About These Slides

These slides were developed by

Prof. Jennifer Welch
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
welch@cs.tamu.edu

for use with the book *Distributed Computing: Fundamentals, Simulations and Advanced Topics* by Hagit Attiya and Jennifer L. Welch, McGraw-Hill, 1998.

Distributed Systems

Distributed systems have become ubiquitous:

- share resources
- communicate
- increase performance
  - speed
  - fault-tolerance

Characterized by

- independent activities (concurrency)
- loosely coupled parallelism (heterogeneity)
- inherent *uncertainty*

Uncertainty comes from

- differing processor speeds
- varying communication delays
- (partial) failures
- multiple input streams and interactive behavior

Reasoning about Distributed Systems

This uncertainty makes it difficult to reason about distributed systems, to be confident that they are correct.

Attack this problem with the *theoretical paradigm*:

- identify and abstract fundamental problems
- state problems precisely
- design algorithms to solve problems
- prove correctness of algorithms
- analyze complexity of algorithms (e.g., time, space, messages)
- prove impossibility results and lower bounds

Potential payoff:

- careful specifications clarify intent
- increased confidence in correctness
- good abstraction is relevant in multiple situations
- indicate inherent limitations

Application Areas

Some application areas that have provided classic problems in distributed computing:

- operating systems
- distributed databases
- software fault-tolerance
- communication networks
- multiprocessor architectures

Course Overview — Part I (Fundamentals)

Introduce two basic communication models:

- message passing
- shared memory

and two basic timing models:

- synchronous
- asynchronous

	MP	SM
synch	yes	no
asynch	yes	yes

(Synchronous SM model, PRAM)

Part I (cont'd)

Cover the canonical problems and issues:

- graph algorithms (Ch 2)
- leader election (Ch 3)
- mutual exclusion (Ch 4)
- fault-tolerant consensus (Ch 5)
- causality and time (Ch 6)

Course Overview — Part II (Simulations)

Here "simulations" means *abstractions*, or techniques for making it easier to program, by making one model appear to be an easier model. For example:

- broadcast and multicast (Ch 8)
- distributed shared memory (Ch 9)
- stronger kinds of shared read/write objects (Ch 10)
- more synchrony (Ch 11)
- more benign faults (Ch 8, 10, 12)

For each technique:

- how to implement it
- an application that uses the technique

Course Overview — Part III (Advanced Topics)

Push further in some directions already introduced.

- randomization (Ch 14)
- stronger kinds of shared objects of arbitrary type (Ch 15)
- bounded timestamps (Ch 16)
- asynchronous solvability (Ch 17)
- sparse network covers (Ch 18)

Relationship of Theory to Practice

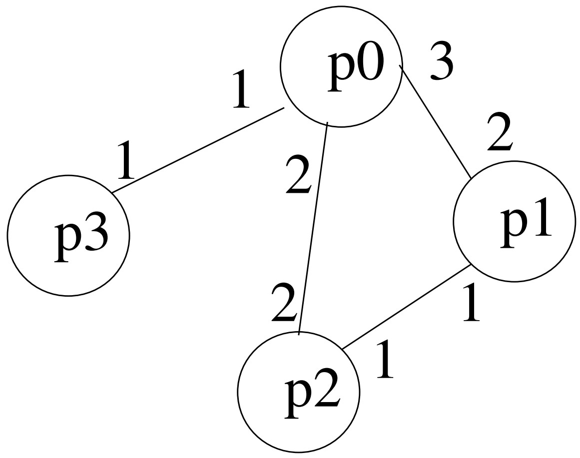
- time-shared operating system: issues relating to (virtual) concurrency of the processes, such as
  - mutual exclusion
  - deadlock

  also arise in physically distributed systems, especially asynchronous ones
- multiprocessors (MIMD):
  - if no common clock then asynchronous model fits
  - if a common clock then synchronous model fits
- loosely coupled networks, such as the Internet: the asynchronous message passing model fits

Failure models:

- crash: fits many actual situations
- Byzantine (arbitrary): conservative assumption, fits when failure mode is unknown or malicious

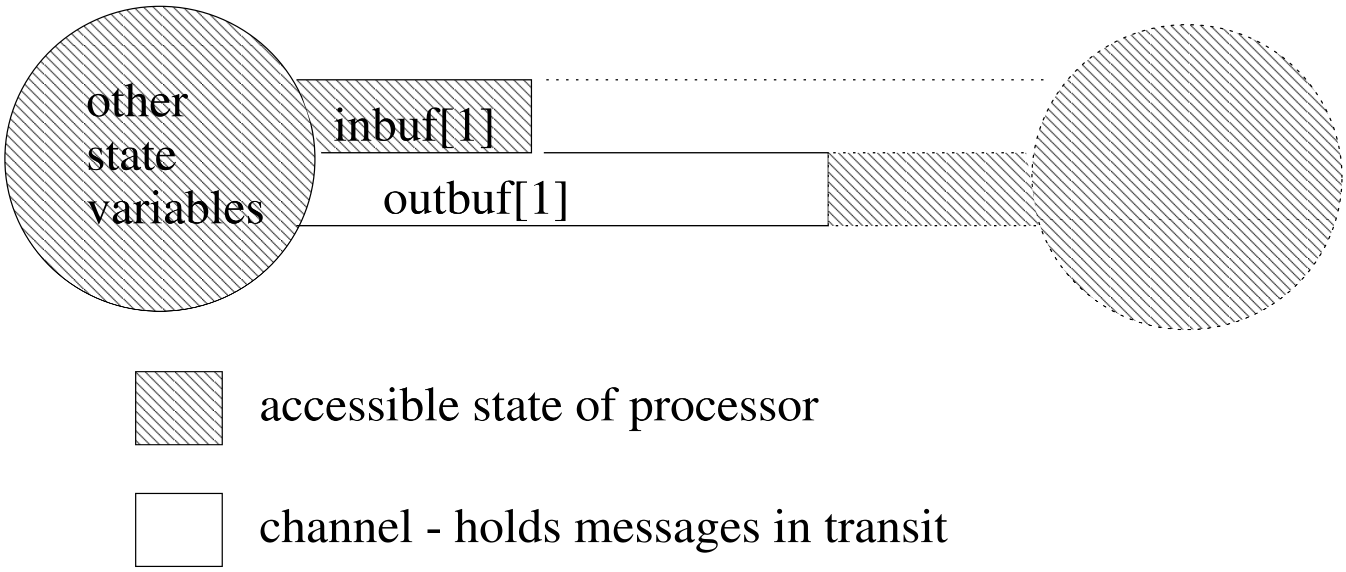
Message-Passing Model



- processors are $p_0, p_1, \ldots, p_{n-1}$ (nodes of graph)
- bidirectional point-to-point channels (undirected edges of graph)
- each processor labels its incident channels $1, 2, 3, \ldots$; might not know who is at other end

Processors, Channels and Configurations

processor is a state machine:



Configuration: vector of processor states, one per processor. Captures current snapshot of entire system.

