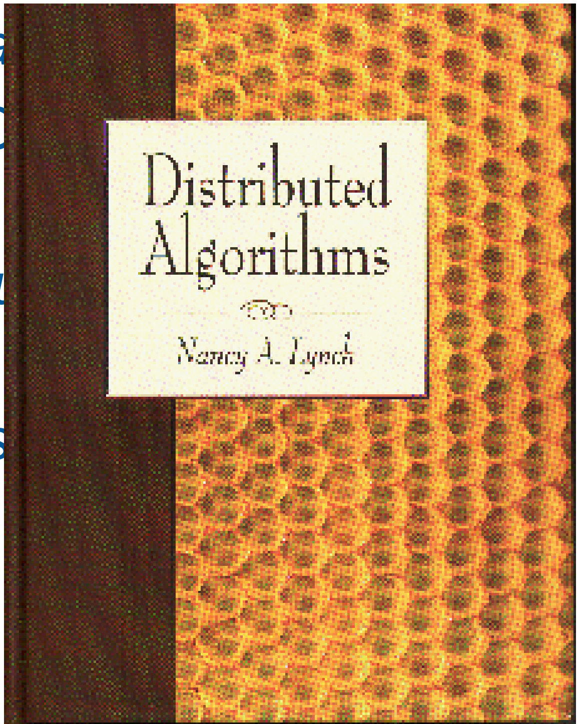


Significance for distributed systems

- Consensus is an important problem in practice, for example, for distributed database commit.
- [FLP] result shows limitations on the kind of algorithm one could hope to find, for agreement problems.
- To get around the impossibility result, one can:
  - Use random choices [Ben-Or, 83]
  - Rely on timing assumptions [Dolev, Dwork, Stockmeyer, 87]
  - Weaken requirements carefully [DLS 88]:
    - Agreement, validity always hold.
    - Termination required if/when system behavior "stabilizes": no new failures, and timing within "normal" bounds.

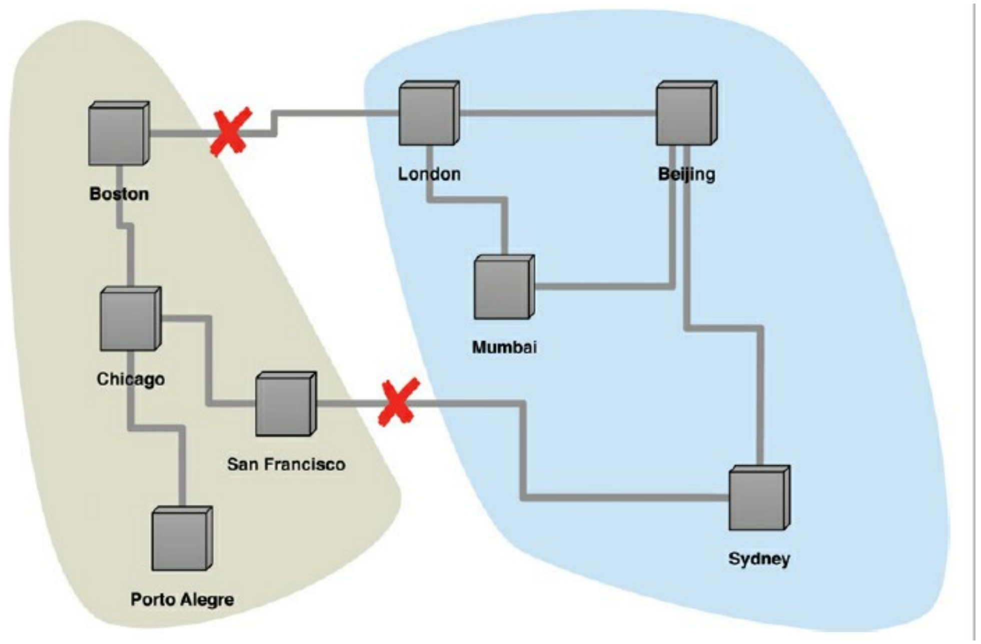
More Impossibility Results

- Lundelius, Lynch. *An upper and lower bound for clock synchronization.* Information and Control, 1984.
- Lynch, Shavit. *Timing-based mutual exclusion.* RTSS, 1992.
- Attiya, Dwork, Lynch, Stockmeyer. *Bounds on the time to reach agreement in the presence of timing uncertainty.* JACM, 1993.
- Attiya, Lynch, Shavit. *Are wait-free algorithms fa*
- Fan, Lynch. *Gradient clock synchronization.* PC Distributed Computing, 2006.
- Lynch. *A hundred impossibility proofs for distribu* PODC 1989.
- Ellen, Ruppert. *Hundreds of impossibility results computing.* Distributed Computing 2003.



2D. The CAP Theorem

- **Gilbert, Lynch.** ***Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.***
  SIGACT NEWS, 2002.

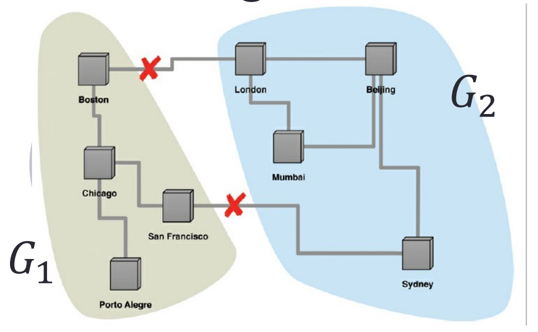


[Gilbert, Lynch]

- *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.*
- Inspired by an informal conjecture described by Brewer in a PODC 2000 keynote.
- Brewer described three desirable properties for Web services:
  - Consistency: Data should appear atomic.
  - Availability: Every request to perform an operation should eventually return some result.
  - Partition-tolerance: Tolerates lost messages.
- Brewer's claim: In general, we can't achieve all three.
- We formalized the properties; identified several different versions, some possible, some impossible.

