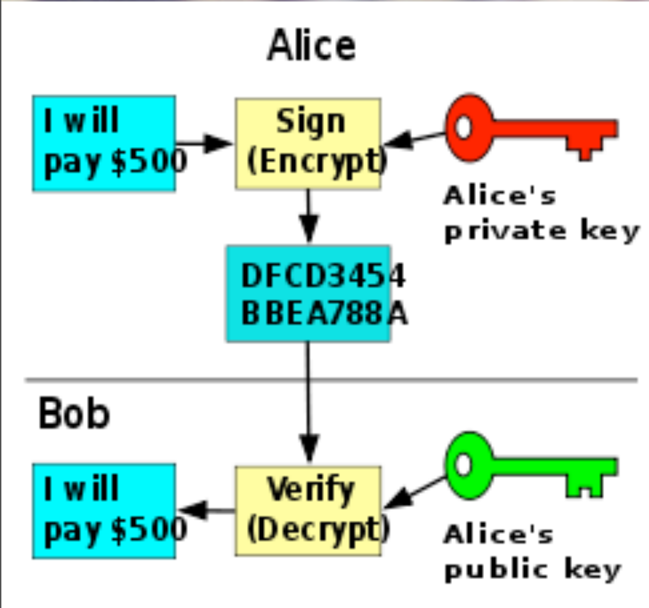
Ordering and Consistent Cuts

Isaac Sheff

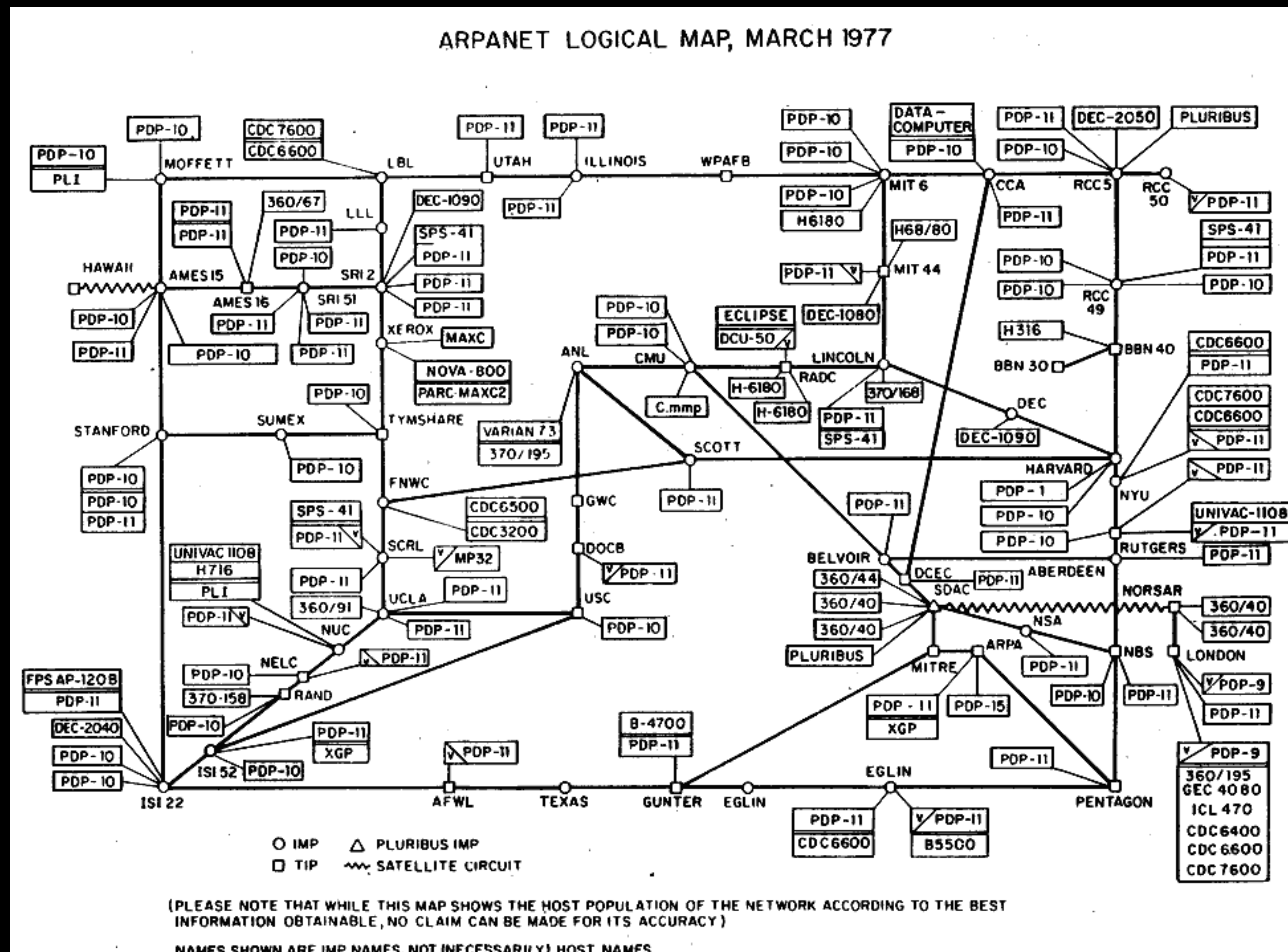
1978



Our story begins in 1978. To give you a feel for where we are in terms of music, movies, video games, politics, computer science and the internet (or ARPAnet at the time), America was just getting over a case of saturday night fever, Star Wars had just broken out, Space Invaders arrived, Christopher Reeve was Superman, Jimmy Carter was President, and RSA and BBS were invented.

Distributed Systems Theory

- Lots of models
- Part reality-driven
- Part abstract questions
    - Byzantine Generals
    - Dining Philosophers



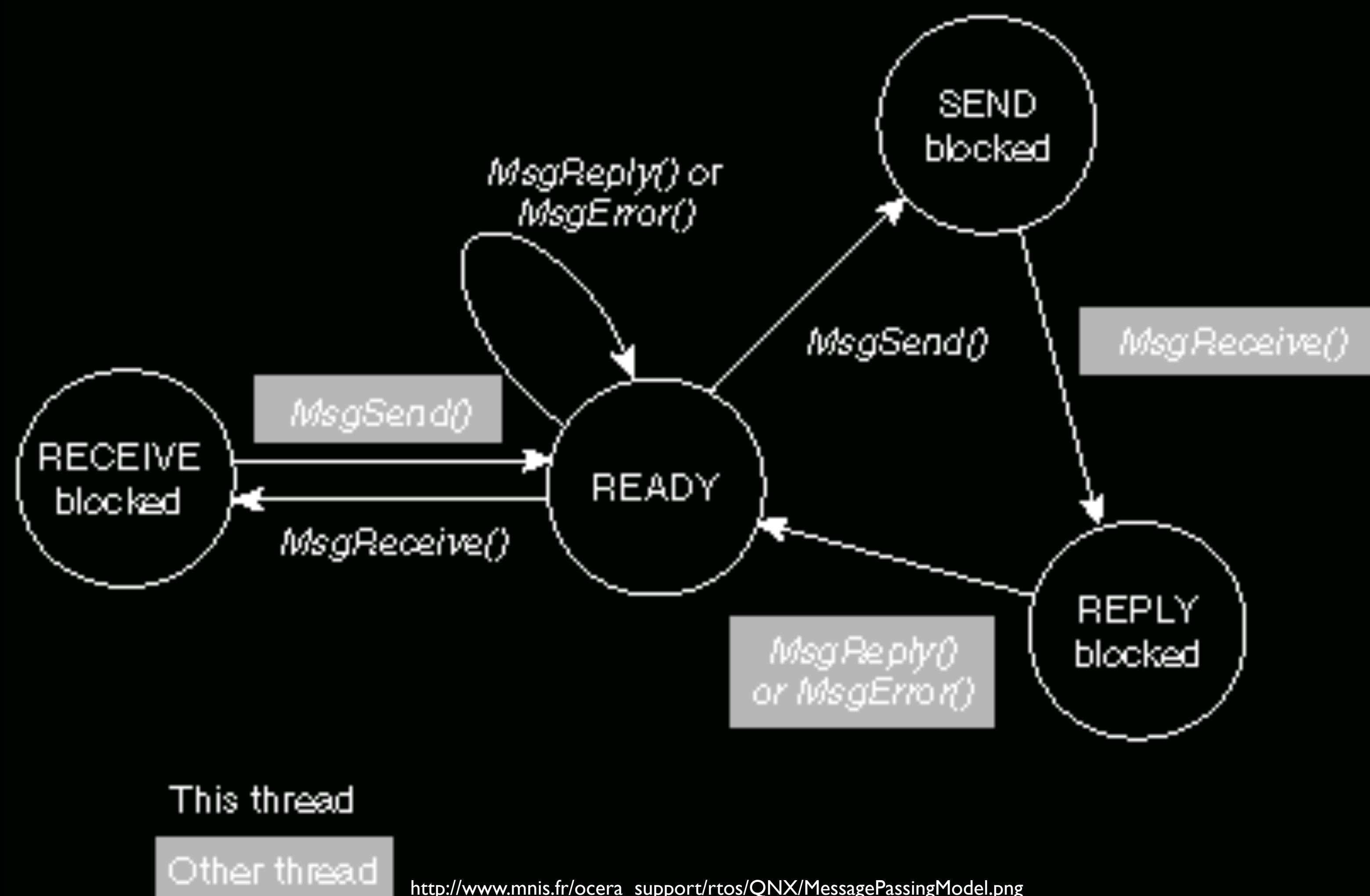
Map of ARPAnet: March 1977

http://upload.wikimedia.org/wikipedia/commons/b/bf/Arpanet_logical_map%2C_march_1977.png

By the late 1970s, Distributed Systems was beginning as an earnest study. Since 1969, ARPAnet had proven to be both a testbed of new techniques and a source of new challenges. Resource management, failures, deadlocks, and all kinds of new system models were being formalized into abstract constructs and applied to or phrased as theoretical questions, such as the well known Byzantine Generals problem (1980) and Dining Philosophers problem (1965).

“Message Passing” Model

- Processes
  - state machines
  - “computers”
- Channels
  - reliable?
  - ordered?
  - directed?
- Messages



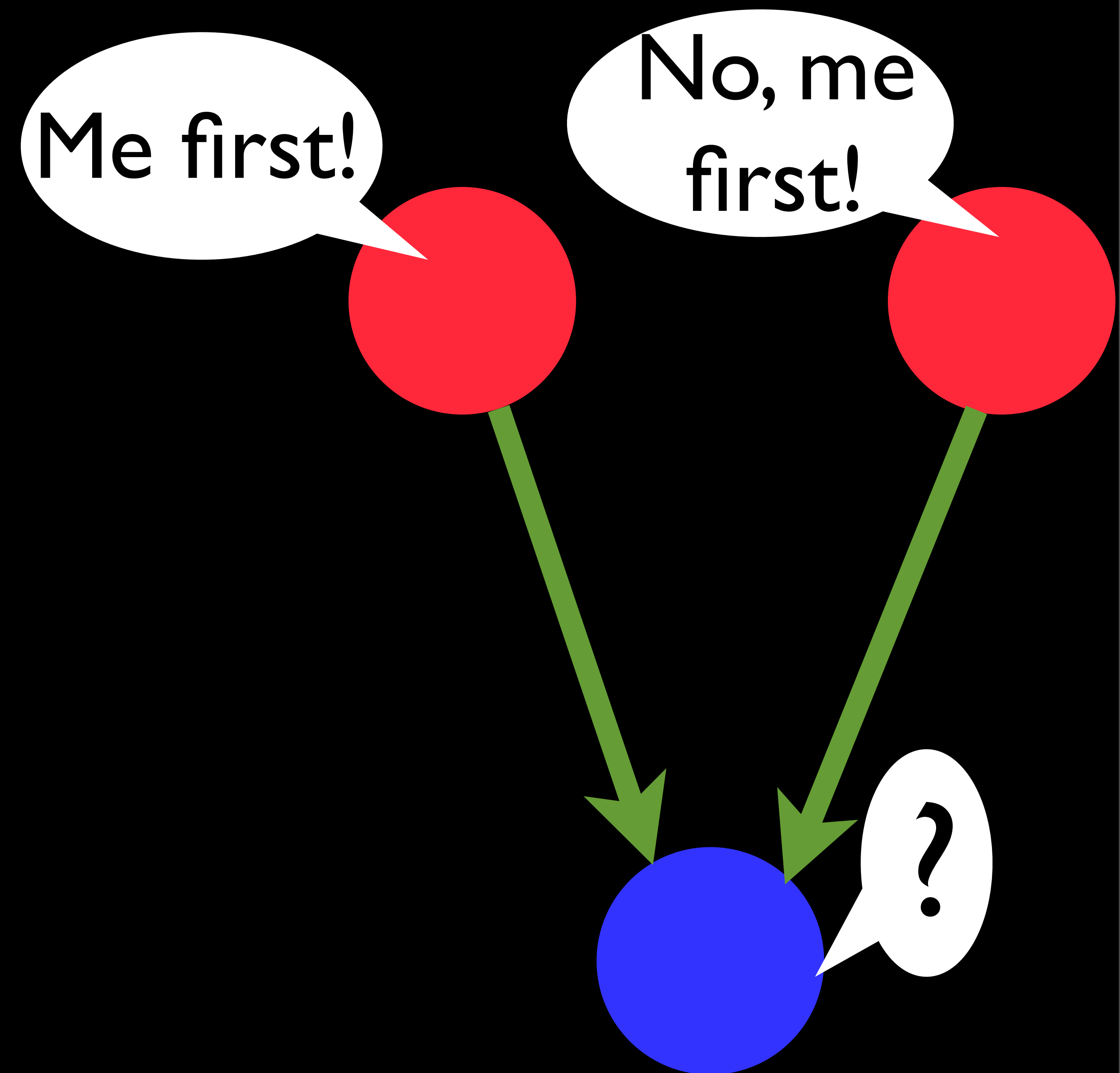
http://www.mnis.fr/ocera_support/rtos/QNX/MessagePassingModel.png

Monday, October 15, 12

At this time, the “Message Passing” model is emerging. This is the model we’ve seen in many publications thus far. Some of the common motifs are a set of processes, representing “computers” in some sense, or often finite state machines, connected by various kinds of channels (reliable, ordered, directed) between pairs of processes, communicating in units of information called messages which travel through these channels.

Ordering Events

- No Universal Time
- Resource Allocation
- Fairness



Monday, October 15, 12

Ordering events has proven critical – What if you get two resource requests and can't tell which came first? Or can't tell if a resource was released before it was granted?
In system analysis, ordering events is also critical. Without it, it's extremely difficult to identify "cause" and "effect."