Events

There are two kinds of **events**:

1. *deliver event*: moves a message from sender's outbuf to receiver's inbuf; message will be processed next time receiver does a computation event
2. *computation event*: a step by a processor

old accessible state

computation event: transition of processor's state machine; handles all incoming messages

new accessible state w/ empty inbufs

+

outgoing messages

Asynchronous Execution

Execution:

config, event, config, event, ..., config, event, config, ...

- in first config: each processor is in initial state (specified by its algorithm) and all inbufs are empty
- for each (config, event, config) triple: new config is same as old config except:
  - if delivery event: specified message is transferred from sender's outbuf to receiver's inbuf
  - if computation event: specified processor's state (including outbufs) change according to its transition function

Admissibility

Definition of execution specifies **safety** conditions that must hold: things that must be true in every finite prefix. "Nothing bad ever happens."

Sometimes we want to impose **liveness** conditions: things that must happen a certain number of times, perhaps infinitely often. "Eventually something good happens."

Executions that also satisfy the appropriate liveness conditions are called *admissible* executions.

Admissible executions are the ones that should "do the right thing."

For the asynchronous model, admissible means:

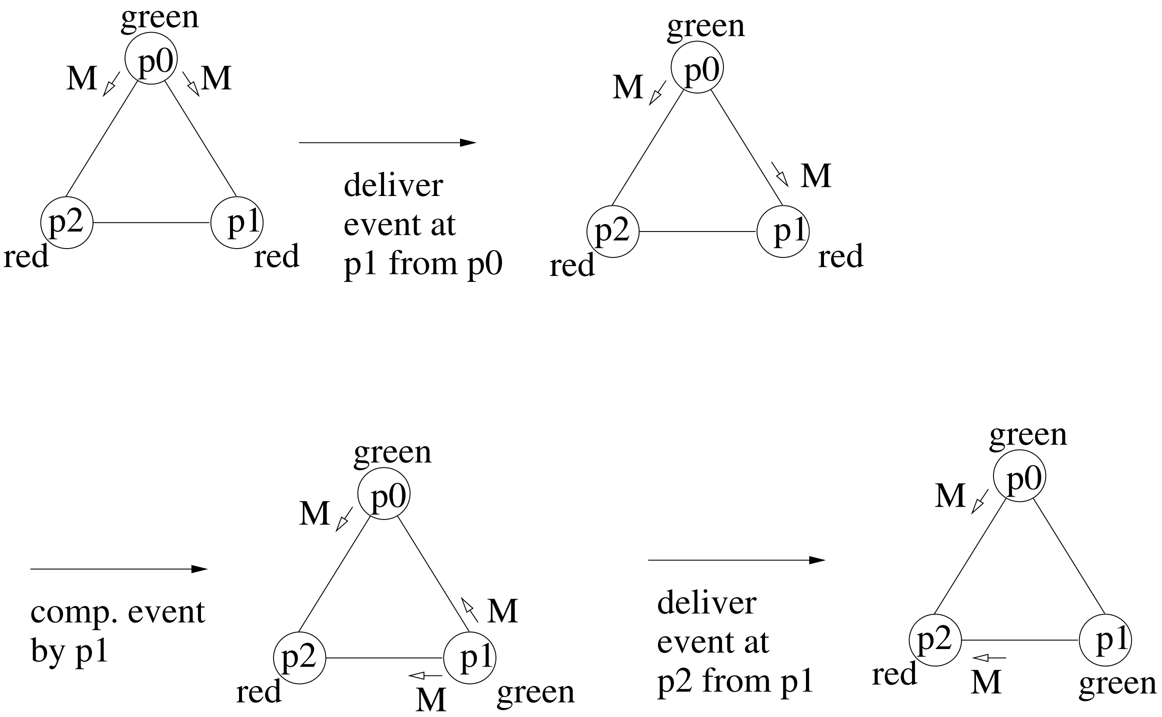
- every message in an outbuf is eventually delivered
- every processor takes an infinite number of steps

Notice there is no constraint on *when* these events must take place. Thus arbitrary message delays and relative processor speeds are not ruled out.

Example: Flooding

Describe a simple flooding algorithm as a collection of state machines. Consider processor p_i.

- *states:* variable "color" that holds green or red; the required inbufs and outbufs
- *initial states:* for p_0, color is green and all outbufs contain M; for others, color is red and outbufs are empty.
- *transitions:* if M is in an inbuf and color is red, then change color to green and send M on all outbufs



Termination

For technical reasons, admissible executions are defined as infinite.

But often algorithms terminate.

To model algorithm termination, identify *terminated* states of processors — states which, once entered, are never left.

An execution has **terminated** when all processors are terminated and no messages are in transit (in inbufs or outbufs).

Complexity Measures

These are *worst-case.*

Message Complexity: maximum number of messages sent in any *admissible* execution.

Time Complexity: (first attempt) maximum time until termination in any *admissible* execution.

But how is time measured?

Produce a **timed execution** from an execution by assigning non-decreasing real times to the events in an execution such that *the time between the sending and receiving of any message is at most 1*.

This essentially normalizes so that the greatest message delay in an execution is one time unit, yet still allows arbitrary interleavings.

Time Complexity: maximum time until termination in any *timed* admissible execution.

Complexity Measures of Flooding Algorithm

Define terminated states to be those when the color is green.

The number of messages sent in every admissible execution is $2 * m$, where m is the number of edges — one message is sent over each edge in each direction.

The maximum time until termination is $diam + 1$, where $diam$ is the diameter of the graph.

Synchronous Message Passing Systems

In the synchronous model, processors operate in lock-step rounds.

To model this, define an **execution** to be a series of rounds.

A **round** looks like this:

deliver ... deliver comp comp ... comp

First, deliver all messages in outbufs.
Then $p_0, p_1, \ldots, p_{n-1}$ take one computation step each.

An **admissible** execution is an execution that is infinite. This definition implies:

- every message sent is delivered
- every processor takes an infinite number of step

Time is measured in synchronous execution as the number of rounds until termination.