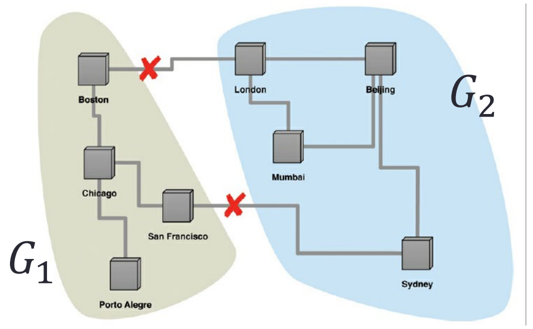
[Gilbert, Lynch]

- Consistency: Atomic Read/Write data objects.
- Availability: Read/Write requests should always return.
- Partition-tolerance: Any set of messages may be lost.
- We considered:
  - Asynchronous and partially synchronous models.
  - Whether consistency may be violated when messages are lost.
- Asynchronous case: Unbounded message delay.
- Theorem 1: It’s impossible to guarantee availability, atomicity in all executions, while allowing any set of lost messages.
- Proof idea: Partition the network into two parts, G_1 and G_2. Suppose a write occurs in G_1, then a read in G_2. Read can’t know about the write.



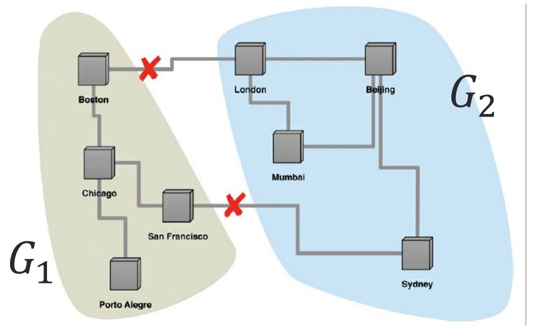
Asynchronous case, cont’d

- Theorem 1: Impossible to guarantee availability, atomicity in all executions, while allowing any set of lost messages.
- Q: What if we drop the atomicity requirement when partitions occur?
- Theorem 2: It’s impossible to guarantee availability in all executions, atomicity in executions in which no messages are lost, while allowing any set of lost messages in general.
- Proof idea: Processors don’t know whether messages have really been lost, or may arrive later.
- A violation of atomicity occurs at a finite point in time; but then we could extend the execution to deliver all messages.



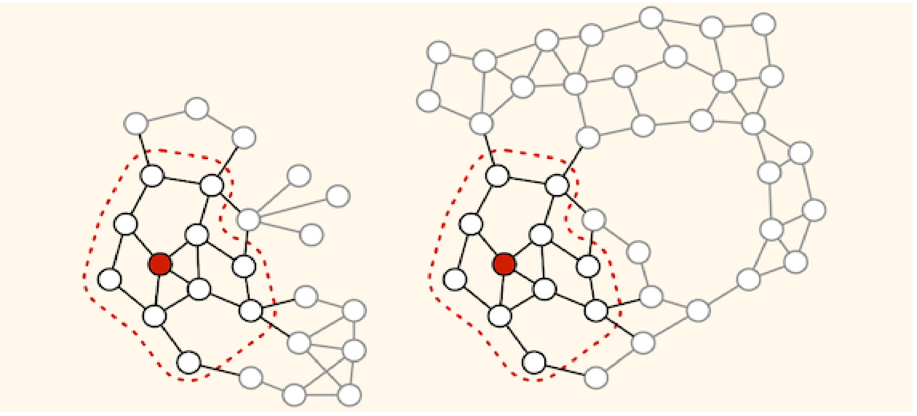
Partially synchronous case

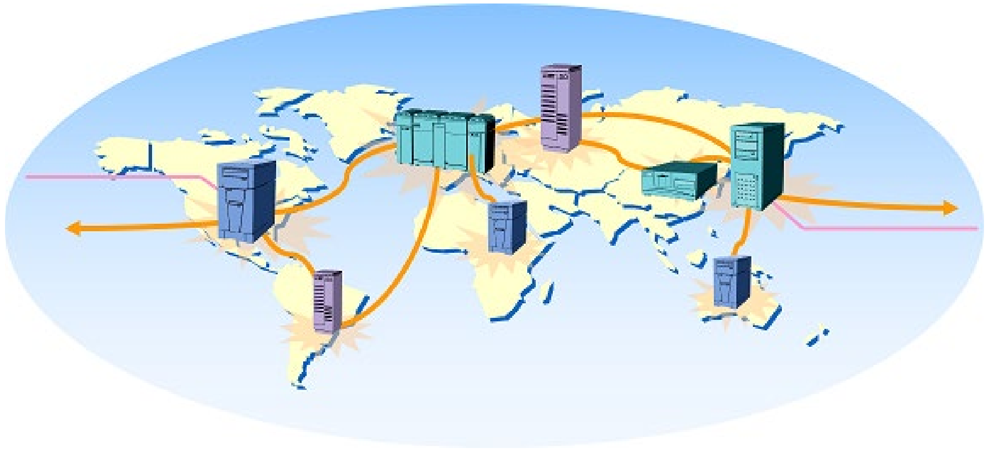
- Local timers; not synchronized, but same rate.
- Can schedule actions to occur at particular local times.
- Messages that aren't lost are delivered within a known delay.
- Theorem 3: (like Theorem 1) It's impossible to guarantee availability and atomicity in all executions, while allowing any set of lost messages.
- But now:
- Theorem 4: It's possible to guarantee availability in all executions, atomicity in executions in which no messages are lost, while allowing any set of lost messages in general.
- Proof idea: Now we can detect lost messages.