Clock Synchronization

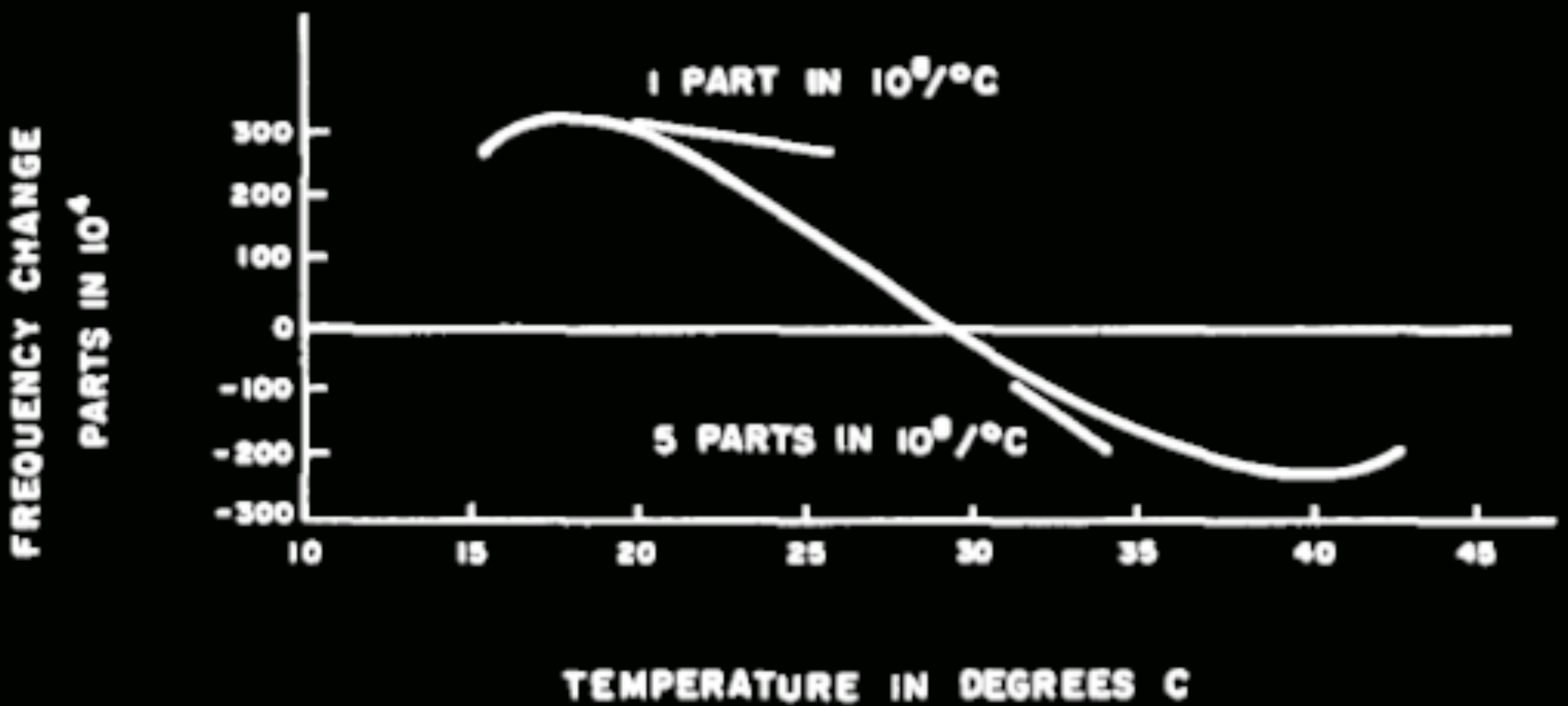
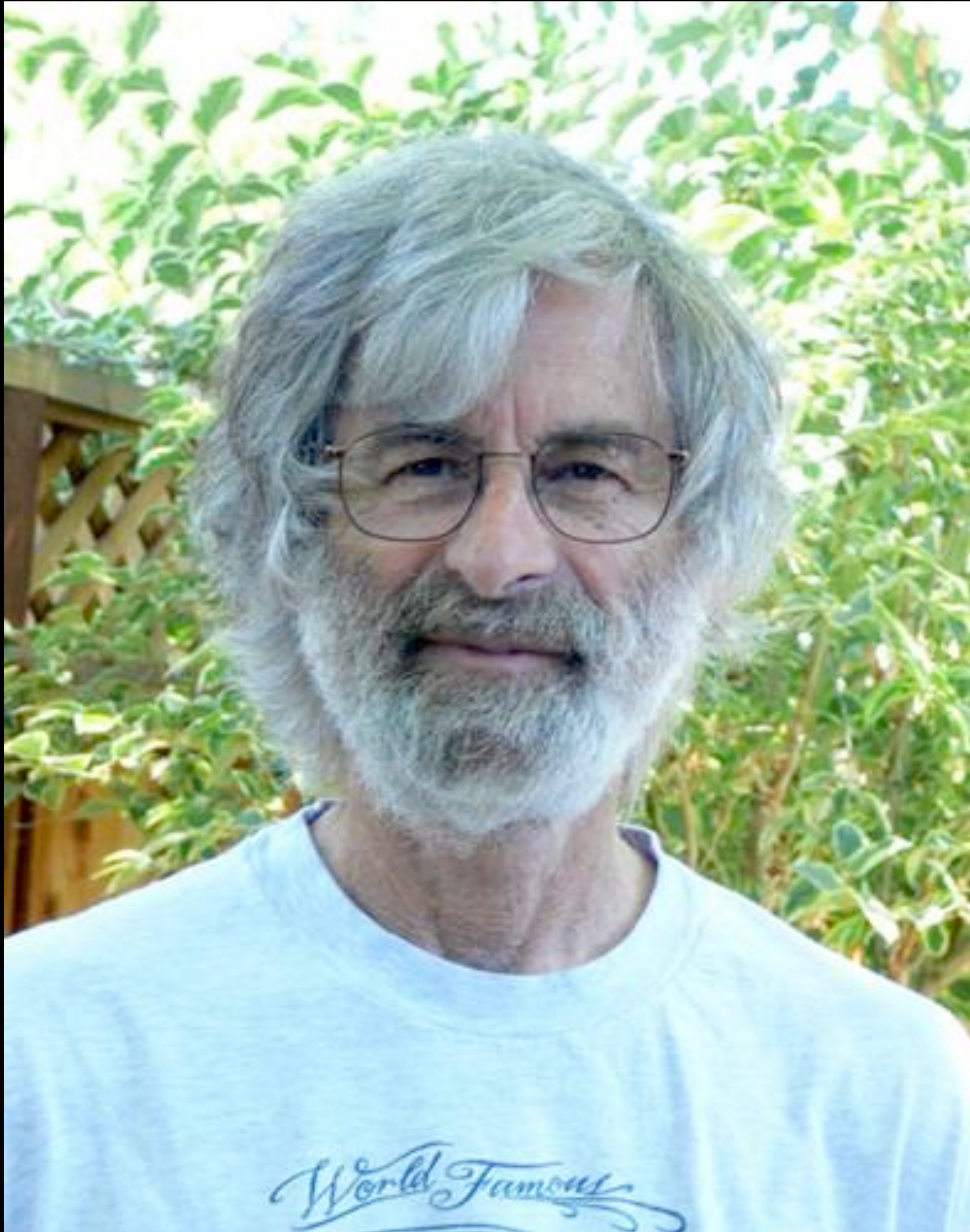


Fig. 4. Frequency change versus temperature, *AT* cut crystal.

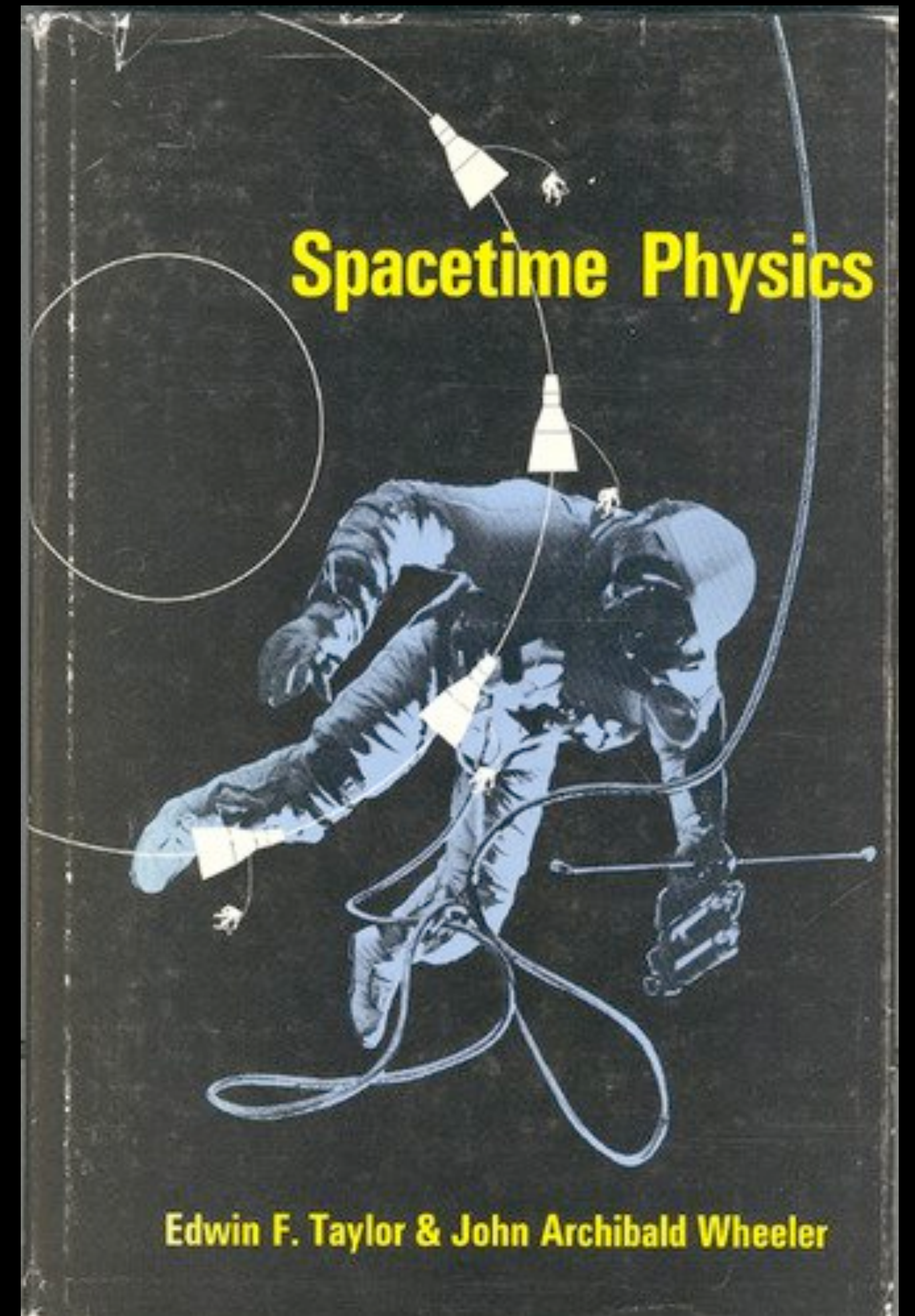
Ellingson, C, and Kulpinski, R.J. Dissemination of system-time. IEEE Trans. Comm. Com-23, 5 (May 1973), 609.

Earlier event-ordering solutions rely on synchronized clocks - Lamort cites a communications paper about syncing clocks, with analysis of electric clock stresses and everything. Syncing clocks is really hard. You have to deal with hardware unpredictability, message delays, and so on. We can't do it perfectly, even today.

Leslie Lamport



http://research.microsoft.com/en-us/um/people/lamport/leslie.jpg



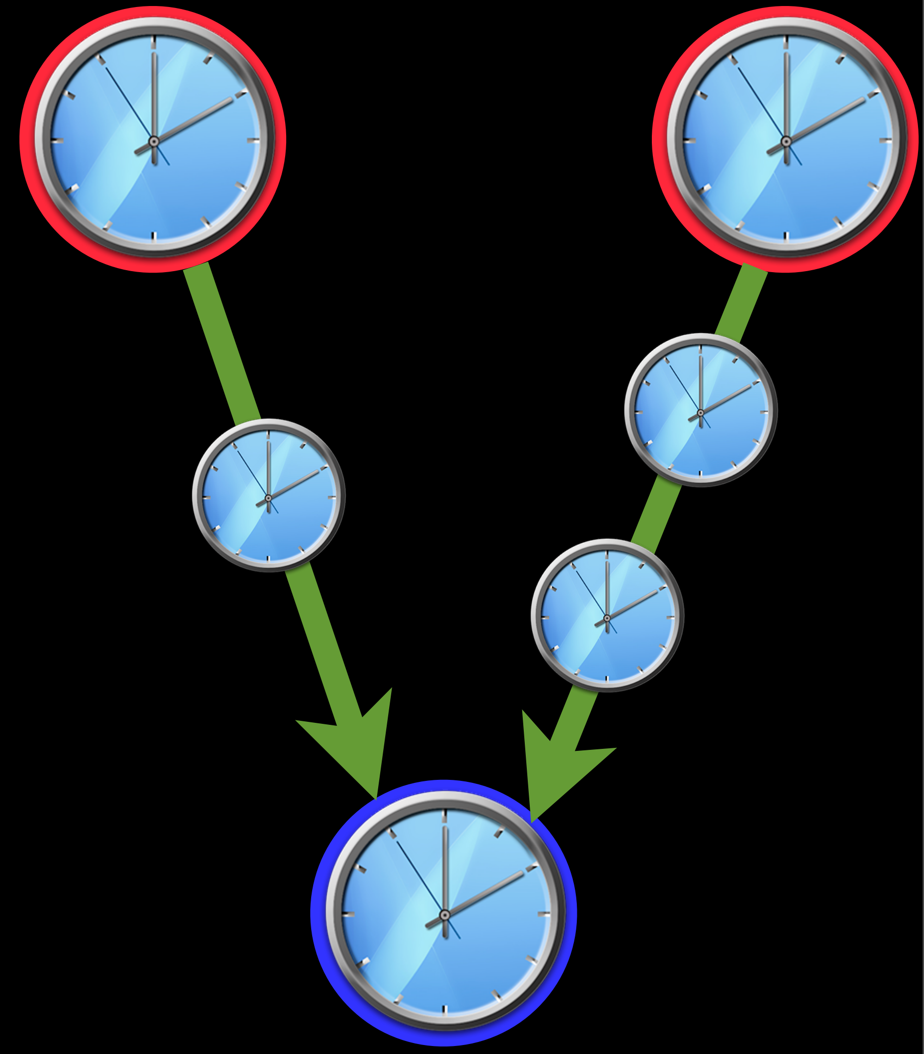
http://ecx.images-amazon.com/images/I/61hFjIKj27L._SL500_.jpg

Monday, October 15, 12

Leslie Lamport, who now works at MSR, but at the time worked at Massachusetts Computer Associates, (presumably after seeing star wars) reads special relativity books, including Taylor and Wheeler (which is terrible), to learn about how causality works in physics, develops his own method in distributed systems.

The Maintenance of Duplicate Databases by Paul Johnson and Bob Thomas

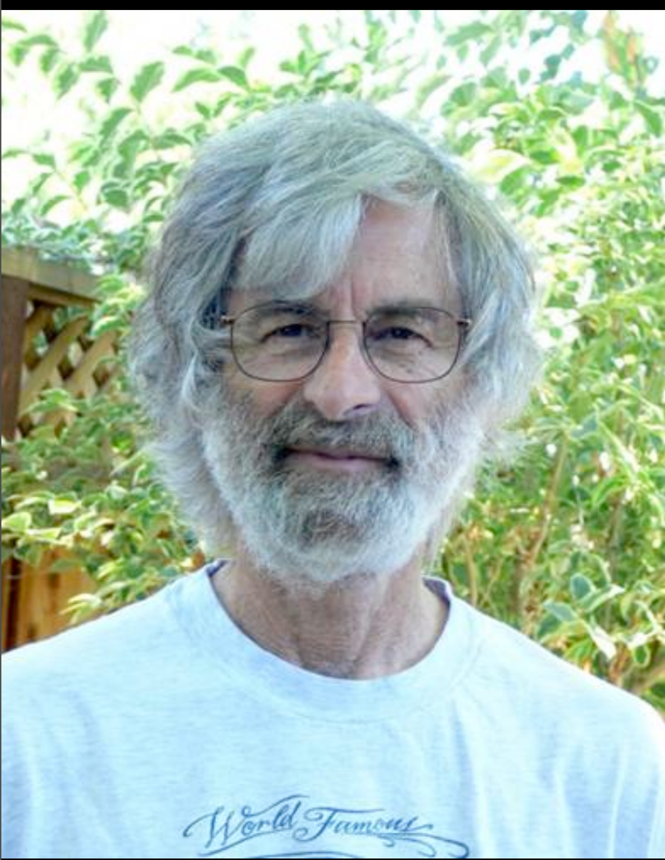
- Timestamps for entries and updates
- Failed if out-of-sync



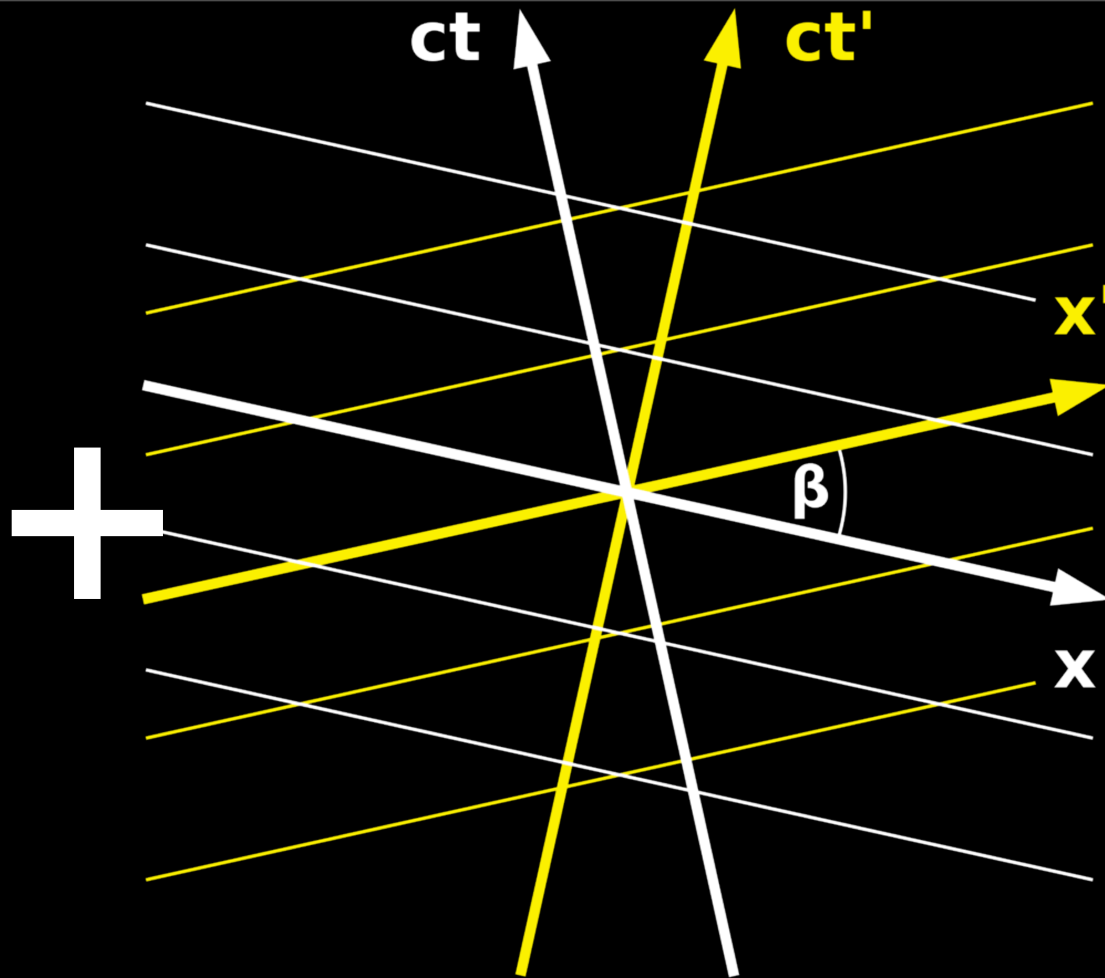
clock image: http://www.dteconsulting.net/wp-content/uploads/2012/07/clock.png