Example of Synchronous System

If the flooding algorithm is executed in the synchronous model on the triangle:
round 1:

- deliver M to p_1 from p_0
- deliver M to p_2 from p_0
- p_0 does nothing
- p_1 turns green and sends M to p_0 and p_2
- p_2 turns green and sends M to p_0 and p_1

round 2:

- deliver M to p_0 from p_1 and p_2
- deliver M to p_1 from p_2
- deliver M to p_2 from p_1
- p_0, p_1, and p_2 do nothing

Time and message complexities are the same as in the asynchronous case ($D + 1$ and $2m$).

Broadcast over a Known Rooted Spanning Tree

Suppose each processor has local variables "parent" and "children" that are initialized to indicate which incident channels lead to parent and children in a spanning tree of the graph.

Each processor also has a boolean variable "terminated".

- root initially sends M to its children
- when a processor receives M from its parent, it sends M to its children and terminates.

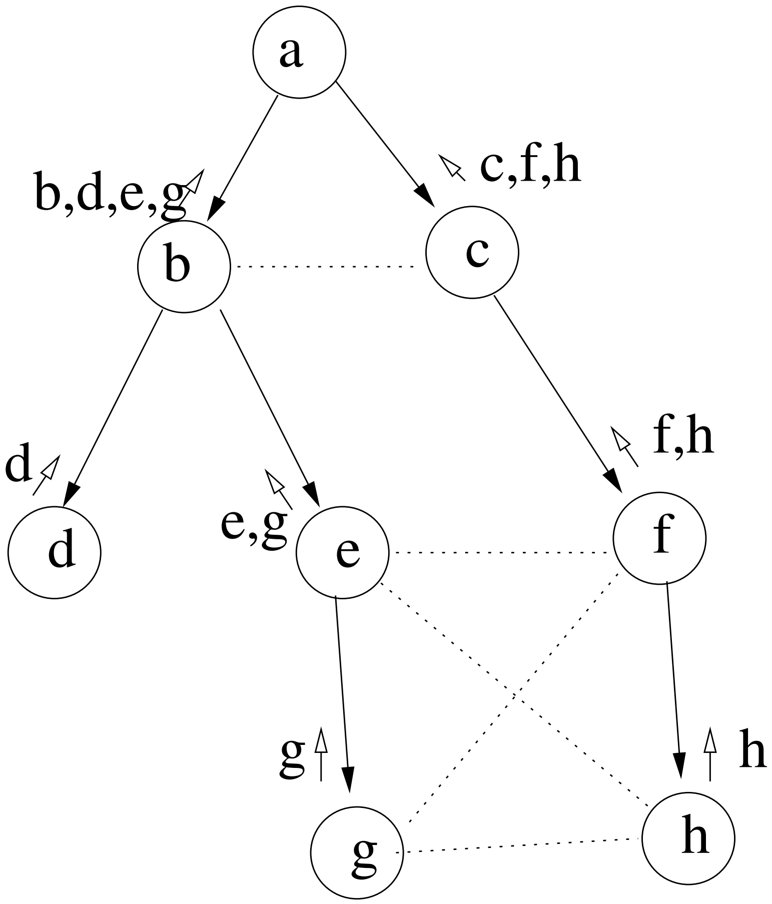
Complexities in both synchronous and asynchronous models:

- Time: d, depth of the spanning tree (no more than $n-1$)
- Messages: $n-1$, one message is sent over each edge in the spanning tree

Convergecast over a Known Rooted Spanning Tree

Opposite of the previous broadcast, with same complexities.

- leaves send message with their info to their parents
- non-leaves wait to get message from each child, then combines info and sned it to its parent



Finding a Spanning Tree When a Root is Known

Having a spanning tree is very convenient. How do you get one?

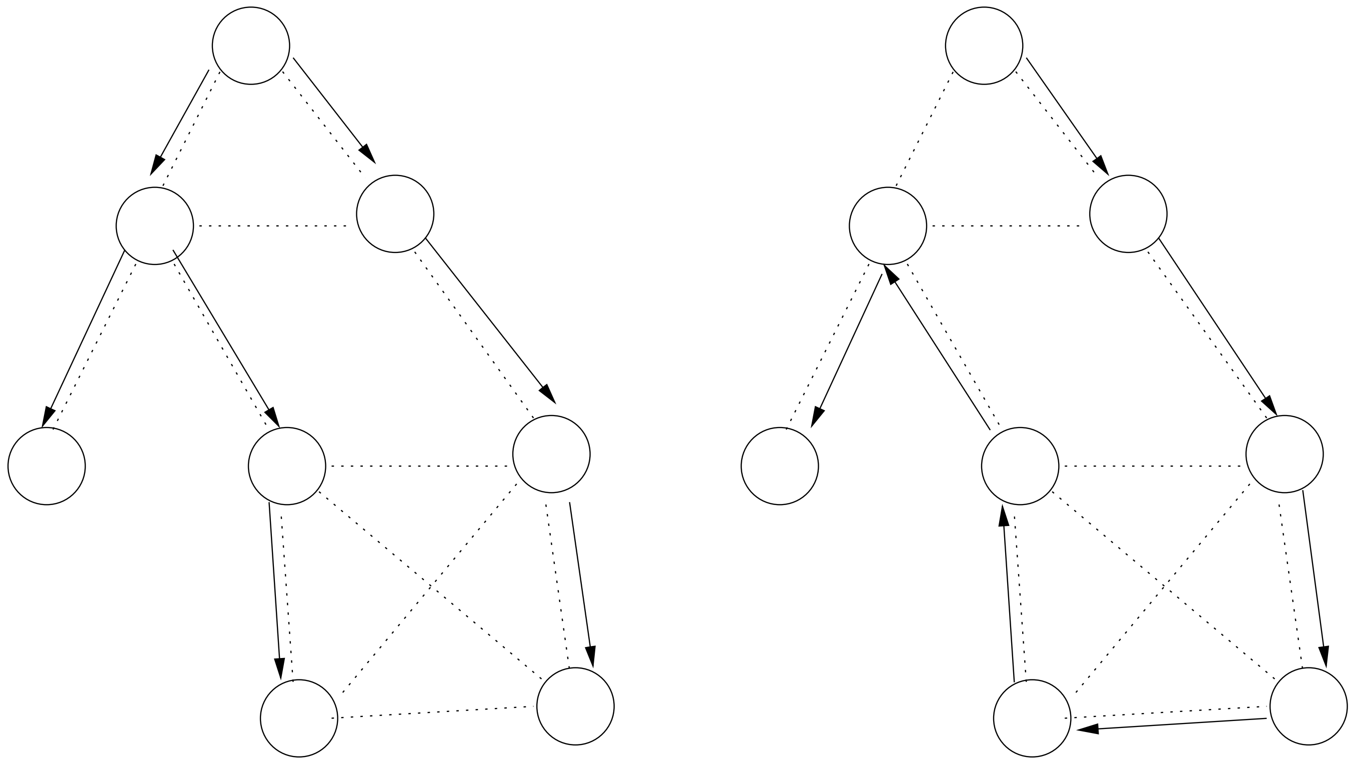
First, suppose a distinguished processor is known, who can serve as the root.