This talk:

1. Algorithms for Traditional Distributed Systems
2. Impossibility Results
3. Foundations
4. Algorithms for New Distributed Systems

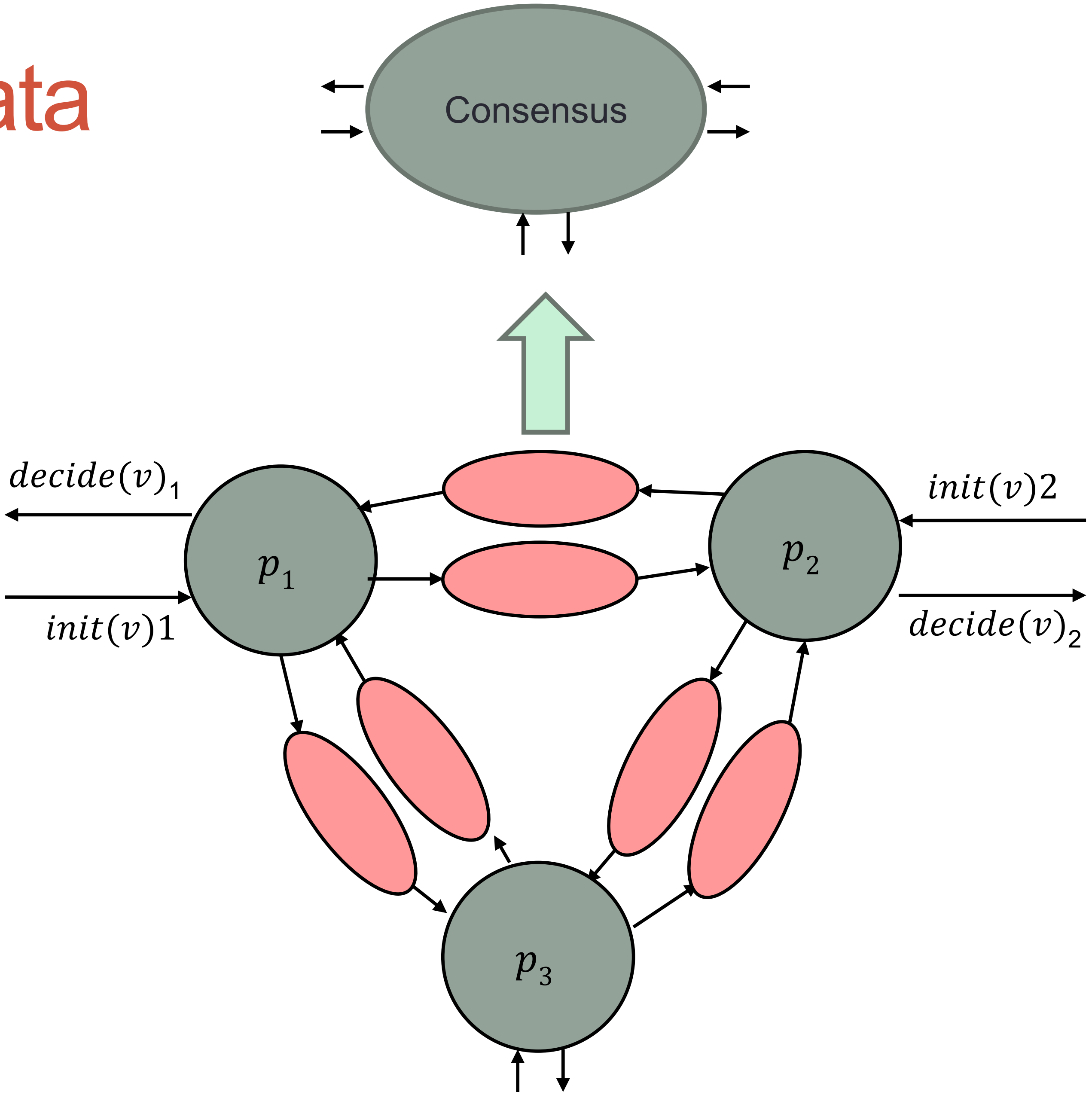




3. Foundations

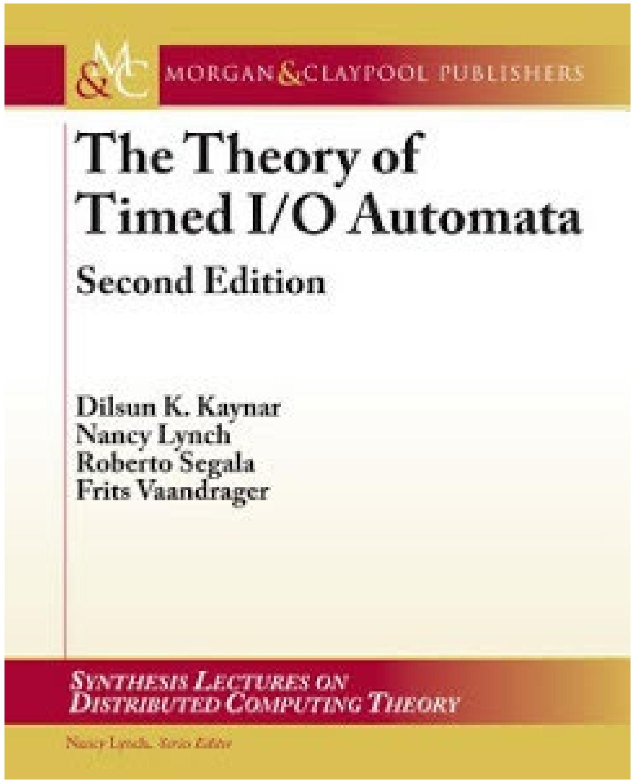
- Lynch, Fischer. *On describing the behavior and implementation of distributed systems.* Theoretical Computer Science, 1981.
- Lynch, Tuttle. *An introduction to Input/Output Automata.* CWI-Quarterly, 1989.
- Lynch,Tuttle. *Hierarchical correctness proofs for distributed algorithms.* PODC, 1987.
- Lynch, Vaandrager. *Forward and backward simulations.* Information and Computation 1995.