Monday, October 15, 12

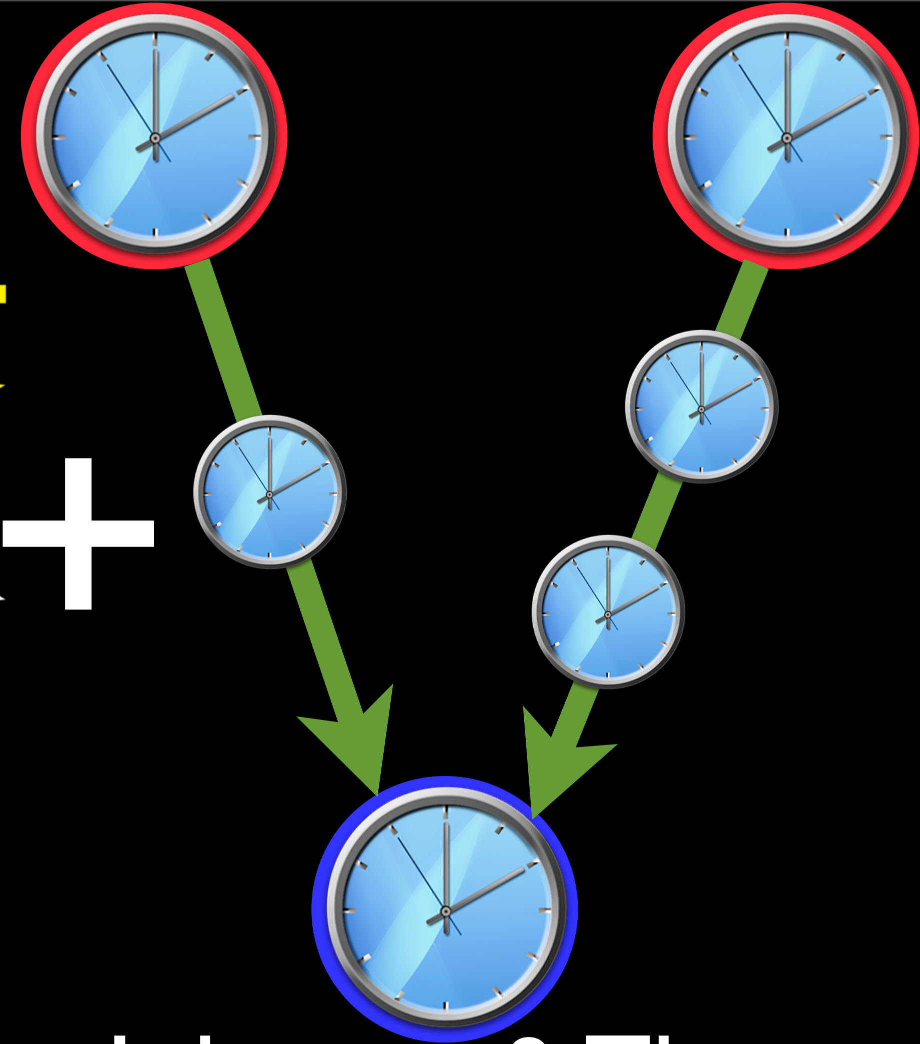
Leslie Lamport encountered a note called “the Maintenance of Duplicate Databases by Paul Johnson and Bob Thomas.” They were trying to maintain synchronized databases over ARPAnet with FIFO channels. The innovation, in their own words, is: “the explicit representation of the time sequence of modifications through timestamps for both modifications and entries [in the database]. This enables the each ... [Processor] to select the same "most-recent" modification of an entry. In addition, timestamps enable the ... [Processors] to decide when a deleted entry is no longer referenced (i.e., still active anywhere) and can be deallocated.”



+



+



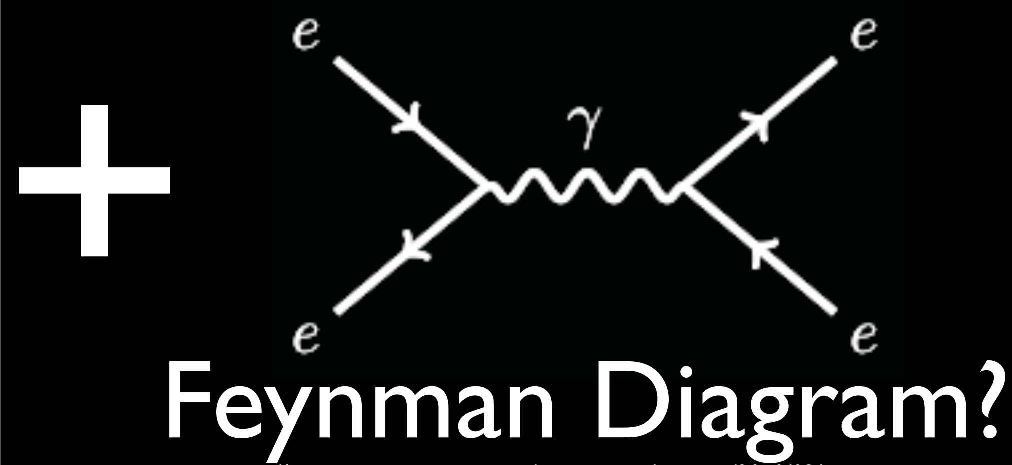
Leslie Lamport

Minkowski diagram

http://upload.wikimedia.org/wikipedia/commons/a/a6/Minkowski_diagram_-_simultaneity.svg

Johnson & Thomas

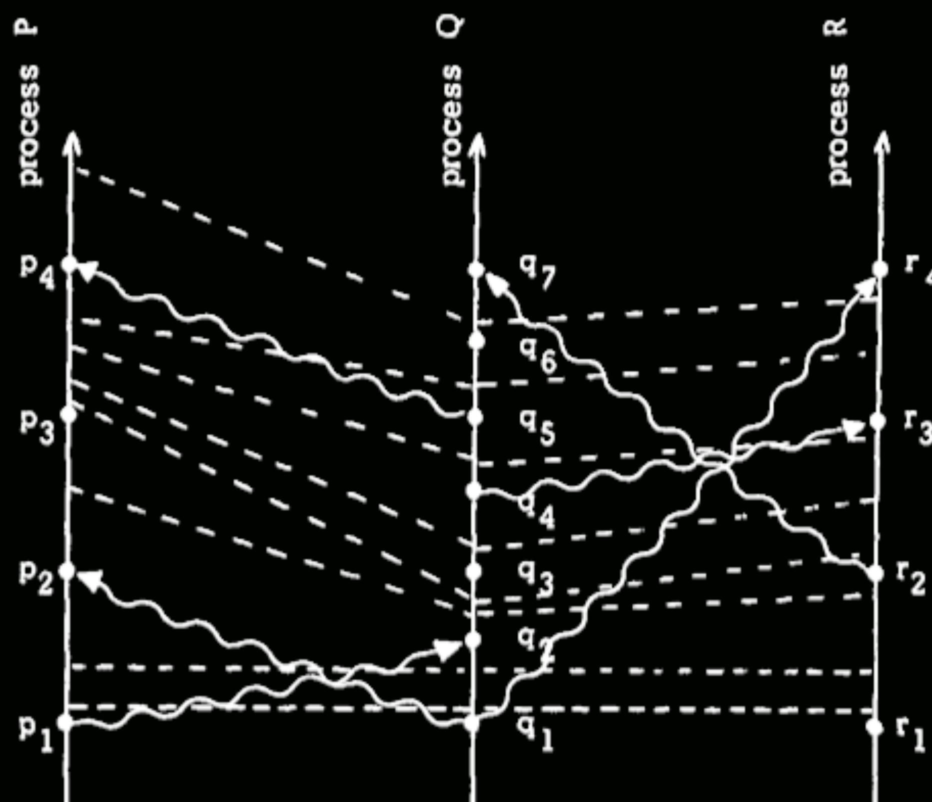
+



Feynman Diagram?

http://www.quantumdiaries.org/wp-content/uploads/2010/03/eetoee1.png

=



???

Monday, October 15, 12

Lamport's website has a story of his thought process:
I happen to have a solid, visceral understanding of special relativity. This enabled me to grasp immediately the essence of what they were trying to do. Special relativity teaches us that there is no invariant total ordering of events in space-time; different observers can disagree about which of two events happened first. There is only a partial order in which an event e1 precedes an event e2 iff e1 can causally affect e2. I realized that the essence of Johnson and Thomas's algorithm was the use of timestamps to provide a total ordering of events that was consistent with the causal order. This realization may have been brilliant. Having realized it, everything else was trivial. Because Thomas and Johnson didn't understand exactly what they were doing, they didn't get the algorithm quite right; their algorithm permitted anomalous behavior that essentially violated causality. I quickly wrote a short note pointing this out and correcting the algorithm.
It didn't take me long to realize that an algorithm for totally ordering events could be used to implement any distributed system. A distributed system can be described as a particular sequential state machine that is implemented with a network of processors. The ability to totally order the input requests leads immediately to an algorithm to implement an arbitrary state machine by a network of processors, and hence to implement any distributed system. So, I wrote this paper, which is about how to implement an arbitrary distributed state machine.
(http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#time-clocks)

Lamport combined Special Relativity's Minkowski diagrams, which can be used to show how simultinaity is different from different viewpoints with the message-timestamping of Johnson and Thomas, added some squiggly lines that look like they came out of a Feynman diagram, and drew them on a timeline of an impossible system execution to form some of the least helpful figures in ACM history.

Lamport's Logical Clocks

- Not *real* clocks
- Integer "timestamps"
- Order "Events"
- "Happens Before" relation: →

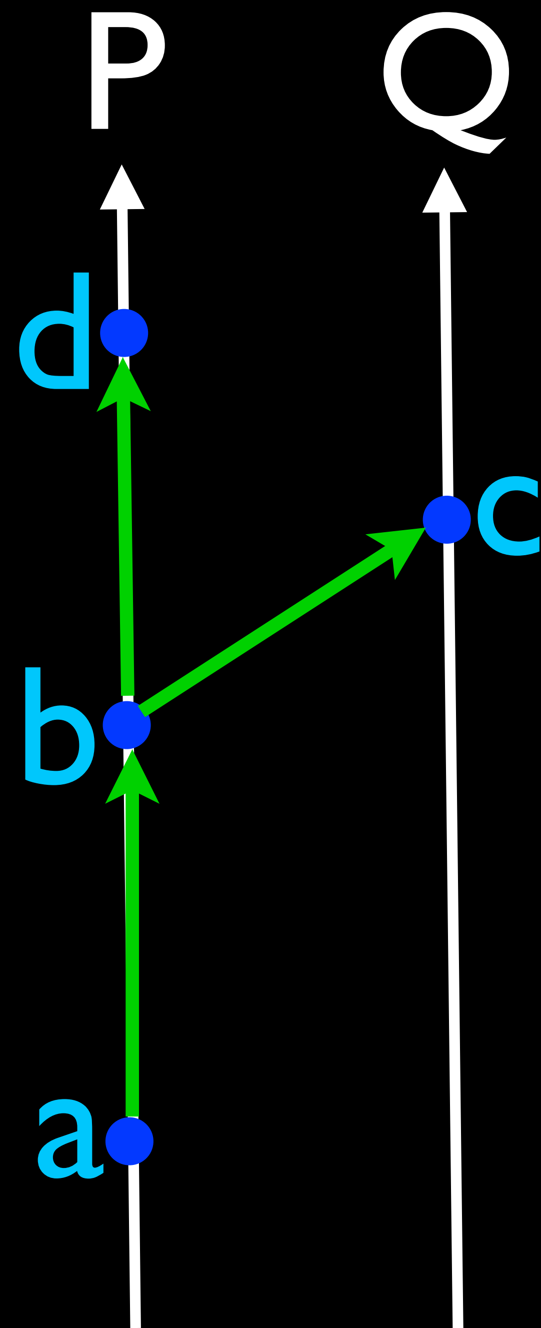
Lamport created what he called a "logical" clock, a clock with no real relation to the passage of time per se, but only the ordering of events. Events are any action or computation which can be said to occur atomically on a process. There are local events on each process, as well as events for the send and receipt of messages. Each event is assigned a logical clock integer "timestamp," representing "when" it occurred.

It is interesting to note that in his story, he recalls the system model as one of state machines connected by channels, but the paper and indeed the clocks have no such requirement. They require only that there exist some set of processes, which may be state machines, which communicate only by sending messages over channels, which need not be FIFO, or reliable.

The minimal implementation of logical clocks provides only a partial ordering of events, ensuring that the timestamp of any event is greater than the timestamps of all events which can be said to "cause" it. A fairly trivial extension provides a complete ordering of events by ordering otherwise unordered events by process id, which is basically arbitrary.

Happens Before

- x precedes y on the same processor: x → y
- x is the sending of a message, and y is the receipt of the message: x → y
- Transitivity: x → y → z implies x → z
- Ex: a → b, b →d, a →d, b →e, a →c
  - but NOT c →d



Monday, October 15, 12

“Happens before,” as Lamport defined it, is a broad notion of causality for generalized systems. For each event, it defines a set of events which distinctly “happened before” it, as opposed to events which are in the future, or concurrent. This can be seen as “potential causality,” since not all events which happened before something are definitely causes, but most system models don’t provide any way for events which don’t happen before something to have caused it.

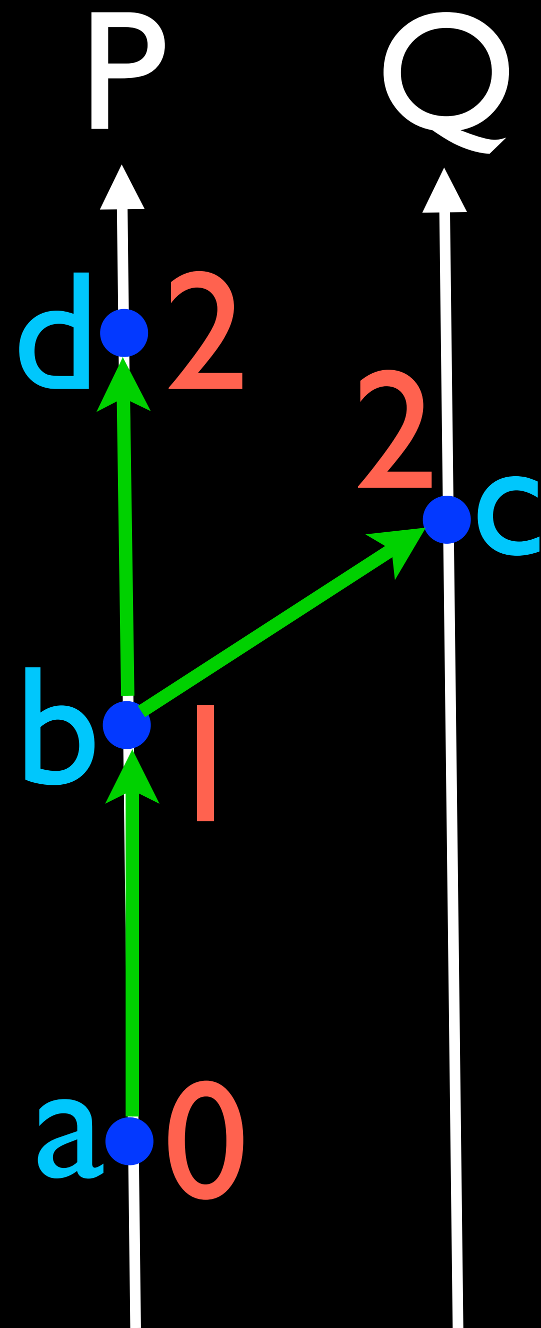
Happens before is based on two basic ideas:
Lamport assumes events are ordered on processes, and so earlier events in this ordering “happen before” later ones.
A message send is said to happen before the receipt of that message. This is in line with most notions of causality.

Happens before is also transitive. If x happens before y, which happens before z, then x happens before z. This means that any chain of local events and message send and receives can form a “happens before” relation.