Modify the flooding algorithm:

- root sends M to all its neighbors
- when non-root *first* gets M, it sets the sender as its parent, sends "parent" to sender, and sends M to all other neighbors
- when a processor gets M otherwise, it sends "reject" to sender
- processors use parent and reject messages to set their children variables and know when to terminate

Execution of Spanning Tree Algorithm

In the synchronous case, it produces a **breadth-first search** (BFS) tree.
In the asynchronous case, it might not produce a BFS tree:



In both cases, it uses $O(m)$ messages and $O(diam)$ time.

Finding a DFS Spanning Tree When Root is Known

Another kind of spanning tree is a **depth-first search**.

- when root first takes step or non-root first receives M:
  - mark sender as its parent (if not root)
  - for each neighbor
    send M to it
    wait to get "parent" or "reject" in reply
  - send "parent" to sender (if not root)
- if processor receives M in any other case, then send "reject" to sender
- processors use parent and reject messages to set their children variables and know when to terminate

Analogous to sequential DFS. Ensures that you get the second figure on previous slide.

Message complexity and time complexity are both $O(m)$.

Finding a Spanning Tree Without a Specified Root

Assume the processors have *unique identifiers* (otherwise impossible!).

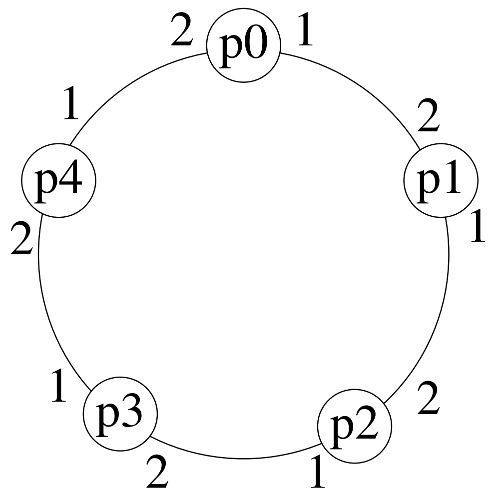
Idea: Each processor starts running a copy of the previous DFS ST algorithm, using its id to differentiate its copy. When copies collide, the copy with the larger id wins.

Message complexity is $O(n \cdot m)$ and time complexity is $O(m)$.

See text for details.

Rings

In an *oriented ring*, processors have a consistent notion of left and right:



1 = left = clockwise

2 = right = counter-clockwise

For example, if messages are always forwarded on incident channel 1, they will cycle clockwise around the ring.

Why study rings?

- simple starting point, easy to analyze
- abstraction of a token ring
- lower bounds for ring topology also apply to arbitrary topologies

Leader Election

Definition:

- each processor has a set of **elected** states and a set of **not-elected** states. Once an elected state is entered, the processor always is in an elected state; similarly for non-elected. I.e., irreversible decision.

- In every *admissible* execution,
  - every processor eventually enters either an elected or a not-elected state (liveness)
  - eactly one processor (the **leader**) enters an elected state (safety)

A leader can be used to coordinate future activities of the system. For instance:

- find a spanning tree using the leader as the root
- reconstruct a lost token for a token-ring

We will study leader election in rings.

Anonymous Rings

Intuition is that processors do not have unique identifiers.

Related issue is whether an algorithm A relies on processors *knowing the ring size*.

- **uniform** algorithm — does not use the ring size (same algorithm for each size ring)

  Formally, every processor in every size ring is modeled with the same state machine A.

- **non-uniform** algorithm — does use the ring size (different algorithm for each size ring; may be only trivially different)

  Formally, for every value of n, there is a state machine A_n such that every processor in a ring of size n is modeled with A_n. Thus A is the collection of all the A_n's.

Leader Election in Anonymous Rings

Theorem 3.2: There is *no* leader election algorithm for anonymous rings, even if the algorithm knows the ring size (i.e., is non-uniform) and the ring is synchronous.

Proof Sketch:

- Every processor begins in the same state with the same messages originally in transit.
- Every processor receives the same messages and thus makes the same state transition and sends the same messages in round 1.
- Every processor receives the same messages and thus makes the same state transition and sends the same messages in round 2.
- Etc.

Eventually some processor is supposed to enter an elected state. But then they all would, a contradiction.
□

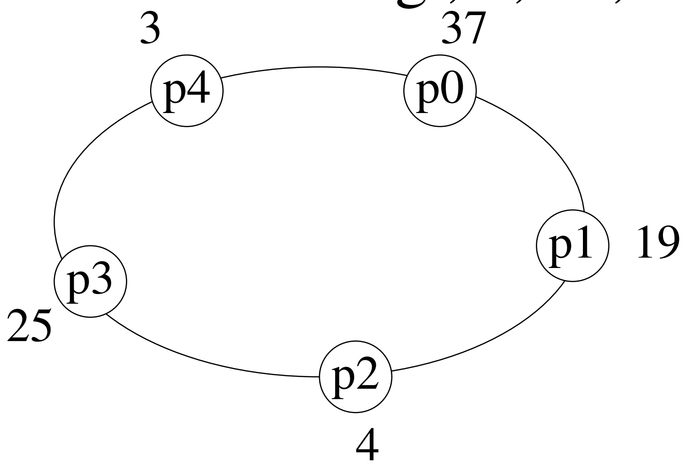
Consequently, there is no uniform or asynchronous leader election algorithm.

Rings with Identifiers

Assume each processor has a unique identifier. Distinguish between indices and identifiers:

- **indices** are 0 through $n - 1$ and are *unavailable* to the processors; used only for analysis
- **identifiers** are arbitrary nonnegative integers and are available to the processors via a special state component called **id**.

Specify a ring by starting with the smallest id and listing ids in clockwise order. E.g., 3, 37, 19, 4, 25.



Uniform algorithm: There is one state machine for every id, no matter what size ring.

Non-uniform algorithm: There is one state machine for every id and every different ring size.

Overview of Leader Election in Rings with Ids

In this case, there are algorithms. We will evaluate them according to their **message complexity**.

Overview of Upcoming Results:

- asynchronous ring: $\Theta(n \log n)$ messages
- synchronous ring:
  - $\Theta(n)$ messages under certain conditions
  - otherwise $\Theta(n \log n)$ messages

All bounds are asymptotically tight.