I/O Automata



Timed I/O Automata, Hybrid I/O Automata

- Lynch, Vaandrager. *Forward and backward simulations II: Timing-based systems.* Information and Computation 1996.
- Lynch, Segala, Vaandrager. *Hybrid I/O automata.* Information and Computation, 2003.
- Kaynar, Lynch, Segala, Vaandrager. *The theory of timed I/O automata.* Synthesis Lectures on Distributed Computing Theory 2006, 2010.

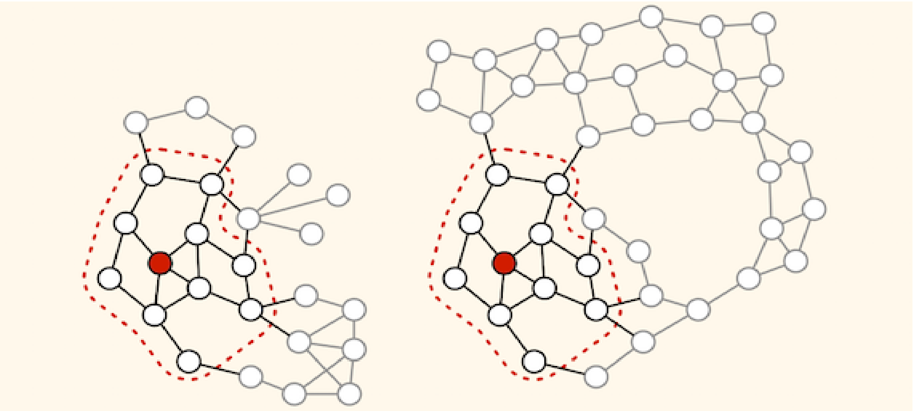

MORGAN & CLAYPOOL PUBLISHERS
The Theory of Timed I/O Automata
Second Edition
Dilsun K. Kaynar
Nancy Lynch
Roberto Segala
Frits Vaandrager
Synthesis Lectures on Distributed Computing Theory
Nancy Lynch, Series Editor

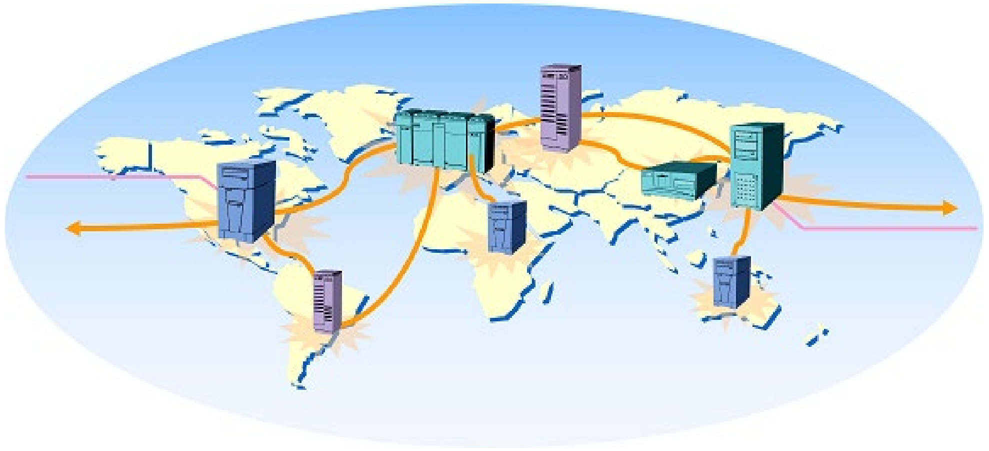
Probabilistic I/O Automata, Probabilistic Timed I/O Automata

- Lynch, Segala. *Probabilistic simulations for probabilistic processes.* CONCUR 1994. CONCUR Test-of-Time award 2020.
- Segala. *Modeling and verification of randomized distributed real-time systems.* Ph.D. Thesis, EECS, MIT, 1995.
- Lynch, Segala, Vaandrager. *Compositionality for probabilistic automata.* CONCUR, 2003.
- Lynch, Segala, Vaandrager. *Observing branching structure through probabilistic contexts.* SIAM J. Computing, 2007.

This talk:

1. Algorithms for Traditional Distributed Systems
2. Impossibility Results
3. Foundations
4. Algorithms for New Distributed Systems





4. Algorithms for New Distributed Systems

- So far, I have described work on algorithms from traditional distributed systems.
- For the past decade, we have been working on new types of distributed systems: those in which noise, uncertainty, and change predominate:
  - Wireless networks
  - Biological systems
- Same general kinds of research:
  - Abstract models for problems and algorithms.
  - New algorithms.
  - Proofs of correctness, performance,…
  - Impossibility results and lower bounds.
  - General foundations.

4A. Wireless Networks

- Ad hoc, no central base station, usually mobile.
- Soldiers, first responders, explorers,…
- Challenge: Find good abstraction layers to make it easier to develop applications for ad hoc wireless networks.

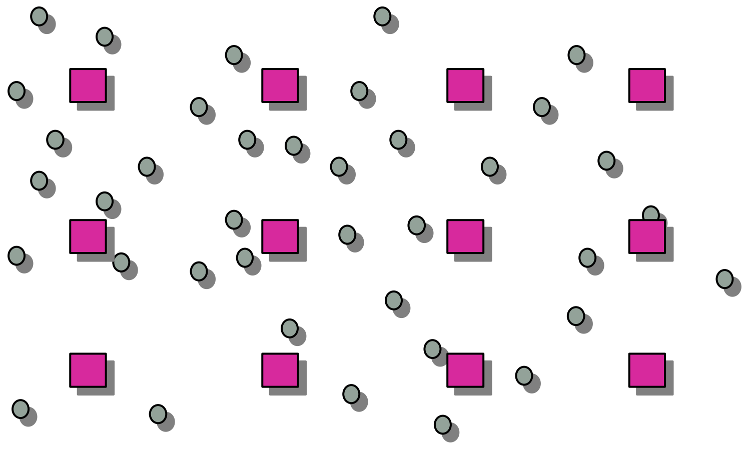
- Idea 1: Virtual Node layers
- Idea 2: Abstract MAC (Reliable Local Broadcast) layers