For example, given two processes named “P” and “Q”, suppose P executes some local event “a”, then sends a message to Q (we’ll call this send event “b.” Then Q receives the message, and we’ll call this receive event “c.” Sometime later, P, executes another local event called “d.”
a happens before b and d, because it occurred earlier on the same process. b happens before c, because c is the receipt of the message sent at b. By transitivity, a therefore happens before c.
however, regardless of real time, by Lamport’s definition, c does not happen before d, and neither does d happen before c.

How it Works

- Every event gets a timestamp (TS)
- Start at TS 0, increase for each event
- Piggyback send event's TS on each message
- On each receive, set timestamp to max(TS in message, local TS) + 1

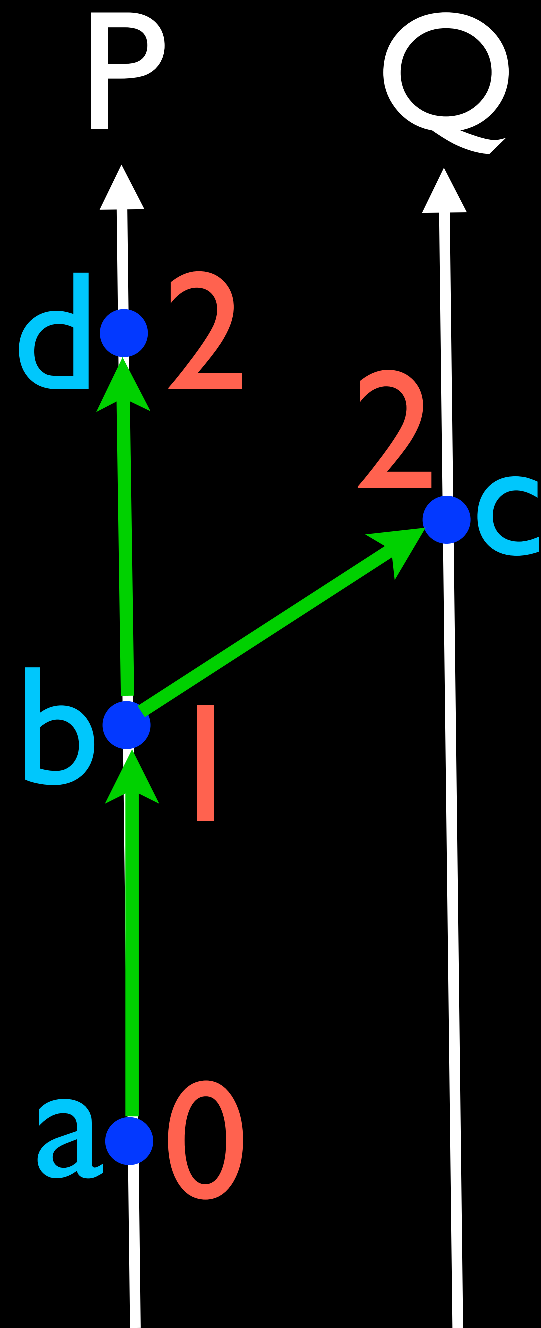


How They Work:
Every event gets a timestamp, which are integers
Processors start with some timestamp (usually 0), and increase timestamp for each event (traditionally by 1)
Piggyback the send event's time on every message sent
On receive event, set timestamp to greater than the local process's current timestamp and the message's piggybacked timestamp (usually the max + 1)

For example, Suppose processes P and Q start with local timestamp values of 0. P has some local event timestamped 0. Later, P sends a message to Q. The timestamp of this send event is 1, sine the local timestamp increments with each event. When Q receives the message, its local timestamp is still 0, but the message carries a piggybacked timestamp of 1, so the max of these is 1, and it is incremented to 2, so the timestamp of the receive event on Q is 2. Sometime later, P executes another local event, again incrementing its local timestamp, which is also now 2.

a→b Implies LC(a) < LC(b)

- Same Process: incrementation
- Different Processes:
  - send-receive: max
  - transitivity: LC(a)<LC(b) & LC(b)<LC(c) implies LC(a)<LC(c)



Monday, October 15, 12

The most important property of Logical Clocks is that Lamport's "Happens Before" relation implies a partial ordering of the integer logical clocks. That is to say that if one event happens before another, then the later event must have a greater logical clock value. The proof is broken down into two cases and a transitive closure:

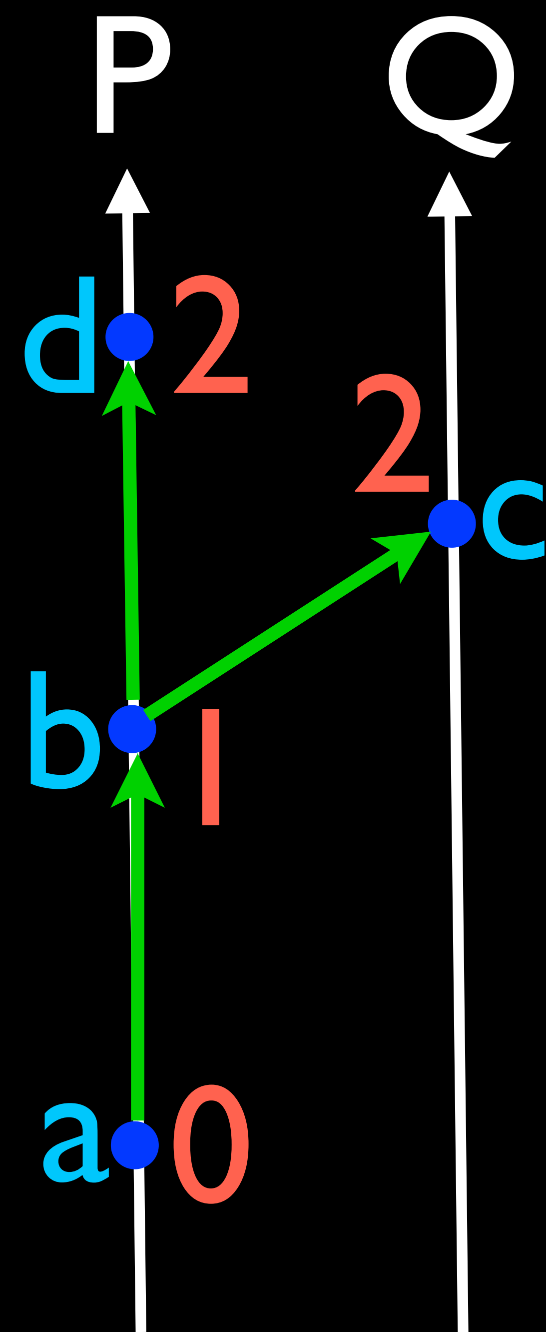
If two events are on the same process, then the later event on that process must have a greater logical clock, since each process only ever increases its logical clock value, and increments between all events.

If one event sends a message, and another receives it, then the receive event has a logical clock value greater than the previous event on its own processor, and the send event, because of the way we piggyback timestamps on messages, and we calculate the receive event's timestamp as greater than max of the old local timestamp and the message's.

Timestamps are integers, and like the happens before relation, the less than relation over integers is transitive, so if there is any chain of events from one event to another, this must be composed of successive local events and send-receive pairs (there is no other way for one action to follow causally from another), and so the "earlier" events in this chain will have lesser timestamps than the "later" ones.

Lamport's Extensions

- Total Ordering: break ties by process
- "Real" Clock Synchronization
- Resource Allocation



Monday, October 15, 12

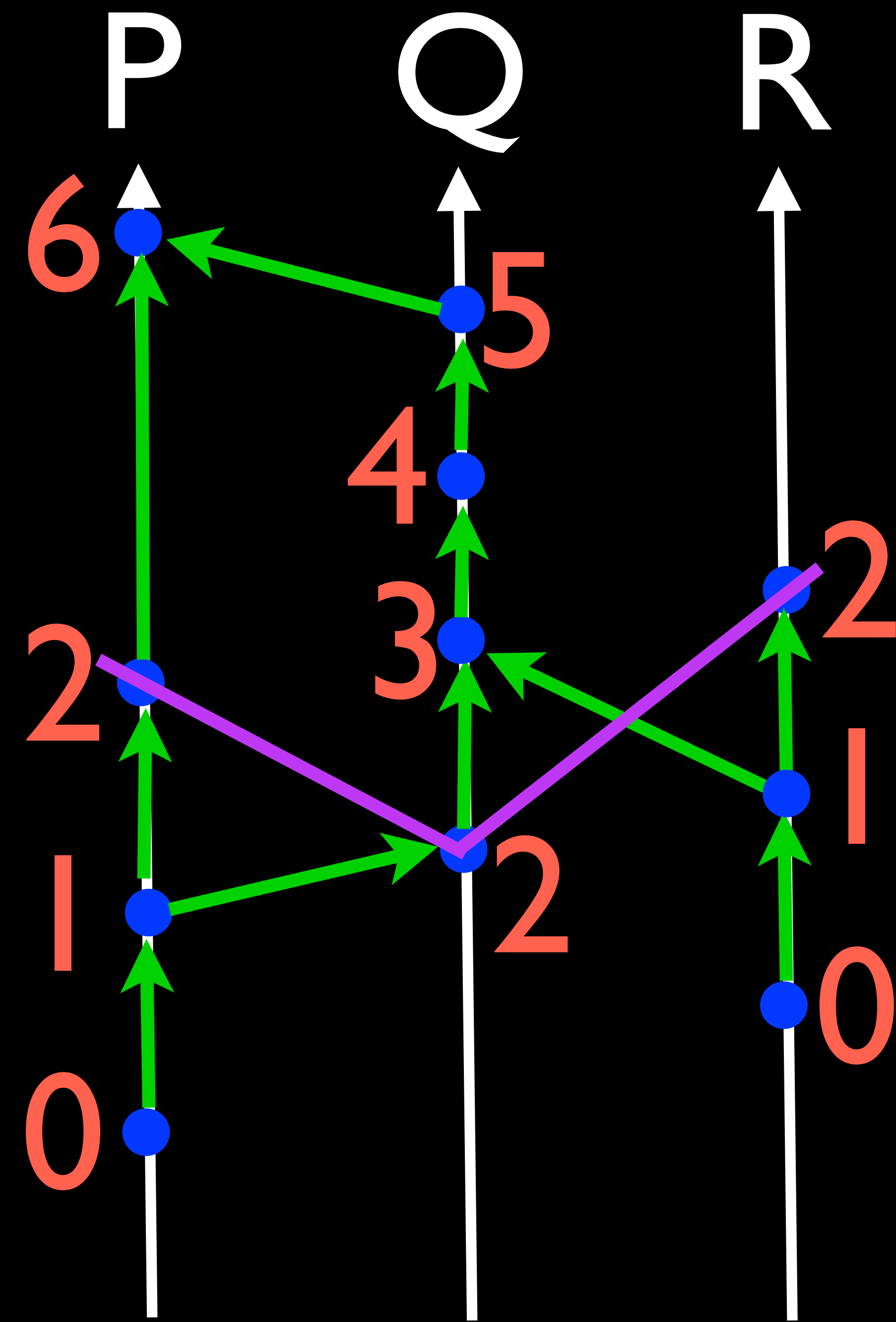
Lamport points out some basic uses of these Logical Clocks in his paper. In addition to the kinds of things Johnson & Thomas were trying to accomplish with their database (which entry value or update is most recent, or what is their exact order), Lamport could build a total ordering of events in a system from his Logical Clocks. The idea is to start with a partial ordering given by the Logical Clock values. No two events on any process can have the same value, but it's possible for two events that don't "happen before" each other to share a logical clock value if they're on different processes (like events c and d). In this case, we break ties with some arbitrary ordering of processes. Suppose, for example, we declare process P to be less than process Q. Now we break the logical clock tie between events d and c, and the total event ordering is a,b,d,c.

Lamport also presents a distributed algorithm for allocating an indivisible resource. It ensures that no two processes can simultaneously consider themselves in possession of a resource, since the ordering of processes claiming and releasing it can be established.

Working backwards towards the goals of prior research, Lamport goes on to present a synchronization for traditional clocks (to within a certain margin), in this case with the added benefit of never moving clocks backwards, which can be extremely useful for ordering events.

Logical Clocks in the Model

- Not necessarily "real" time
- Allows more analytic reasoning about causality
- Example: "Frontiers" of events with equal LCs



Monday, October 15, 12

Logical clocks have proven an extremely useful tool when reasoning about models of distributed systems. They have fairly minimal model requirements, which again are synchronous processes communicating only by sending messages to each other. They are in some ways a more lax notion than that of "real" time.

Consider this (three process) system. Each process is represented by a timeline. In "real" time, it is clear that there are many cases of events on different processes with larger LCs taking place with real time less than events with smaller ones. The event with LC 3 on process Q and the event with LC 2 on process R, for example.

One example of the kind of analysis that logical clocks facilitate is the notion of Frontiers, in this case, a set of events, with one on each process, such that no event after the frontier (on a process) causes any event before the frontier (on a process). Any set of events on all processes that all share the same LC form a frontier. Note that if a message is received before the frontier, it must also therefore have been sent before it. However, a message sent before the frontier is not necessarily received before it.

The proof follows cleanly from the fact that causality implies that the cause has a greater LC than the effect, and logical clocks always increase with time for events on any process.

Vector Clocks

- Given a set of processes
  - store # events in causal past per process
- $a \rightarrow b$ IFF $VC(a) < VC(b)$
- Can identify "holes" in record of events

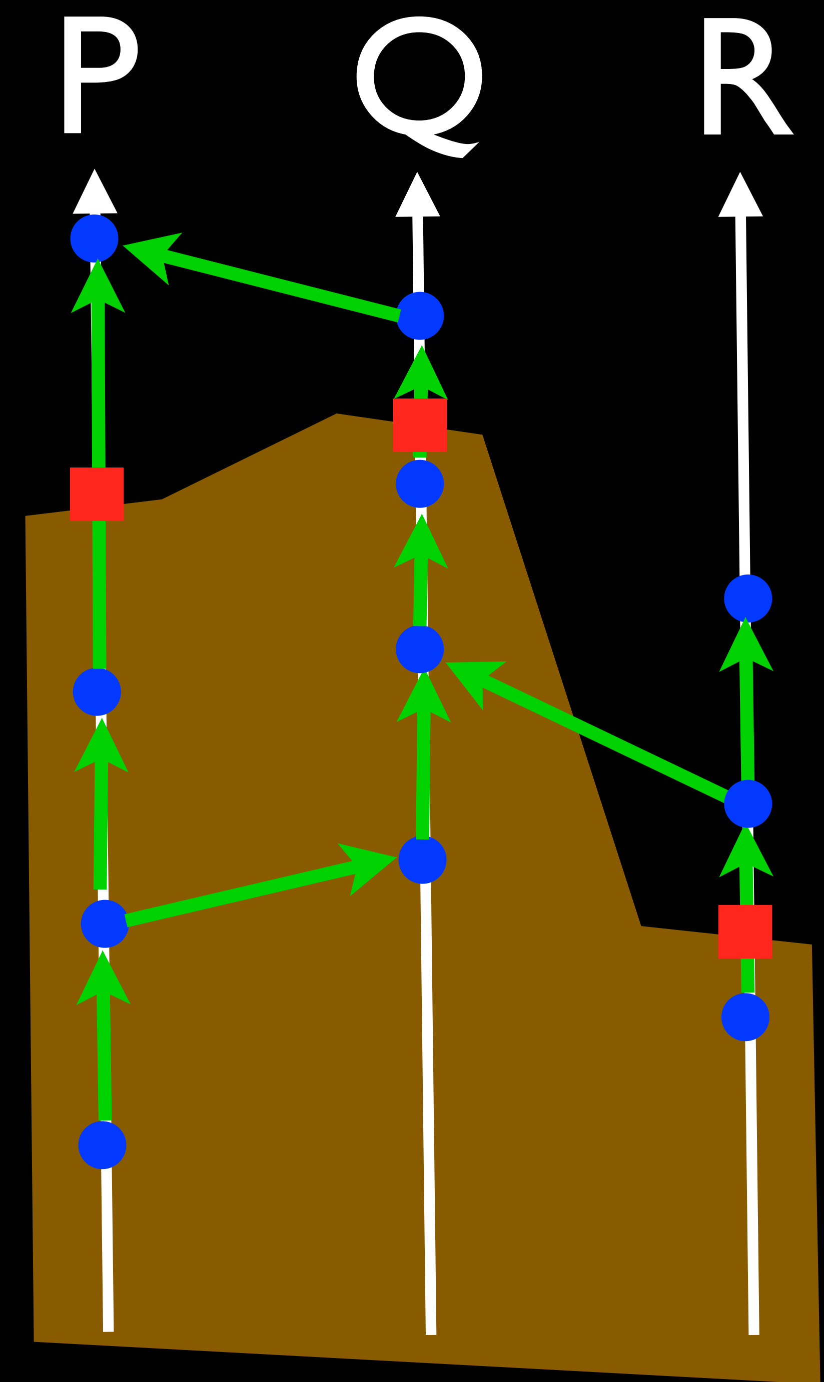
Monday, October 15, 12

There is a generalization of the idea of logical clocks known as Vector Clocks. Given a number of processes, each with a unique ID known in advance, all timestamps consist of a vector containing the number of events in the causal past on each process. Messages still carry piggyback clocks, and componentwise max is performed, with the local process' entry being incremented.