$O(n^2)$ Messages Leader Election Algorithm

Each processor follows these rules:

- Initially send your id to the left
- When you receive an id (from the right):
  - if it is greater than your id then forward it to the left (you will never be the leader)
  - if it is equal to your id then elect yourself leader
  - if it is smaller than your id then do nothing

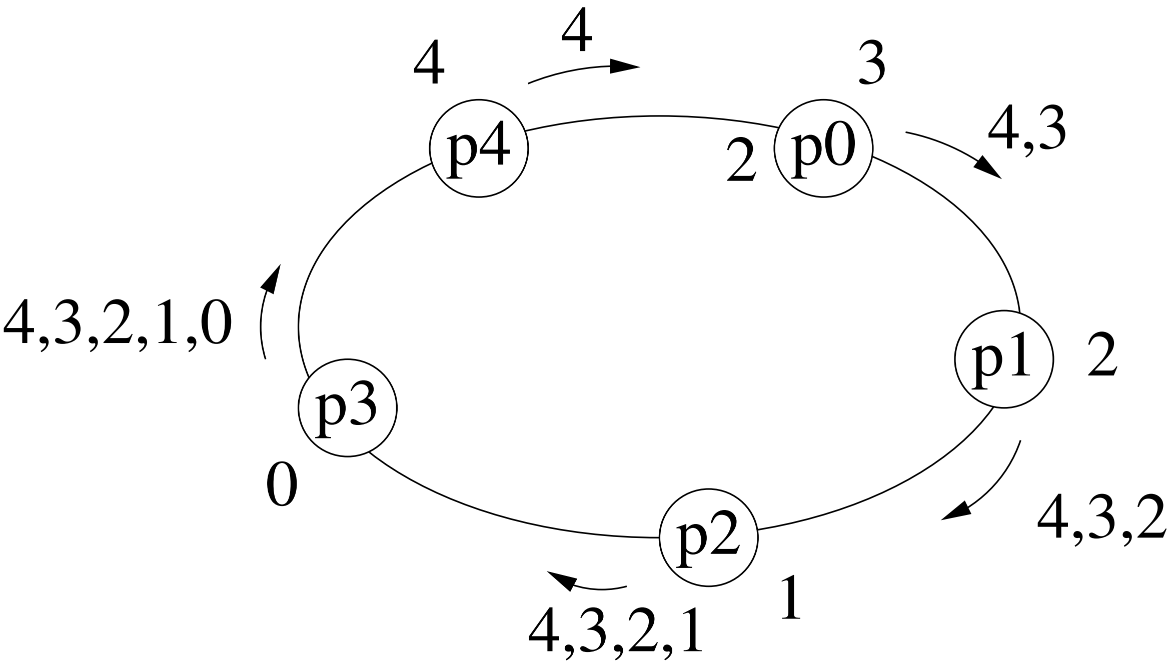
Correctness: Elects processor with largest id. Message containing that id passes through every processor.

Message complexity: Depends how ids are arranged.

- Largest id travels all around the ring, n messages
- Second largest id travels until reaching largest
- Third largest id travels until reaching largest or second largest
- Etc.

$O(n^2)$ Messages Algorithm (cont'd)

Worst way to arrange the ids is in decreasing order:



- Second largest id contributes $n - 1$ messages.
- Third largest id contributes $n - 2$ messages.
- Etc.

Total number of messages is

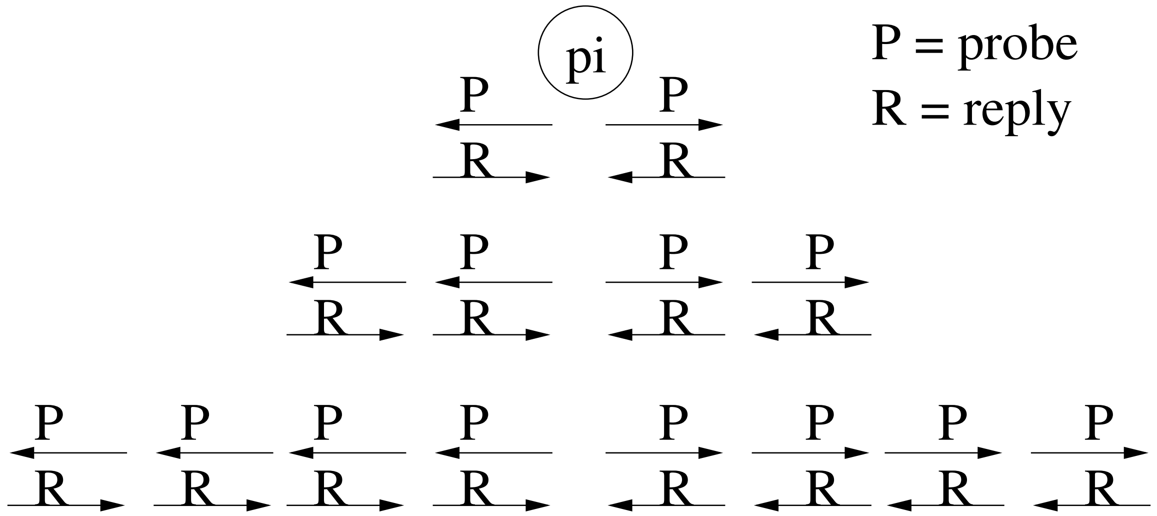
$$\sum_{i=1}^{n} i = \Theta(n^2).$$

$\square$

$O(n \log n)$ Messages Leader Election Algorithm

- Each processor tries to probe successively larger neighborhoods. Size of neighborhood doubles in each *phase*.
- A probe is initiated by sending a probe message containing the initiator's id.
- If a processor receives a probe message whose id is larger than its own id, the processor will either forward it on or send back a reply, as appropriate.
- If a processor receives a probe message whose id is smaller than its own id, it does nothing.
- If a processor receives a probe message with its own id, then it becomes the leader.
- If a processor receives a reply message not destined for itself, it forwards it.
- If a processor receives both reply messages destined for itself, it proceeds to the next phase.

$O(n \log n)$ Messages Algorithm (cont'd)



Correctness: Similar to previous algorithm.