Idea 1: Virtual Node Layers

- Dolev, Gilbert, Lynch, Shvartsman, Welch. *GeoQuorums: Implementing atomic memory in mobile ad hoc networks.* DISC 2003.
- Dolev, Gilbert, Lynch, Schiller, Shvartsman, Welch. *Virtual Mobile Nodes for mobile ad hoc networks.* DISC, 2004.
- Dolev, Gilbert, Lahiani, Lynch, Nolte. *Timed Virtual Stationary Automata for mobile networks.* OPODIS, 2005.
- Dolev, Lahiani, Lynch, Nolte. *Self-stabilizing mobile node location management and message routing*. SSS 2005.
- **Brown, Gilbert, Lynch, Newport, Nolte, Spindel. *The Virtual Node layer: A programming abstraction for wireless sensor networks.* SIGBED Review, 2007.**
- Gilbert, Lynch, Mitra, Nolte. *Self-stabilizing robot formations over unreliable networks.* ACM Transactions on Autonomous and Adaptive Systems, 2009.

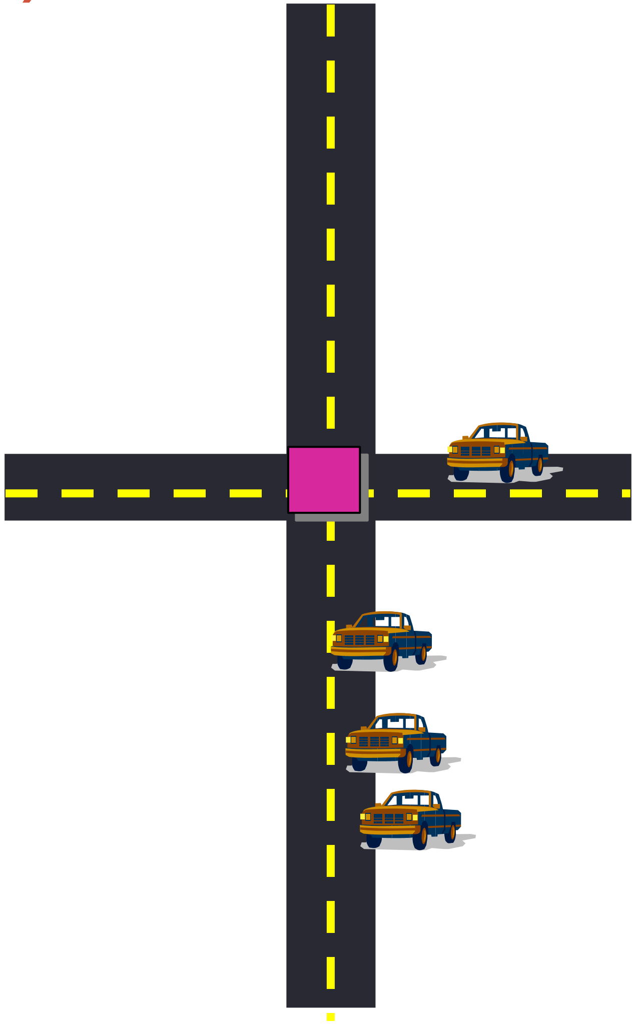
Virtual Node Layers

- Simplify programming for an ad hoc mobile network by adding Virtual Nodes (VNs) at known locations.
- Write algorithms and applications using Virtual Nodes.
- Mobile nodes emulate Virtual Nodes:
  - Each VN is emulated by nodes in its vicinity.
  - Use a full replication or leader-based strategy.
- Applications:
  - Implement atomic memory in a mobile network
  - Geographical message routing
  - Regional motion coordination: robot swarms, Virtual Traffic Lights, Virtual Air-Traffic Controllers,…



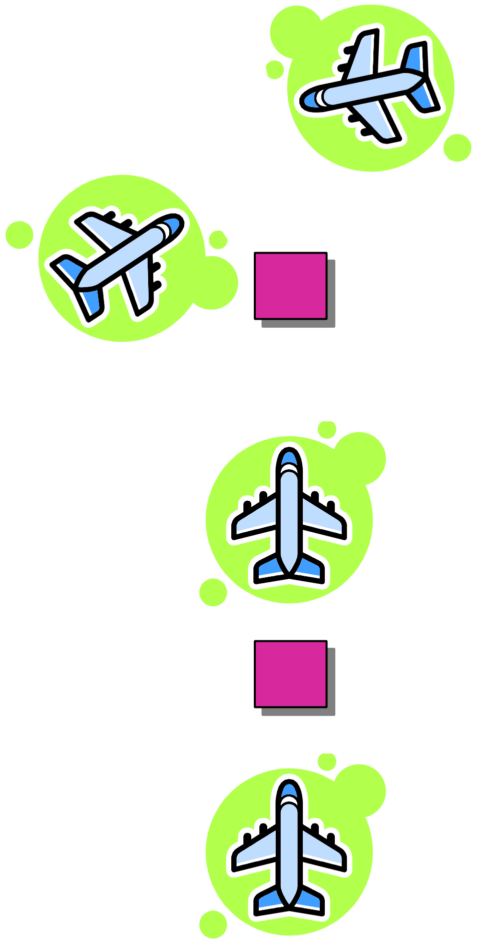
Virtual Traffic Light (VTL)

- For an intersection without a real traffic light.
- Computers in cars emulate a VN, which is programmed to act like a traffic light.
- Any policy desired, e.g., 30 sec in each direction.
- Cars see red or green, on local displays.
- VTL dies when no cars are around, but that's OK.