These also allow, given a set of events, the detection of any "missing" events. You can check to see which event would come before which other one, and identify "missing" ones. Given a frontier, you can identify exactly how many events have occurred on each process before the frontier.

Cuts

- State of each process (not necessarily at the same time)
- Set of events on each process prior to some point on that process



Monday, October 15, 12

When trying to understand distributed systems in an analytic way, it's often tempting to try to think about the "state" of the system. Logically, such a system state would consist of the state of each process, as well as each communication channel, at some "time." Unfortunately, it's really hard to get a notion of universal time in a distributed system.

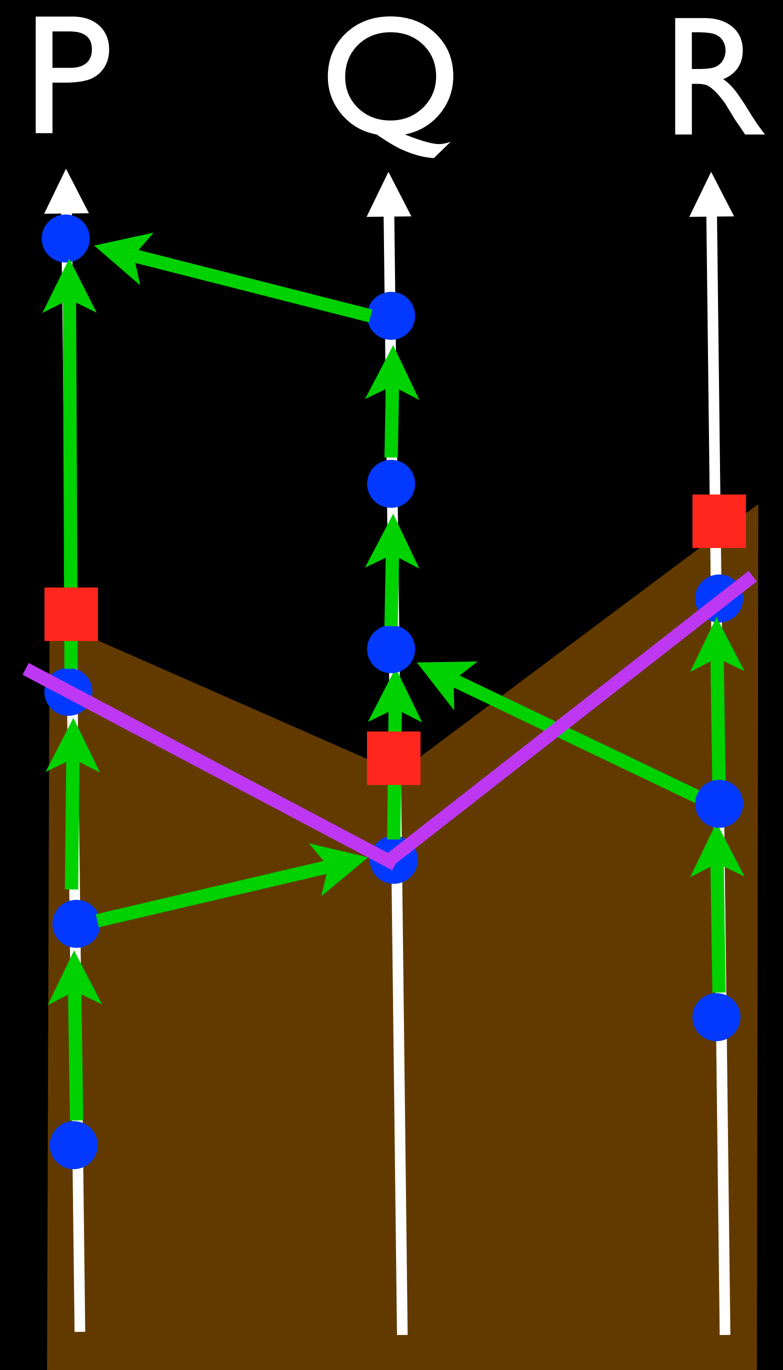
When discussing the notion of the state of a system as a whole, we talk a lot about Cuts. A cut, quite simply, is the state of each process (although not necessarily at the same time). Since we often assume processes are deterministic, the state of each process at some point is wholly determined by, and in some ways equivalent to, the set of events that have taken place on that process prior to that point.

In this example, the red squares represent the states of their respective processors, and the dark orangish area covers the set of events these states reflect on their respective processors.

This Cut is what we call INCONSISTENT. The state of process Q in this cut reflects that it has received a message from process R. The state of process R, however, does not reflect that any such message has yet been sent.

Consistent Cuts

- State of each process (not necessarily at the same time)
- No event not "in the cut" happens before an event "in the cut"
- Last events "in the cut" form a frontier



Monday, October 15, 12

For a cut to be "consistent," it must be true that if some event "a" happens before "b", then if b is in the cut, a must also be in the cut. That is to say that no event not in the cut may happen before any event in the cut.

If you think of a cut as a state of each process, then it is sufficient to say that a cut is consistent if not message sent by a process after the state it held in the cut arrives at another process before it holds it's state from the cut.

The last event each process executes in the cut, which would be the last event they execute before entering the state they're in in the cut, form a frontier: a set of events such that no event after a frontier causes an event before a frontier.

As we've mentioned before, one way to find such frontiers is to find an event on each process with the same Logical Clock value. Not all frontiers, however, will have this feature.

Stable Properties

- Property of a System
- Once it becomes true, it stays that way
- Termination
- Tokens
- Deadlock
- Failure
- Database Commits (hopefully)

http://upload.wikimedia.org/wikipedia/commons/c/c8/Ouroboros-simple.svg

One thing that you might like to detect with a Cut is a stable property. A stable property is some fact about the system, some property which, once it becomes true, stays that way. There are a lot of really useful stable properties to detect. They include, but are not limited to:

Termination: The computation being performed by this system is done.

Tokens: Some indivisible allocation exists. There are a finite number of processes which can have the “token” representing this allocation, and it can be at some number of processes, or in transit, but not at more than that number. The property “no more than foo processes have the a token” is supposed to be stable.

Deadlock: One process cannot proceed without progress, ultimately, from itself. Perhaps it is waiting for some other process which is in turn waiting for it. Once deadlock arises, it is permanent (barring some deadlock breaking system which would mean we weren’t “really” deadlocked).

Failure: Some process or set of processes has ceased to function. Failure is assumed to be permanent in many models, such as the popular “crash failure” model.

Database Commits: Once a particular update has been accepted, you don’t want to accept it again, lest a value be updated twice with the same update.

Consistent cuts accurately reflect stable properties.

K. Mani Chandy

K. Mani Chandy

http://www.eas.caltech.edu/images/engenious/2011-Issue8/chandy.jpg

Jayadev Misra

http://www.cs.utexas.edu/~misra/misra.jpeg

Monday, October 15, 12

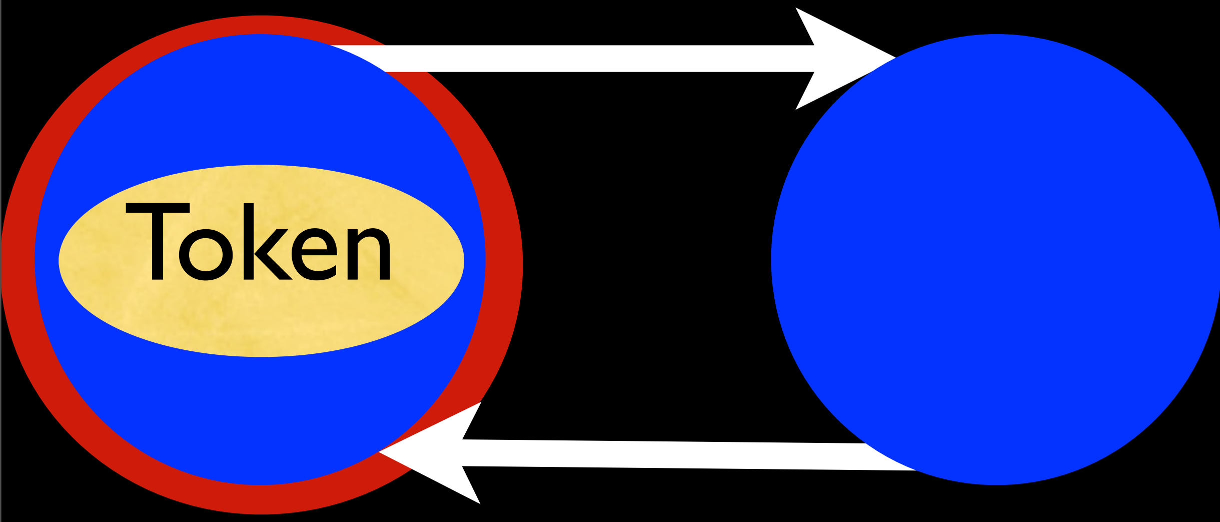
By 1985, cuts and stable properties, as well as deadlock detection and formal reasoning about models of distributed systems had become significant topics of study, attracting the attention of two gentlemen at the University of Texas, Austin. Mani Chandy and Jayadev Misra had been working together on a number of similar projects, many of which were along the lines of Joe Halpern’s research concerning knowledge possessed by system processes.

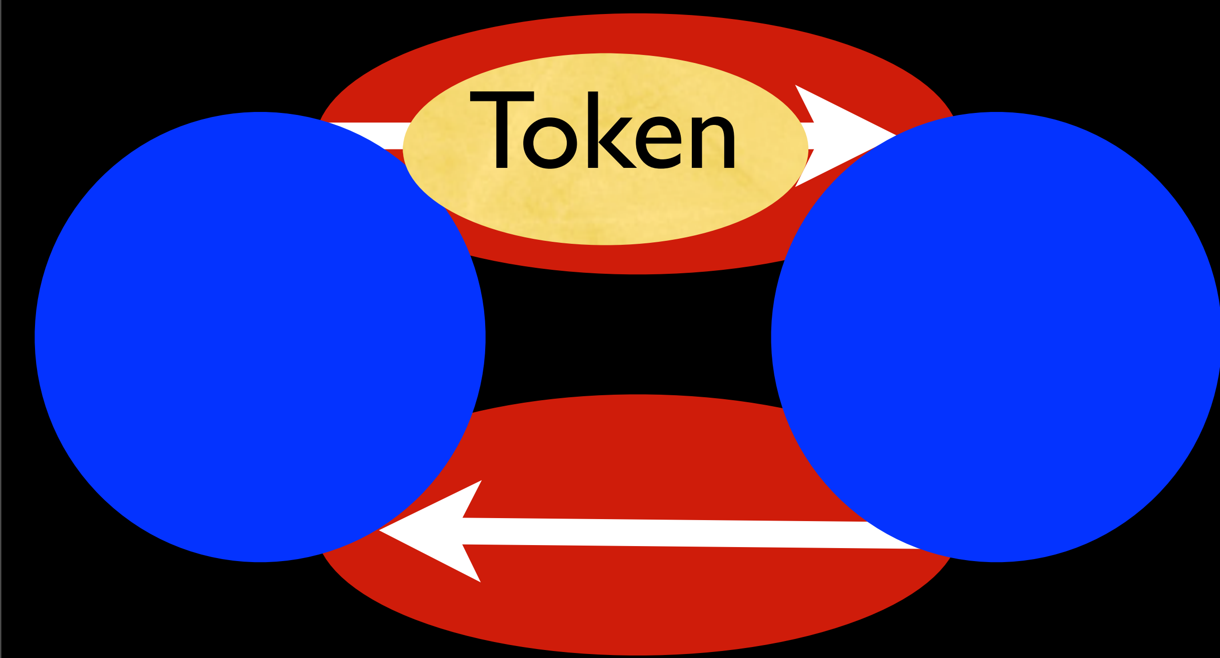
They had worked rather specifically on the model of processors as finite state machienes communicating via FIFO channels, which presents a number of advantages when reasoning about abstract properties, and in many ways can be applied to computers using TCP on, say, ARPAnet.

The two had recently gained some notoriety for their solution to the dining philosophers problem, a classic problem in indivisible resource allocation, and their own extension, the Drinking Philosophers Problem.

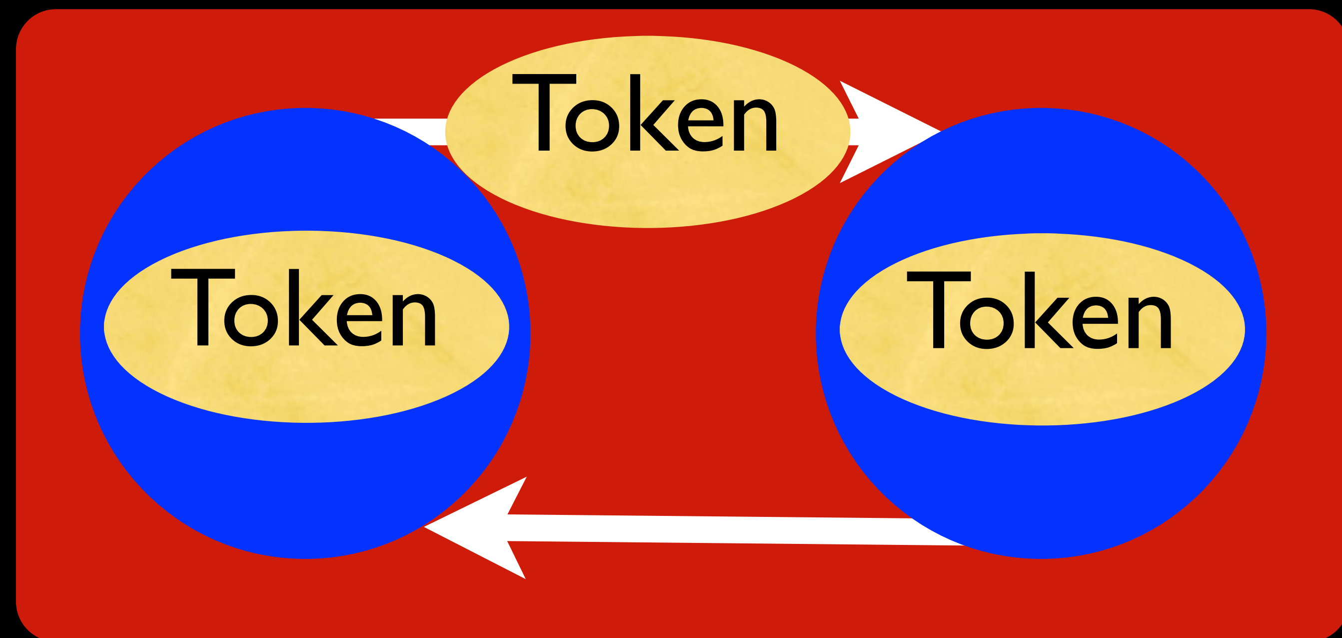
This, by the way, is the happiest expression I’ve ever seen on Chandy’s face. His usual expression is a lot like Misra’s. Maybe it’s just how he looks, or maybe he knows that all his students refer to him as “Man Candy” when he’s not within earshot.