Message Complexity:

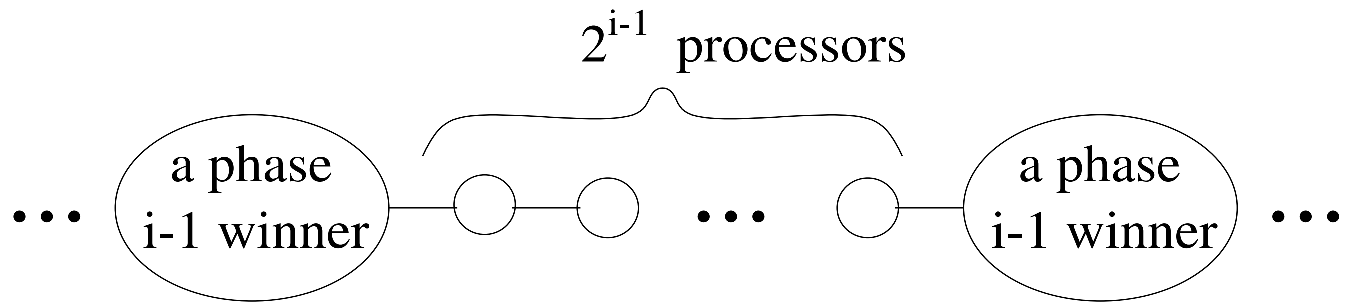
- Each message belongs to a phase and is initiated by a particular processor.
- Probe distance in phase i is 2^i.
- The number of messages initiated by a particular processor in phase i is $4 \cdot 2^i$ (probes and replies in both directions).

$O(n \log n)$ Messages Algorithm (cont'd)

How many processors initiate probes in phase i?

- For $i = 0$, all n of them do.
- For $i > 0$, every processor that is a "winner" in phase $i - 1$ does (has largest id in its 2^{i-1} neighborhood)

Maximum number of phase $i - 1$ winners occurs when they are packed as densely as possible:



Total number of phase $i - 1$ winners is at most

$$\frac{n}{2^{i-1} + 1}$$

How many phases are there? Phases continue until there is only one winner, so $\log n$ phases suffice.

$O(n \log n)$ Messages Algorithm (cont'd)

Total number of messages is

$$\begin{aligned}
&\leq 4 \cdot n + \sum_{i=1}^{\log n} 4 \cdot 2^i \cdot \frac{n}{2^{i-1}+1} + n \\
&\leq 5n + 4n \sum_{i=1}^{\log n} \frac{2^i}{2^{i-1}} \\
&= 5n + 8n \log n \\
&= O(n \log n).
\end{aligned}$$

□

Asynchronous Lower Bound on Messages

Consider any leader election algorithm A that

1. works in an asynchronous ring
2. is uniform
3. elects maximum id
4. guarantees everyone learns the id of the winner

We will show message complexity is $\Omega(n \log n)$.

- Condition 1 is necessary for the lower bound to hold.
- Condition 2 is necessary for this particular proof to work.
- Conditions 3 and 4 are made without loss of generality: any algorithm that does not satisfy these two conditions can be converted into one that does with $O(n)$ additional messages.

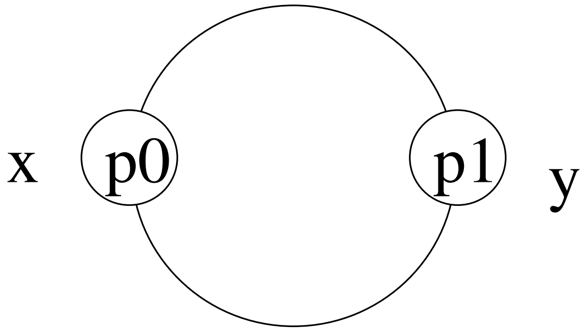
Asynchronous Lower Bound on Messages (cont'd)

Theorem 3.5: For every n that is a power of 2 and every set S of n ids, there is a ring using those ids on which algorithm A has an *open* schedule in which at least $M(n)$ messages are sent, where

- $M(2) = 1$ and
- $M(n) = 2 \cdot M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$, $n > 2$.

A schedule is **open** if there is an edge over which no message is delivered. (An open schedule is not admissible, but it is a prefix of an admissible schedule).

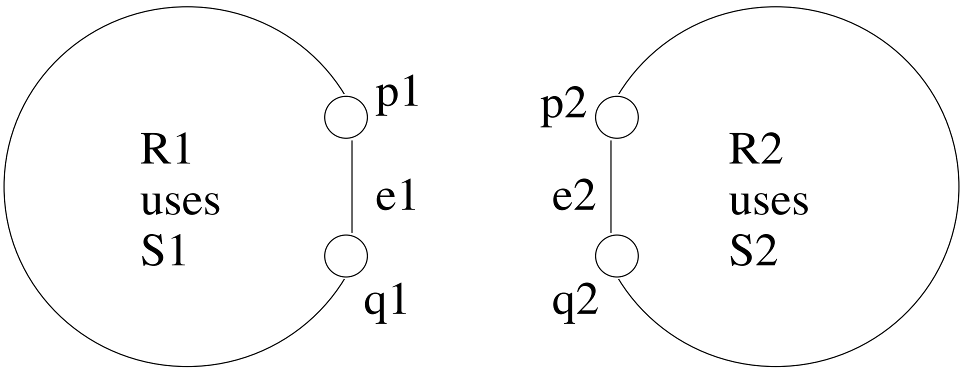
Proof: By induction on n.
Basis: $n = 2$.



Suppose $x > y$. At some point p_0 must send a message to p_1 so that p_1 can learn x. Truncate immediately after sending of first message to get desired schedule.

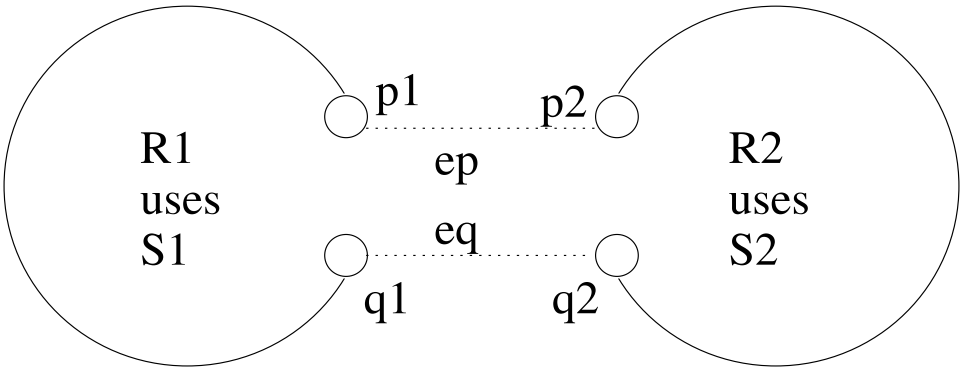
Asynchronous Lower Bound on Messages (cont'd)

Induction: $n \geq 4$. Split S into two halves, S_1 and S_2. By inductive hypothesis, there are rings:



- R_1 has open schedule α_1 in which at least $M(n/2)$ messages are sent and $e_1 = (p_1, q_1)$ is open edge
- R_2 has open schedule α_2 in which at least $M(n/2)$ messages are sent and $e_2 = (p_2, q_2)$ is open edge

Paste R_1 and R_2 together to make a big ring R:



Asynchronous Lower Bound on Messages (cont'd)

To build an execution of R with $M(n)$ messages:

- Execute α_1: Processors on left cannot tell difference between being in R_1 and being in R, so they will behave the same and send $M(n/2)$ messages in R.
- Then execute α_2: Similarly processors on right will send $M(n/2)$ messages in R.