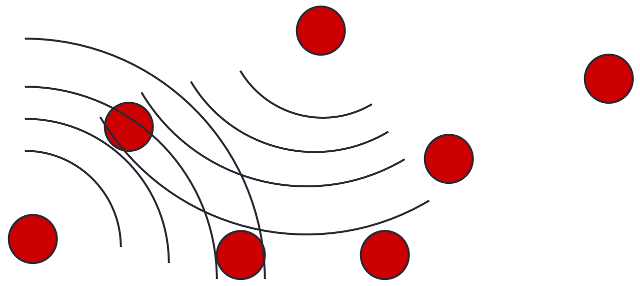
Virtual Air-Traffic Controllers

- Aircraft in regions of airspace without ground-based controllers, for example, over the ocean.
- To control access to regions, use VATCs, emulated by computers on the aircraft.
- VATC behaves like a human ATC:
  - Keeps track of aircraft in local region.
  - Tells neighbor ATCs when to hand off aircraft.
  - Tells aircraft how to move within local region.



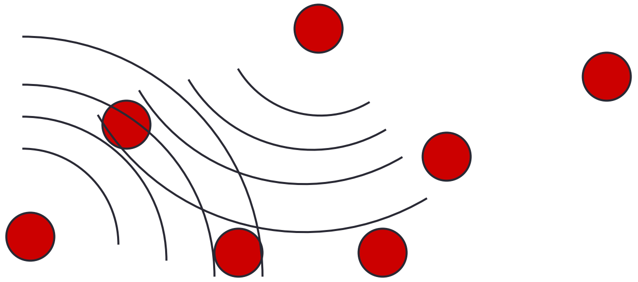
Dealing With Unreliable Communication

- Our Virtual Node work assumed a wireless communication model based on reliable local broadcast.
- But real wireless communication is not so reliable---it's subject to collisions, with resulting message losses.



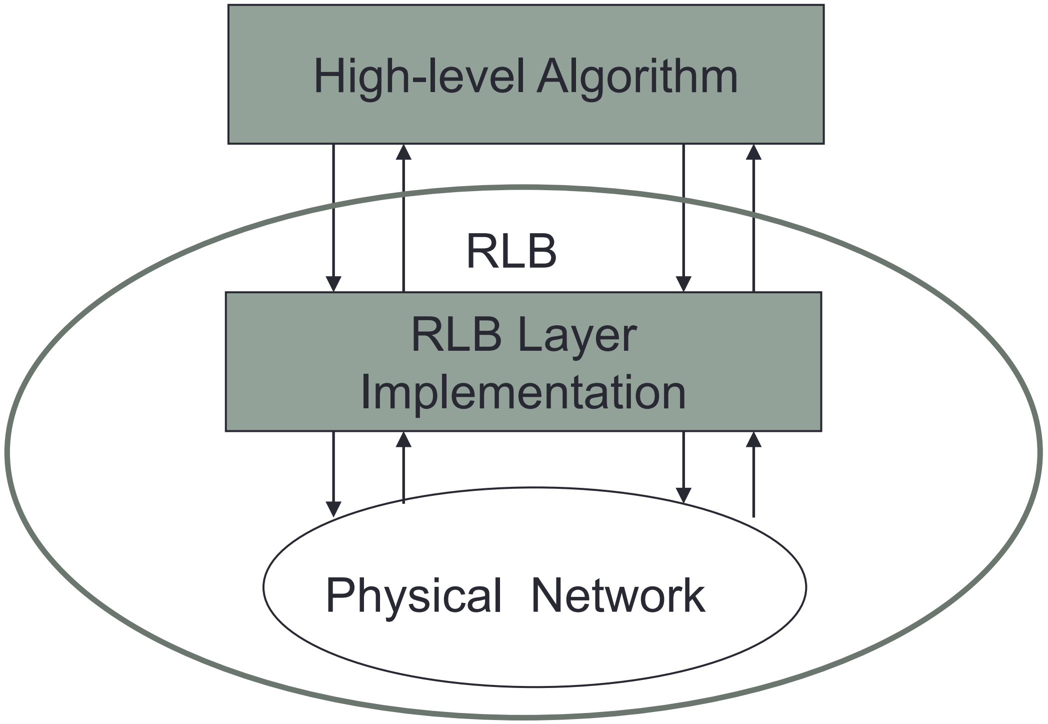
Message Collision Model

- In each round, some nodes transmit, the others listen.
- Transmitter hears only its own message.
- Listener hears:
  - Silence ($\perp$), if none of its neighbors (in an assumed communication graph G) transmits.
  - A message, if exactly one of its neighbors transmits.
  - Collision ($\top$), if two or more neighbors transmit.



Idea 2: Abstract MAC layers

- Mask collisions within an abstract MAC layer, also known as a Reliable Local Broadcast (RLB) layer.
- Implement RLB using low-level collision-management algorithms.
- Build higher-level algorithms over RLB.



Abstract MAC layers

- Kuhn, Lynch, Newport. *The Abstract MAC layer.* DISC, 2009. Dist. Comp. 2011.
- Khabbazian, Kowalski, Kuhn, Lynch. *Decomposing broadcast algorithms using Abstract MAC layers.* Ad Hoc Networks, 2014.
- Halldórsson, Holzer, Lynch. *A local broadcast layer for the SINR network model.* PODC 2015.