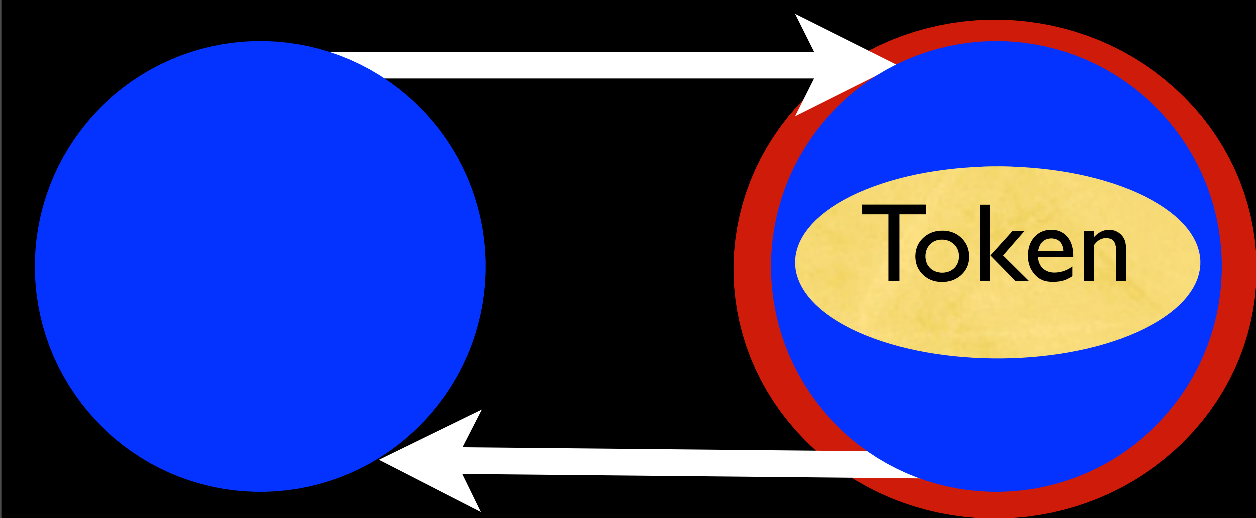
The Trouble With State





System is moving while you record it





Monday, October 15, 12

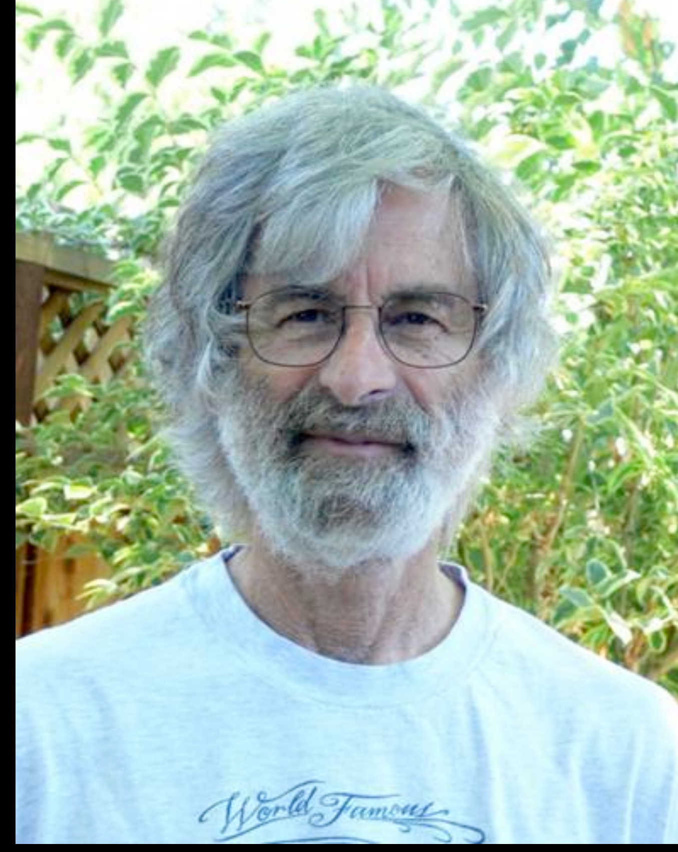
The Trouble with trying to record the state of a system is that the system is changing, undergoing state transitions, while you record it.

Consider a simple system of two processes with channels in both directions between them. There is some kind of indivisible token being passed, which might symbolize, for instance, control over a resource. It can be passed through channels via messages to the effect of "I used to have the token, and now you have it." A Token can therefore be said to be part of the state of a process or channel. It should be a stable property of the system that there is exactly one token.

Someone is recording the state of this system. First, the state of the Left process is recorded, then the state of the channels, and lastly, the state of the Right process. Unfortunately, during this time, the system underwent some transitions. The Token went from the Left process to a channel to the Right Process. As a result, in the state we've recorded, there appear to be three tokens, so we haven't done a very good job of observing stable properties.

Chandy & Lamport

K. Mani Chandy



Leslie Lamport

http://static.zoonar.com/img/www_repository3/ca/c2/8c/10_508d47f0addf4840798d571940b63611.jpg shower: http://www.colourbox.com/preview/3103057-301528-man-and-the-shower-isolated-on-the-black-background.jpg

Monday, October 15, 12

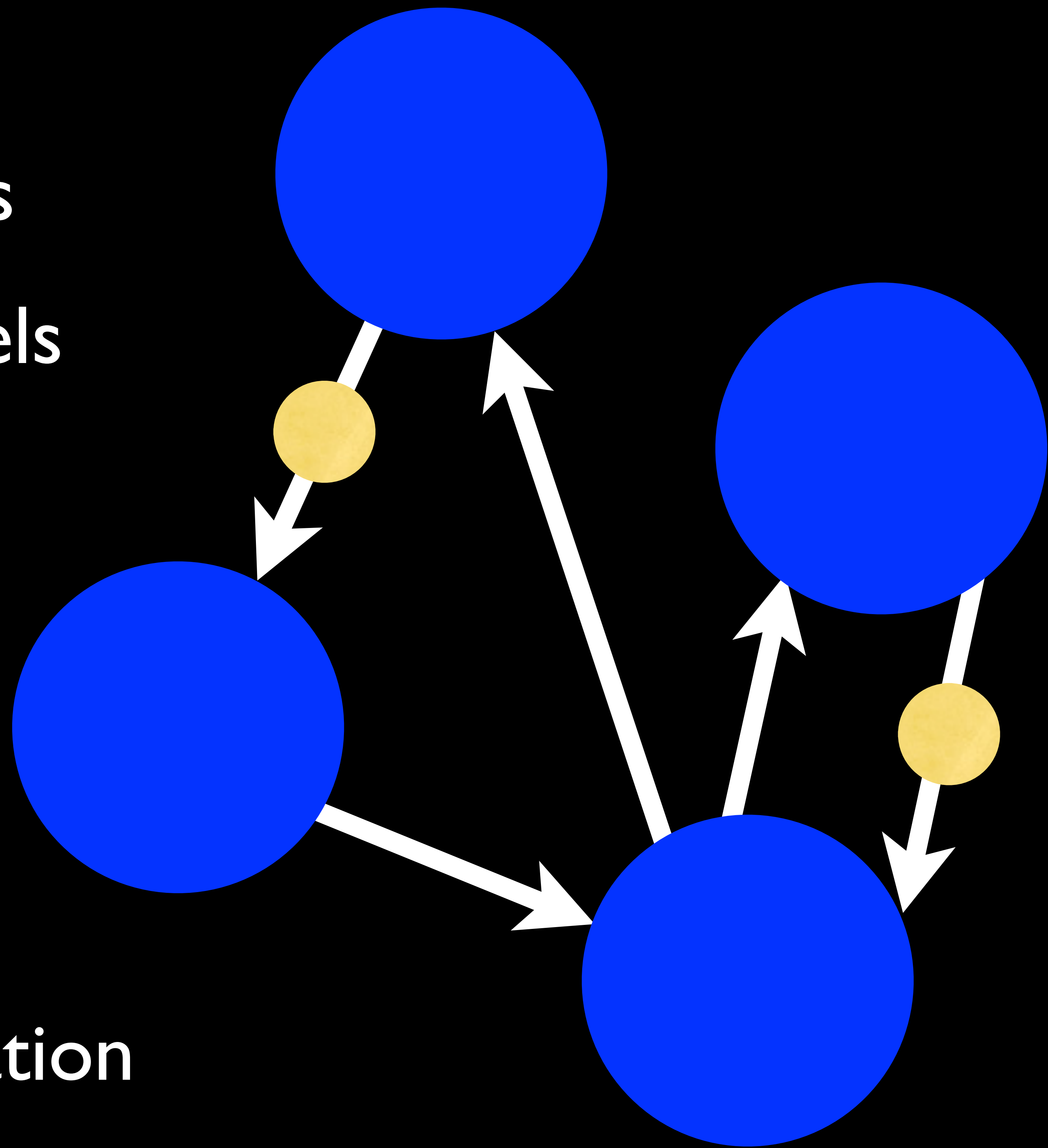
Chandy invites Lamport over for dinner, and they discuss this difficulty, and test Chandy's "Drinking Philosopers" solution. Lamport describes the experience:

"I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution."

(http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html)

Chandy - Lamport Snapshots

- State machine processes
- Communication Channels
  - pairs of processors
  - reliable
  - FIFO
- Atomic Events
- Don’t Disturb Computation

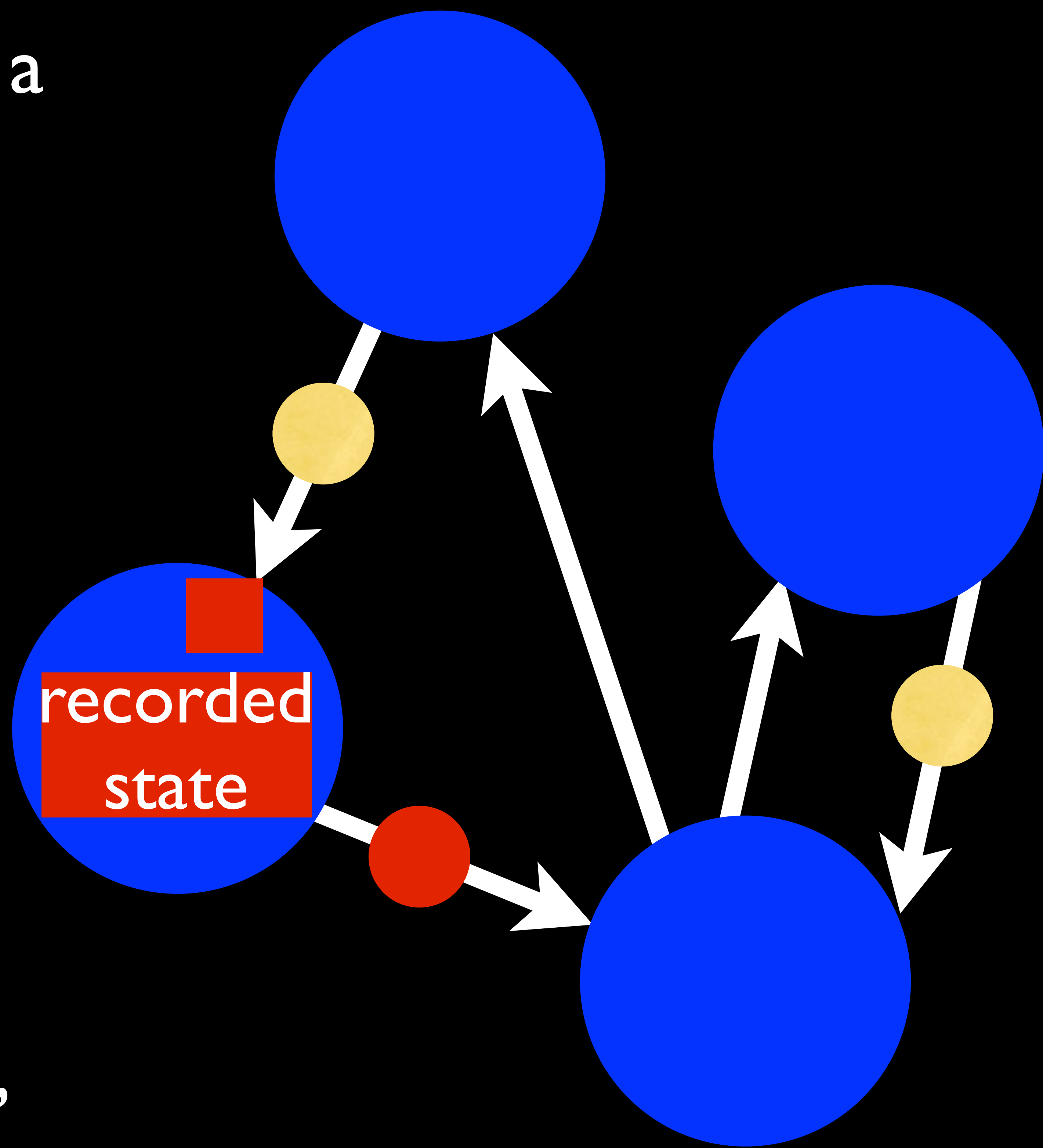


Chandy and Lamport’s Snapshot algorithm is designed to produce a recorded state for each process and channel which preserves stable features of the system. In particular, it should be a system state which is reachable from the state of the system before you started the algorithm, and from which the state of the system after the algorithm is reachable. More on that later.

The model Chandy and Lamport use is:
processors are finite state machines
some pairs of processors have unidirectional FIFO channels
processors send messages
events (including message send/receive) are atomic
we don't want to disturb the computation. That is to say, we’re not going to pause or even significantly slow down any process or channel.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"



Monday, October 15, 12

The Snapshot algorithm creates a new message, called a "snapshot message" or "marker" which can be sent along the channels. It assumes processes are capable of performing atomic, sequential actions, and that they can store things like a record of their own state and a log of messages received.

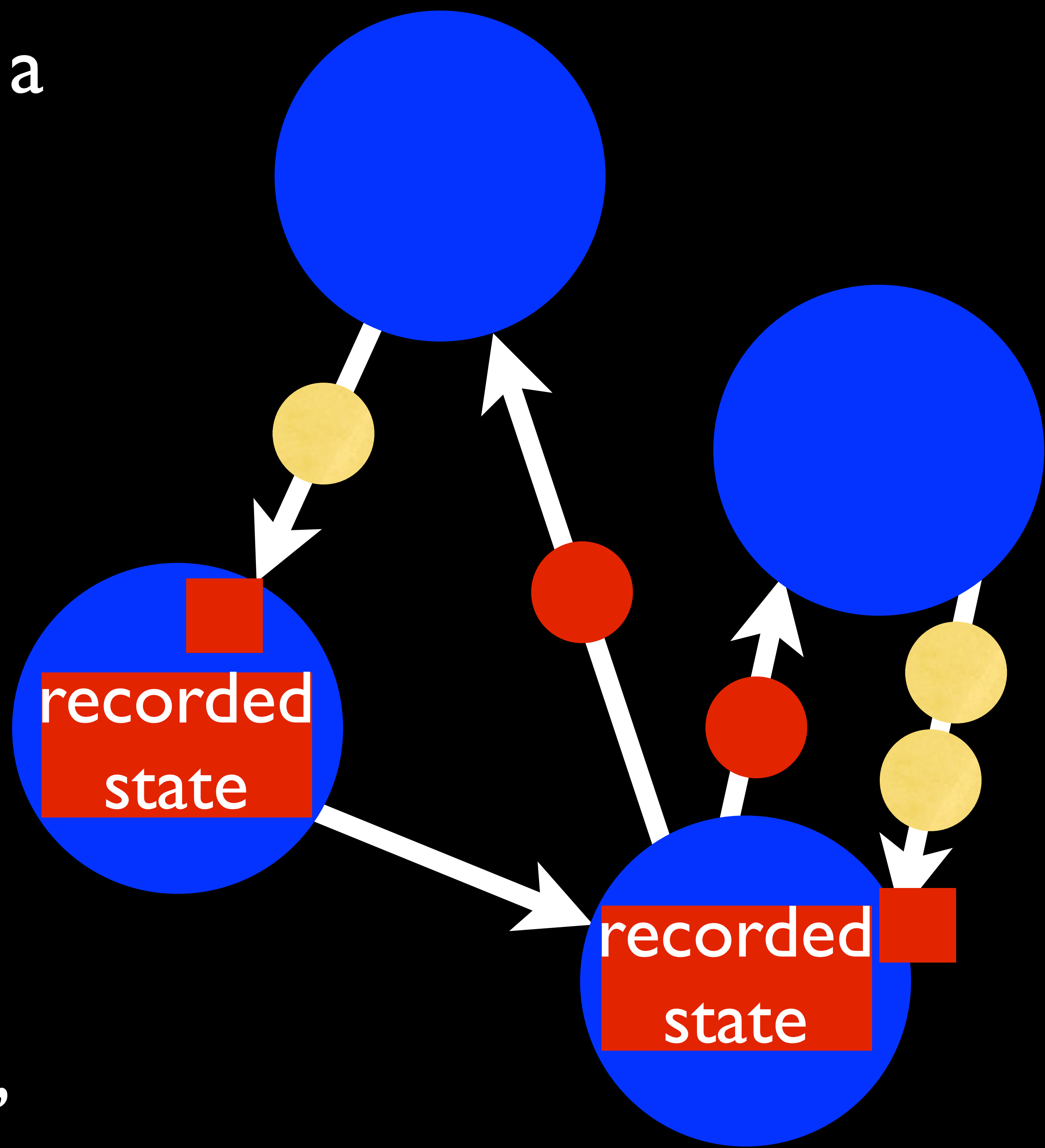
They also assume the graph of processes and channels to be stongly connected.

The idea is that whenever a process receives a marker, it stores a record of its current state, and broadcasts a marker along all outgoing channels. It then continues as normal, logging all messages on each incoming channel until it receives a marker along that channel. At the completion of the algorithm, each process is left with a record of its own state at some time, and a record of the state of each of its incoming channels. Together, these form a snapshot. How and if these are all compiled together is another matter.