Note crucial dependence on uniform assumption!

Case 1: Without unblocking e_p or e_q, there is an extension of $\alpha_1\alpha_2$ on R in which an additional $\frac{1}{2}(\frac{n}{2} - 1)$ messages are sent. This is the desired schedule.

Case 2: Without unblocking e_p or e_q, every extension of $\alpha_1\alpha_2$ on R leads to *quiescence*: no processor will send another message unless it receives one and no messages are in transit except on e_p and e_q.

- Let α_3 be the schedule extending $\alpha_1\alpha_2$ that causes the system to become quiescent.

Asynchronous Lower Bound on Messages (cont'd)

- Let $\alpha 4''$ be a schedule extending $\alpha_1\alpha_2\alpha_3$ after which the algorithm has terminated.

Claim: At least $n/2$ messages are sent in α_4''.

Reason:

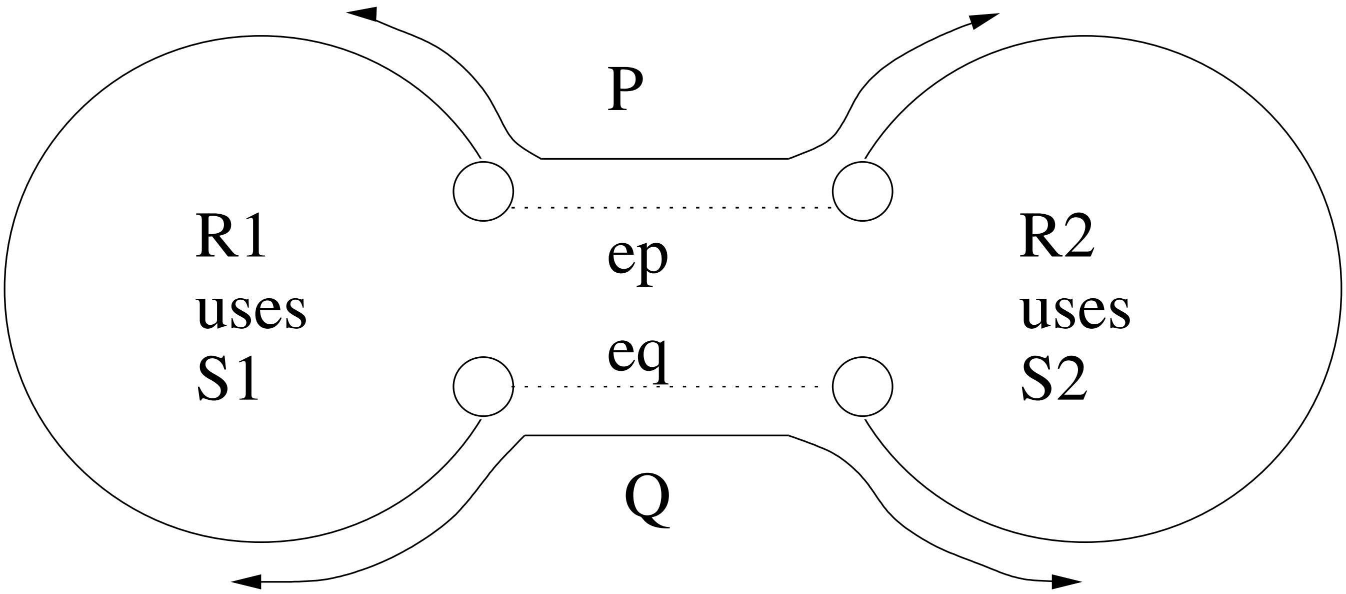
- Each of the $n/2$ processors in the half of R that does *not* contain the leader must receive a message to learn the leader's id.
- And until α'' there has been no communication between the two halves of R.

Note heavy reliance here on the assumptions that the max is elected and that all learn the leader's id!

However, $\alpha_1\alpha_2\alpha_3\alpha_4''$ is not our desired schedule since it might not be *open*.

Asynchronous Lower Bound on Messages (cont'd)

As messages in e_p and e_q are delivered in α_4'', processors "wake up" from the quiescent state and send more messages. The sets of awakened processors expand outward around e_p and e_q:



- Let α_4' be the prefix of α_4'' when $\frac{n}{2} - 1$ messages have been sent.
- P and Q cannot meet in α_4', since less than $\frac{n}{2}$ messages are sent in α_4'.
- WLOG, suppose the majority of the messages sent in α_4' are sent by processors in P.

Asynchronous Lower Bound on Messages (cont'd)

- Let α_4 be the sequence of events obtained from α_4' by keeping only those events involving processors in P. This schedule never delivers any messages over e_q!

Claim: In $\alpha_1\alpha_2\alpha_3\alpha_4$, processors in P behave the same as they do in $\alpha_1\alpha_2\alpha_3\alpha_4'$.

Reason: Since P and Q are disjoint, there is no communication between them and thus processors in P cannot tell whether or not processors in Q are active.

Note heavy reliance on the asynchrony assumption here!

Since the processors in P do the same thing, they still send $\frac{1}{2}(\frac{n}{2} - 1)$ messages in α_4.

Thus $\alpha_1\alpha_2\alpha_3\alpha_4$ is the desired schedule:

- $2 \cdot M(\frac{n}{2}) + \frac{1}{2}(\frac{n}{2} - 1)$ messages are sent,
- e_q is open.

□

Leader Election in Synchronous Rings

First, a simple algorithm for the synchronous model:

- Group the rounds into **phases** so that each phase contains n rounds.
- In phase i, the processor with id i, if there is one, sends a message around the ring and is elected.

Example: $n = 4$ and 7 is smallest id.

- In phases 0 through 6 (corresponding to rounds 1 through 28), no message is ever sent.
- At beginning of phase 7 (round 29), processor with id 7 sends message which is forwarded around ring.

Note reliance on synchrony and knowledge of n!

Correctness: Convince yourself.

Message Complexity: $O(n)$. Note that this is optimal.