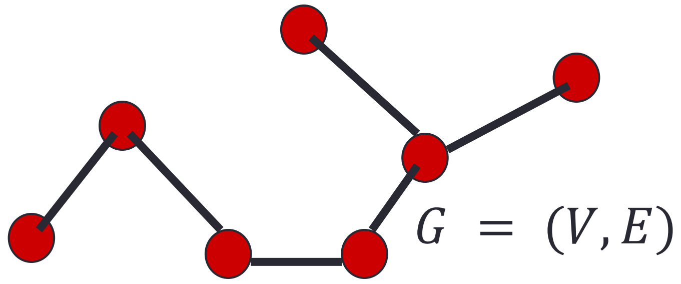
Remarks

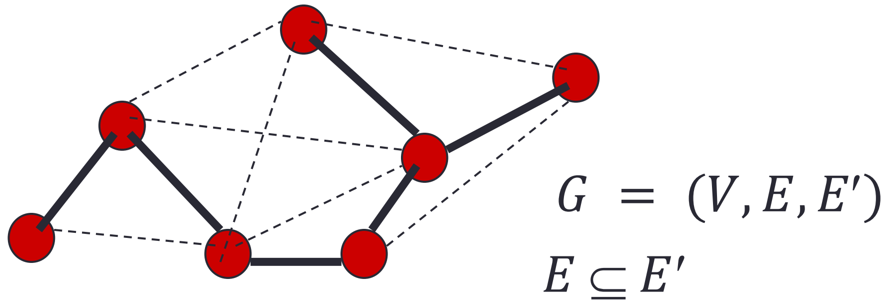
- So, we can mask message collisions inside a Reliable Local Broadcast layer.
- Use RLB as an abstraction layer for developing higher-level algorithms.

- But: This work considers message collisions, but not communication uncertainty, i.e., uncertainty in where the messages reach.

Communication Uncertainty

- Use two graphs, G and G':
  - G: Messages must reach.
  - G': Messages may reach.



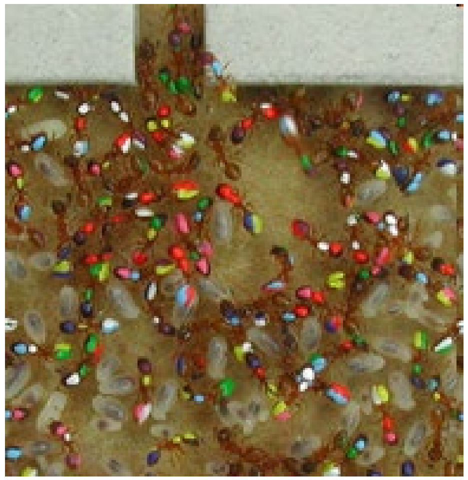


- [Clementi, Monti, Silvestri 04]
- [Kuhn, Lynch, Newport 09]

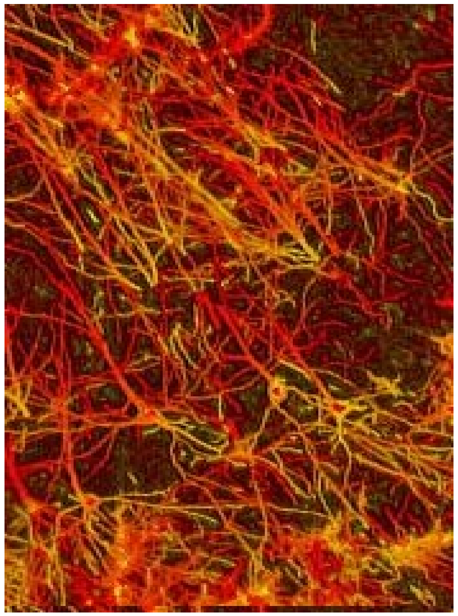
Results for the Dual Graph Model

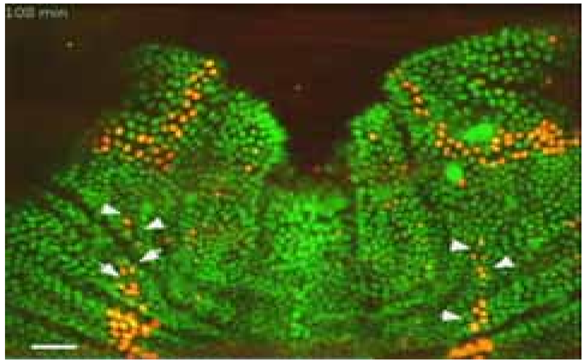
- **Kuhn, Lynch, Newport. *Hardness of broadcasting in wireless networks with unreliable communication.* PODC 2009.**
- Kuhn, Lynch, Newport, Oshman, Richa. *Broadcasting in unreliable radio networks.* PODC 2010
- Ghaffari, Haeupler, Lynch, Newport. *Bounds on contention management in radio networks.* DISC 2012
- Ghaffari, Lynch, Newport. *The cost of radio network broadcast for different models of unreliable links.* PODC 2013.
- Censor-Hillel, Gilbert, Kuhn, Lynch, Newport. *Structuring unreliable radio networks.* DISC 2014.
- Ghaffari, Kantor, Lynch, Newport. *Multi-message broadcast with abstract MAC layers and unreliable links.* PODC 2014.
- Lynch, Newport. *A (truly) local broadcast layer for unreliable radio networks.* PODC 2015.
- **Gilbert, Lynch, Newport, Pajak. *On Simple Back-Off in Unreliable Radio Networks.* OPODIS 2018, Best Paper award.**

4B. Biological Systems



- Biological distributed algorithms:
  - Insect colonies
  - Developing organisms
  - Brains
- Simple models.
- Simple, flexible, robust, adaptive algorithms.
- Two goals:
  - Help to understand biological systems.
  - Suggest new ideas for engineered algorithms.