The snapshot is initiated when one or more processes act as if they have received a marker along some invisible input channel. They record their own state, prepare to log all incoming channels, and send out markers on all outgoing channels.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"

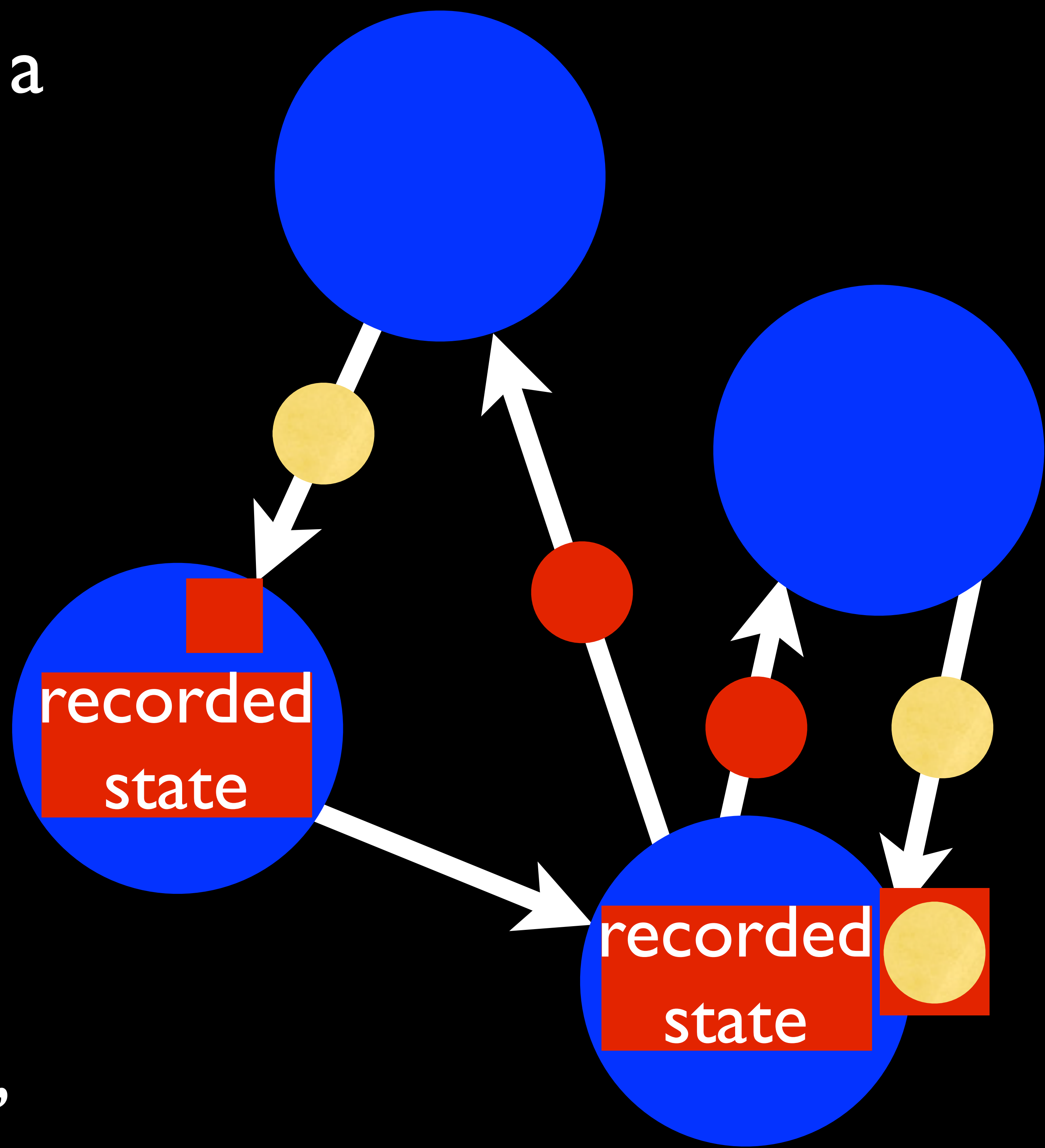


The marker has reached another process, which records its own state, and sends a marker down all outgoing channels.

Meanwhile, the system is still computing, The far right process, for example, has sent another message along its outgoing channel.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"

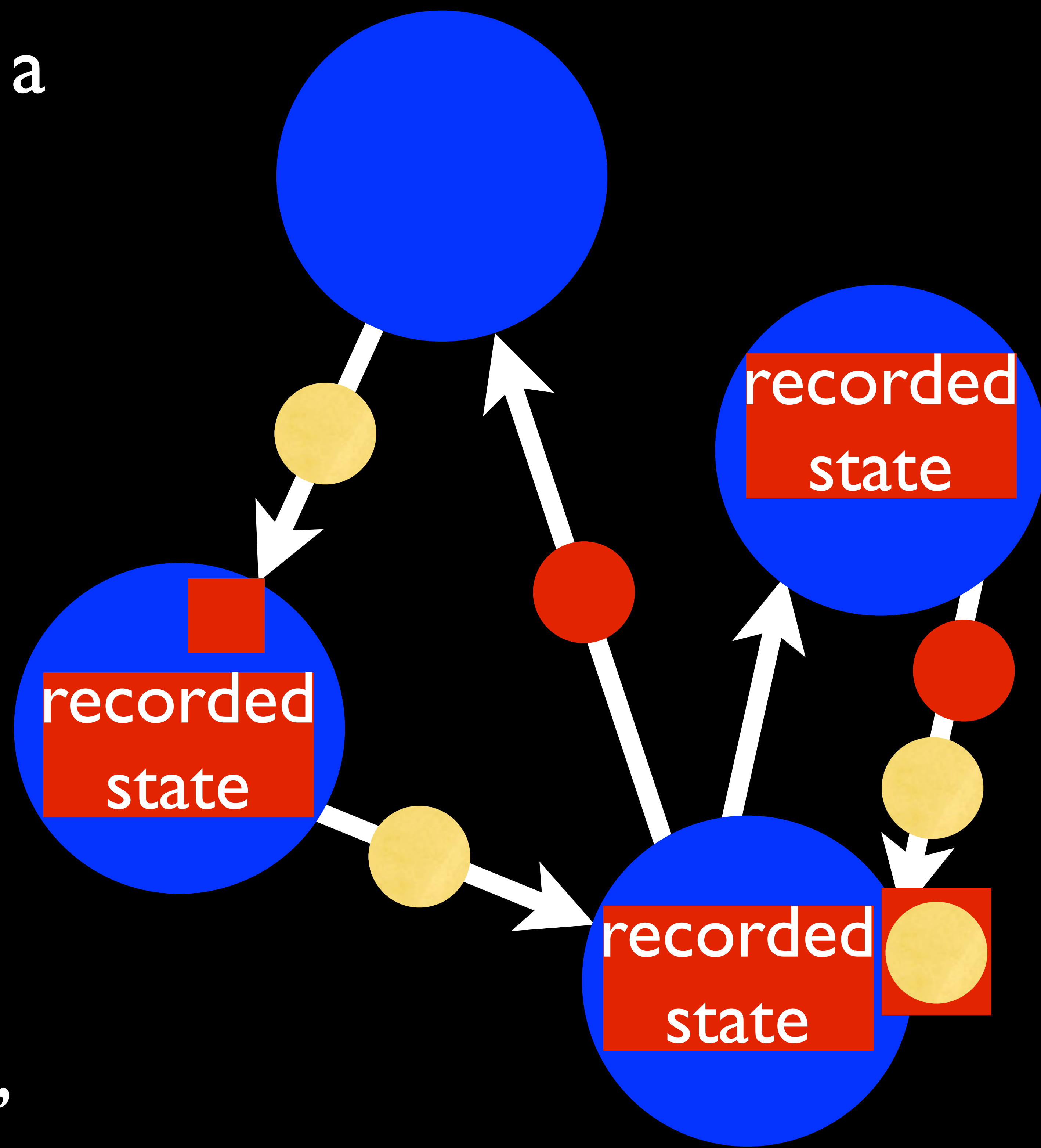


Monday, October 15, 12

When a message is received on the far right channel, it is logged.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"



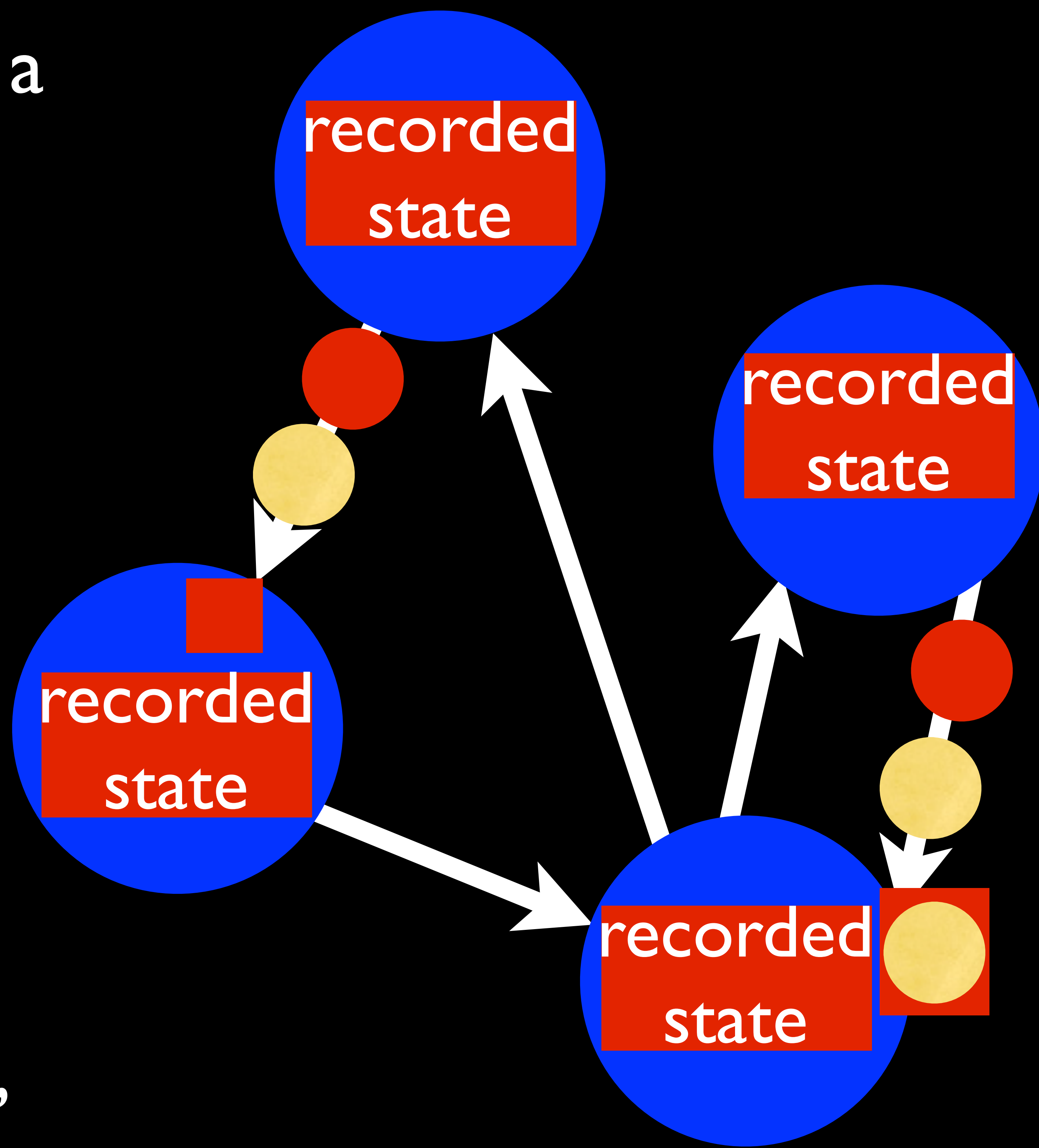
Monday, October 15, 12

The process on the right receives a marker, records its own state at that time, and sends a marker on its outgoing channel.

The leftmost process sends a message not related to the snapshot along its outgoing channel.

Chandy - Lamport Snapshots

- When a process receives a “marker” message:
  - record own state
  - send “marker” on all outgoing channels
  - record incoming channels until “marker”
- One or more processes begin by “pretending” they’ve received “marker”

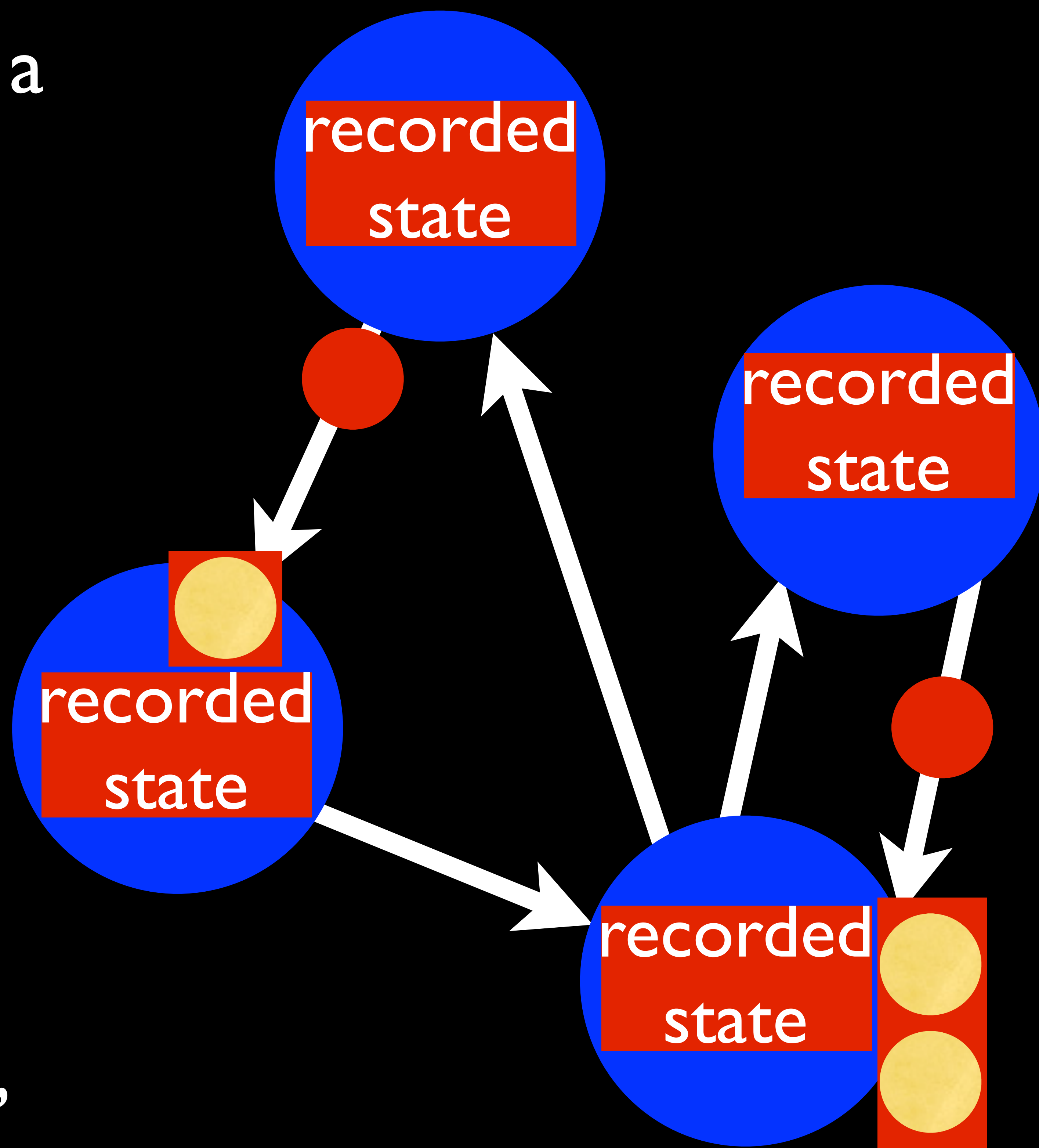


The top process on the right receives a marker, records its own state at that time, and sends a marker on its outgoing channel.

The message the leftmost process sent has been received by the bottom process, but the bottom process has already received a marker along this channel, so this message does not need to be logged.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"

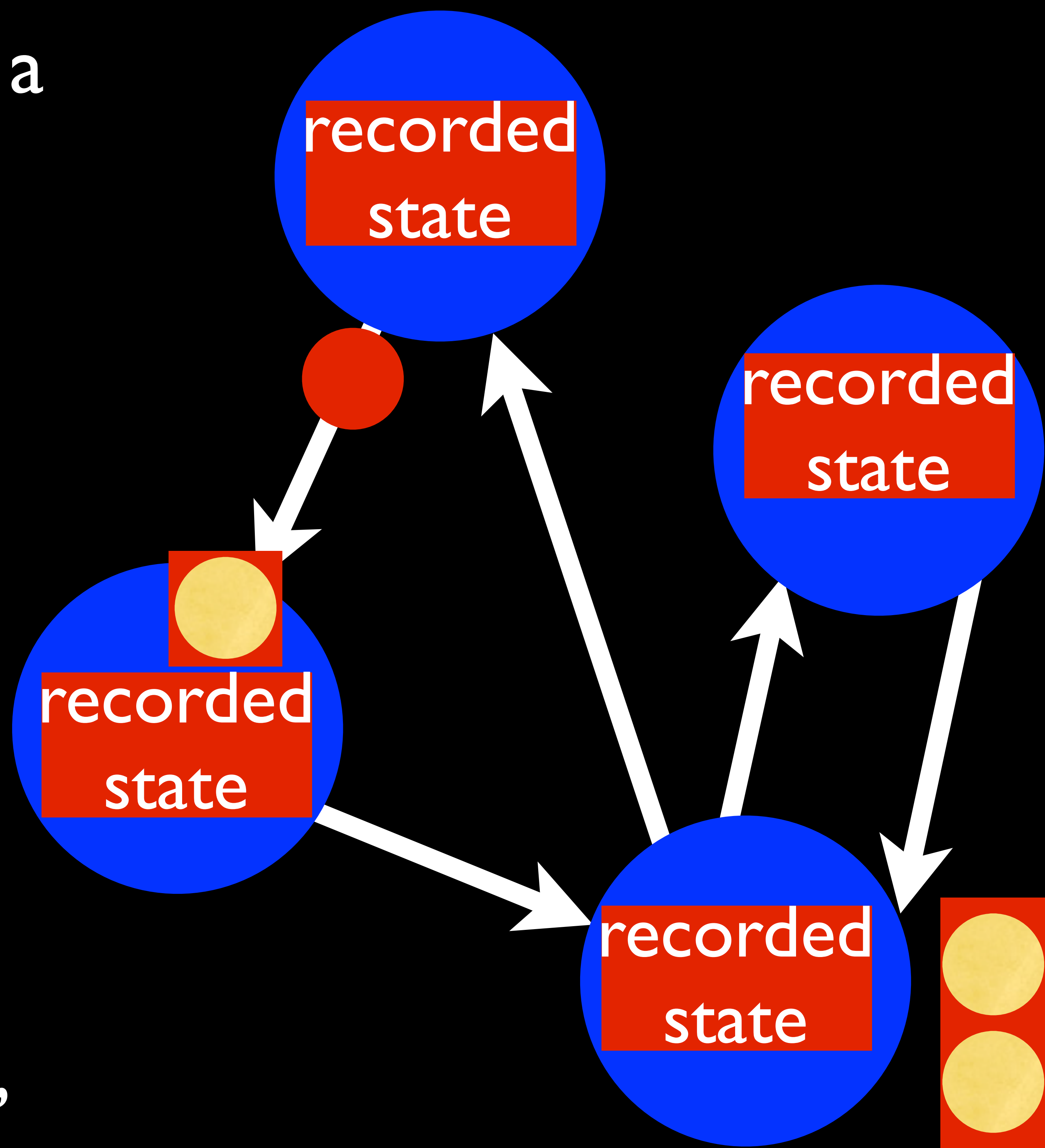


Monday, October 15, 12

The messages on the leftmost and rightmost channels are received, and logged.

Chandy - Lamport Snapshots

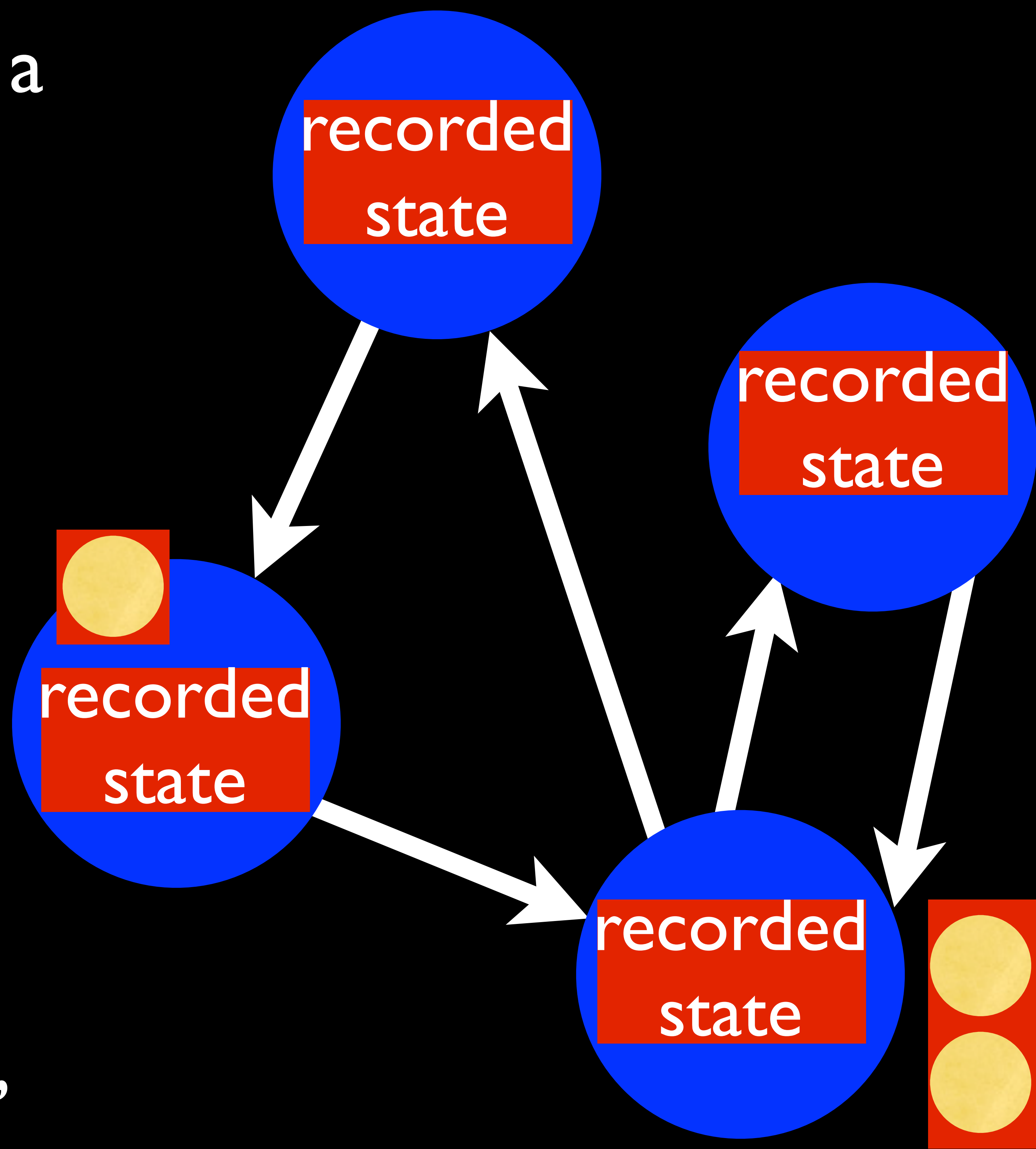
- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"



The marker on the rightmost channel is received, and so the bottom process can stop recording that channel. The log is set aside.

Chandy - Lamport Snapshots

- When a process receives a "marker" message:
  - record own state
  - send "marker" on all outgoing channels
  - record incoming channels until "marker"
- One or more processes begin by "pretending" they've received "marker"



Monday, October 15, 12

The marker on the leftmost channel is received, and so the leftmost process can stop recording that channel. The log is set aside.

At this point, each process has a recorded state and a log of all its incoming channels (channel logs not shown are considered empty), and together these are a snapshot. How and if these are assembled into a complete piece of information is a matter for another algorithm.

Snapshots are Consistent Cuts

- Proof by contradiction: Assume we have a snapshot that isn't a consistent cut.
- There exists a→b such that a is after the snapshot state recording on its process, and b is before the snapshot state recording on its process.
  - can't be on the same process
  - can't be send/receive of same message (FIFO)
  - can't be some chain of local and send/receive pairs
- Contradiction!

The Process states recorded by a snapshot form a consistent cut. We can prove this by contradiction.

Suppose we have a snapshot that isn't a consistent cut. This would mean that there exist two events a and b such that a happens before b, by Lamport's Definition, but on their respective processes, a is after the snapshot state recording and b was before it.

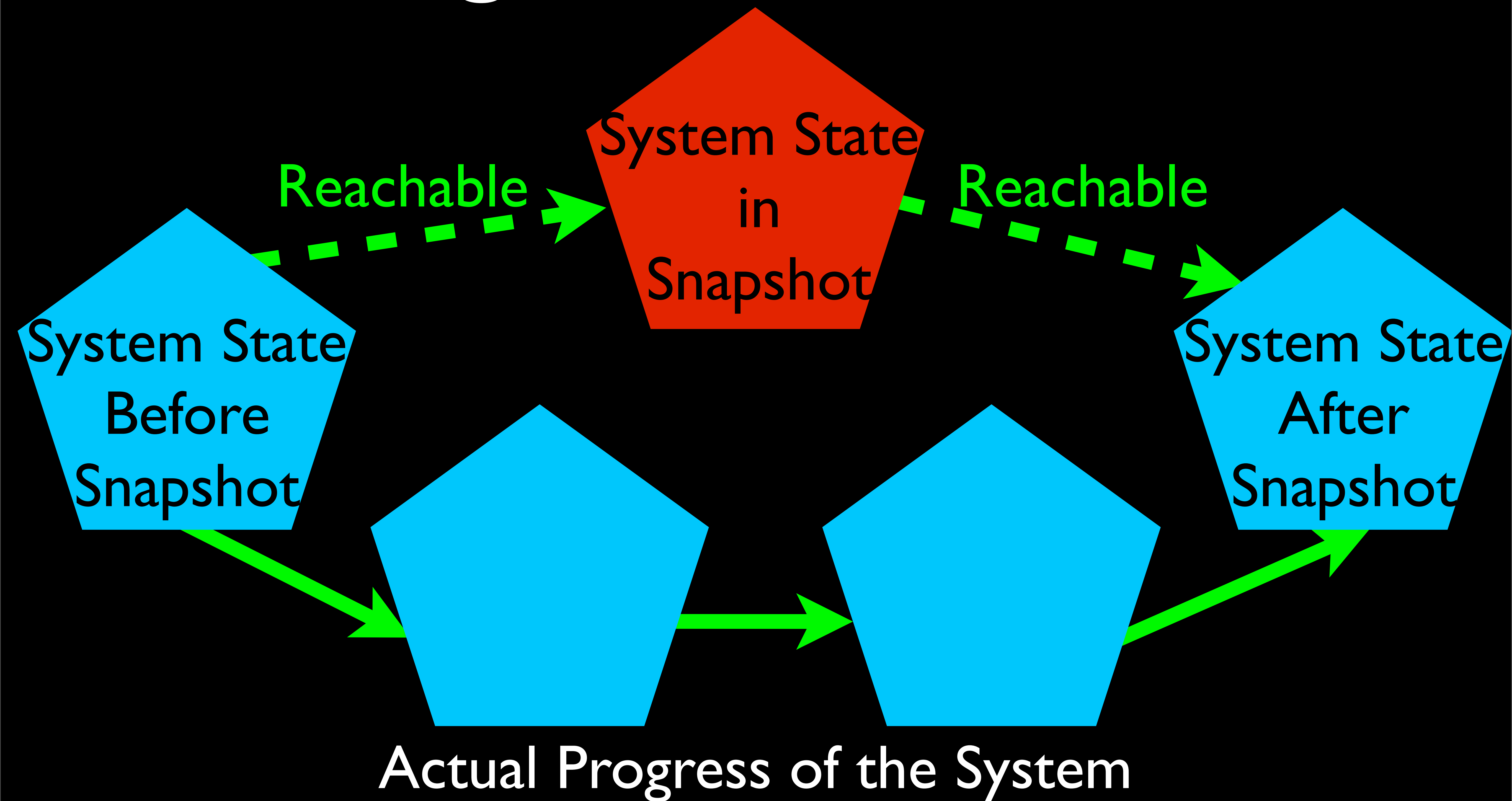
If a and b are on the same process, then if a was after the recording and b was before it, then b was before a, so a couldn't happen before b. Obviously, they can't be on the same process.

If a and b are the send and receive events of a message, then a was sent after the snapshot was recorded on its process. When that process recorded its state, it also sent out markers on all outgoing channels. The channels are FIFO, so the marker would arrive before whatever message was sent at event a would arrive. Therefore, the process to which the message was headed would have recorded its state (when it received the marker) before the receipt of a's message. The receive event b would therefore occur after the state was recorded, so a and b can't be a send and receive pair.

If no events on the same process can do it, and no send-receive pair can do it, then no sequence of events on the same process, sends, and receives can do it either. Some sequential pair in that sequence would have to be either a pair of local events or a send/receive of a message that would go from after the snapshot to before it.

Therefore, there can exist no a→b such that a is after the snapshot and b is before it, which makes the snapshot a consistent cut.

What's more, snapshots include a record of all messages received on all channels between the time when a snapshot state was recorded at the destination process and on the sending process, forming a record of all messages "in transit" at the "time of the snapshot."



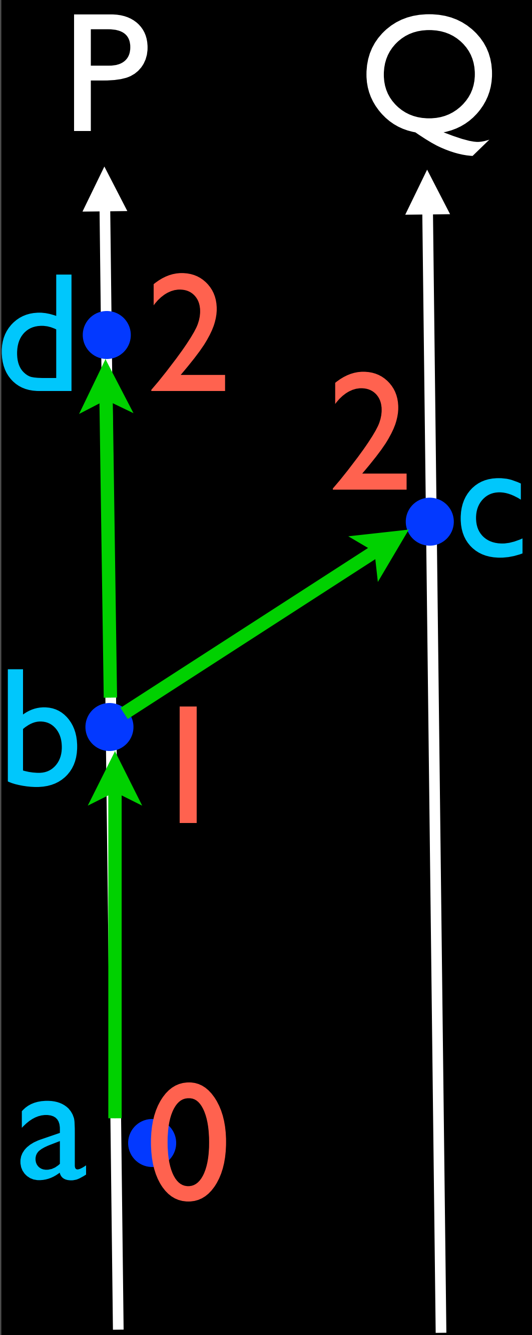
Monday, October 15, 12

Snapshots don't represent the system at any particular "real time"

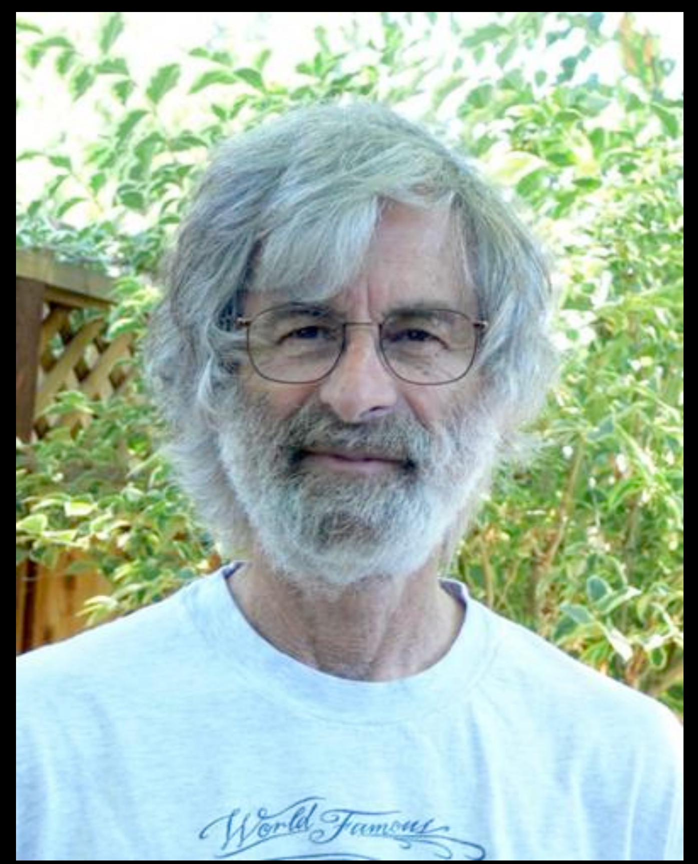
Consider the “Reachable” relation. One system state is reachable from another if you can start in one state and transition to the other via some series of legal system transitions (local events, message send/receives, etc. in line with the rules of this particular computation).

Snapshots do represent a total system state reachable from the system before the start of the algorithm, from which the system state after completion of the algorithm is reachable. It's a "possible past"

Summary



- Logical Clocks
  - order events
  - Few model assumptions
- Snapshots
  - form consistent cuts
  - detect stable properties
  - state machine processes
  - reliable FIFO channels


Leslie Lamport

K. Mani Chandy

Monday, October 15, 12

You people had better have questions.