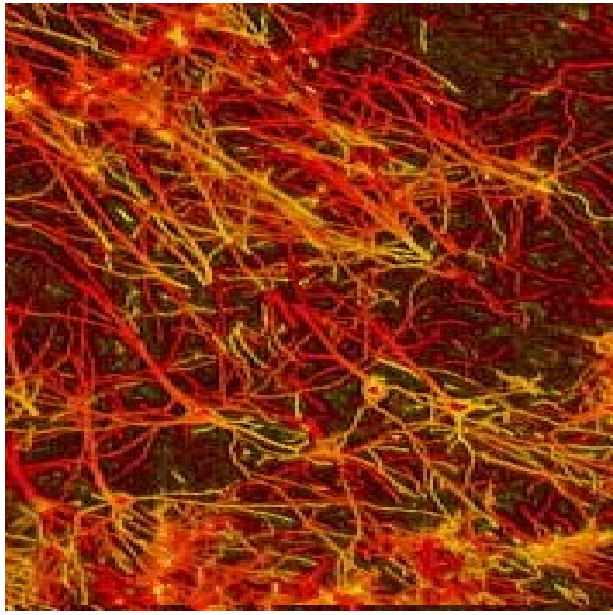




Our work: Insect colonies

- Task allocation
- Exploring for food (searching)
- Agreeing on a new nest (consensus)
- Colony density estimation
- Radeva. *A Symbiotic Perspective on Distributed Algorithms and Social Insects.* Ph.D Thesis, EECS, MIT, 2017.
- Radeva, Dornhaus, Lynch, Nagpal, Su. *Costs of task allocation with local feedback: effects of colony size and extra workers in social insects and other multi-agent systems. PLOS Computational Biology,* 2017.
- Musco, Su, Lynch. *Ant-Inspired Density Estimation via Random Walks.* arXiv:1603.02981, v2, 2019.
- Zhao, Lynch, Pratt. *The Power of Social Information in Ant-Colony House-Hunting: A Computational Modeling Approach.*

Our work: Brains



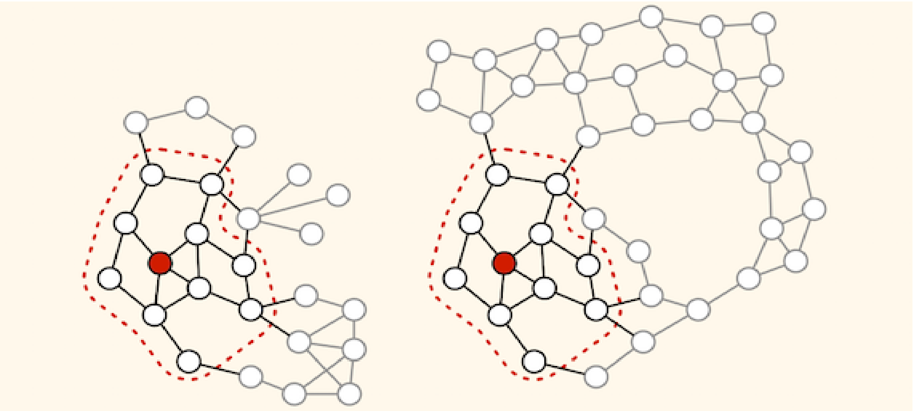
- Stochastic Spiking Neural Network model
- Winner-take-all
- Neural coding, similarity detection
- Learning structured concepts
- Lynch, Musco. *A basic compositional model for Spiking Neural Networks.* arXiv:1808.03884, 2018.
- Lynch, Musco, Parter. *Winner-Take-All computation in Spiking Neural Networks.* arXiv:1904.12591, 2019.
- Hitron, Lynch, Musco, and Parter. *Random sketching, clustering, and short-term memory in Spiking Neural Networks.* ITCS 2020.
- Lynch, Mallmann-Trenn. *Learning hierarchically-structured concepts.* arxiv:1909.04559v4, 2021.

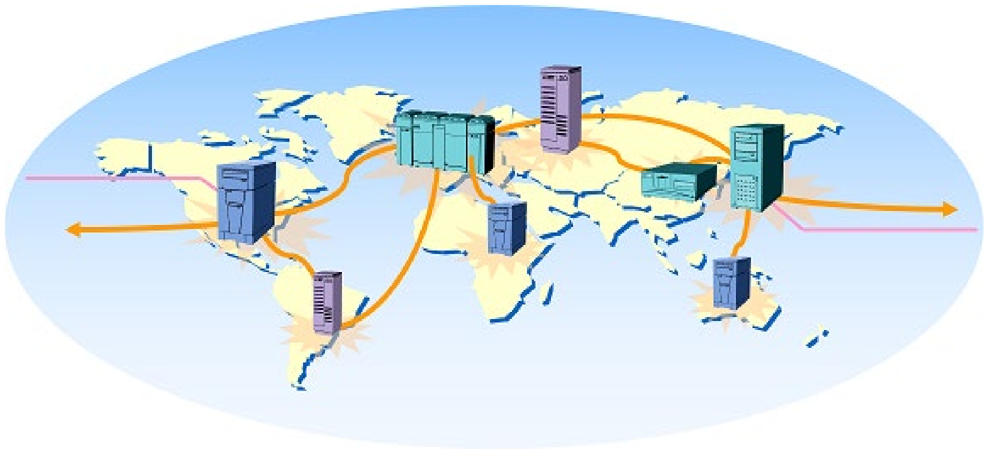
Conclusions

- We have worked on theory for distributed systems, to help understand their capabilities and limitations.
- This work has included:
  - Abstract models for systems problems and algorithms.
  - New algorithms.
  - Rigorous proofs of correctness, performance,…, discovery of errors.
  - Impossibility results and lower bounds, expressing inherent limitations.
  - General foundations for modeling, analyzing distributed systems.
- Many kinds of systems:
  - Distributed data-management systems
  - Wired, wireless communication systems
  - Biological systems: Insect colonies, brains
- But there is still much more to be done!

Thanks to my many, many collaborators!

Thank you!





Some background

- No family academic background.
- Hunter College High School, math team, enrichment activities, mentor.
- Computer programming but no computer; foreshadowed a career in theoretical CS?
- Brooklyn College, encouragement from professors.
- MIT, theory of computation, complexity theory.
- Georgia Tech, switched to distributed computing theory.
- Early papers in the area were not written as math papers; I realized I could contribute by turning the ideas into math.