Time Complexity: $O(n \cdot m)$, where m is the smallest id in the ring. Not bounded by n.

Another Synchronous LE Algorithm

This algorithm

- works in a slightly weaker model: Processors might not all start at same round; a processor either wakes up spontaneously or when first gets a message.
- is uniform (does not rely on knowing n).

Idea:

- A processor that wakes up spontaneously is *active*; sends its id in a *fast* message — 1 edge/round.
- A processor that wakes up when receiving a message is *relay*; never in the competition.
- A fast message becomes *slow* if it reaches an active processor — 1 edge/2^m rounds (m is msg id)
- Processors (active or relay) only forward a message whose id is smaller than any id this processor has seen so far (ignoring the id of relay processors).
- If a processor gets own id back, leader.

Analysis of Synchronous LE Algorithm

Correctness: Convince yourself that active processor with smallest id is elected.

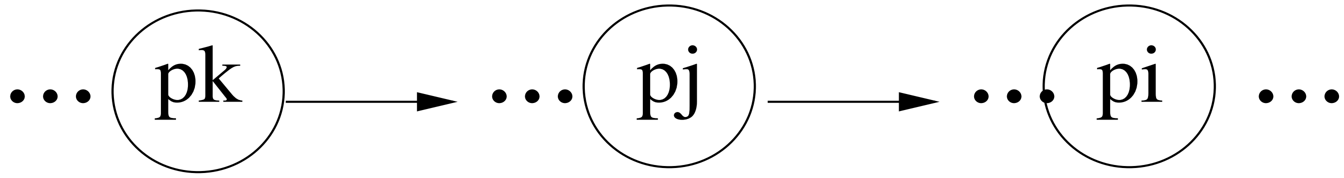
Message Complexity: Winner's message is the fastest. While it traverses the ring, other messages are slower, so they are overtaken and stopped before too many messages are sent.

More carefully, divide messages into three kinds:

1. fast messages
2. slow messages sent while the leader's message is fast
3. slow messages sent while the leader's message is slow

Analysis of Synchronous LE Algorithm (cont'd)

Number of type 1 messages (fast):
Show that no processor forwards more than one fast message.



If p_i forwards p_j's fast msg and p_k's fast msg, then when p_k's fast message arrives at p_j:

1. either p_j has already sent its fast message, so p_k's message becomes slow, or
2. p_j has not already sent its fast message, so it never will.

Number of type 1 messages is at most n.

Analysis of Synchronous LE Algorithm (cont'd)

Number of type 2 messages (slow while leader's is fast):

- Leader's message is fast for at most n rounds.
- Slow message i is forwarded $n/2^i$ times in n rounds.
- Worst case (largest number of messages) is when ids are as small as possible, 0 to $n-1$.

Number of type 2 messages is at most $\Sigma_{i=1}^{n-1} \frac{n}{2^i} \leq n$.

Number of type 3 messages (slow while leader's is slow):

- Once leader's message x becomes slow, it takes at most $n \cdot 2^x$ rounds to return to leader.
- No messages are sent once leader's message has returned to leader.
- Slow message i is forwarded $n \cdot 2^x/2^i$ times in $n \cdot 2^x$ rounds.
- Worst case is when ids are 0 to $n-1$ and $x = 0$.

Number of type 3 messages is at most $\Sigma_{i=0}^{n-1} n \cdot \frac{2^0}{2^i} \leq 2n$.
$\square$

Time Complexity of Synchronous LE Algorithms

Time Complexity: $O(n \cdot 2^x)$, where x is the minimum id. Even worse than the previous algorithm.

Both these algorithms have two potentially undesirable properties:

- rely on the numeric values of the ids to count
- number of rounds bears no relationship to n, but depends on the minimum id

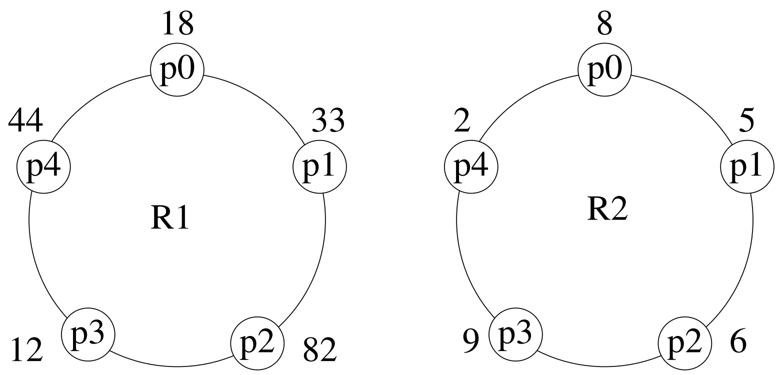
Next result shows that to obtain linear message complexity, an algorithm must rely on the numeric values of the ids.

(The book also shows that to obtain linear message complexity, an algorithm must have time complexity that depends on the values of the ids.)

Comparison-Based LE Algorithms

Definition: An LE algorithm is **comparison-based** if, in any two *order-equivalent* rings R_1 and R_2, *matching* processors p_i in R_1 and p_j in R_2 have *similar* behaviors in *exec*(R_1) and *exec*(R_2).

- Rings $R_1 = (x_1, x_2, \ldots, x_n)$ and $R_2 = (y_1, y_2, \ldots, y_n)$ are **order-equivalent** if $x_i < x_j$ iff $y_i < y_j$. Example:



- Processors p_i in R_1 and p_j in R_2 are **matching** if they are same distance from minimum id. Example: p_0 in R_1 and p_1 in R_2.
- Behaviors are **similar** if, in every round,
  - one processor sends a message to the left (right) iff the other one sends a message to the left (right)
  - one processor is elected iff the other one is.

Synchronous Message Lower Bound

Theorem 3.18: For every $n \geq 8$ that is a power of 2, there is a ring of size n on which any synchronous comparison-based algorithm sends $\Omega(n \log n)$ messages.

Proof: Construct, for each n, a highly symmetric ring S_n on which every comparison-based algorithm is expensive in terms of messages.

Symmetry means many processors have order-equivalent neighborhoods and thus do the same thing (in particular, send messages).

Technical complication: have to show that two processors with order-equivalent neighborhoods *on the same ring* behave similarly.

Does not immediately follow from the definition of comparison-based, which refers to matching processors on *different rings*.

Order-Equivalent Means Similar Behavior

Lemma 3.17: If R is *spaced* (at least n unused ids between any two used ids) and p_i and p_j have order-equivalent k-neighborhoods, then p_i and p_j have similar behaviors through the k-th *active* round.

A round is **active** if at least one processor sends a message in it.

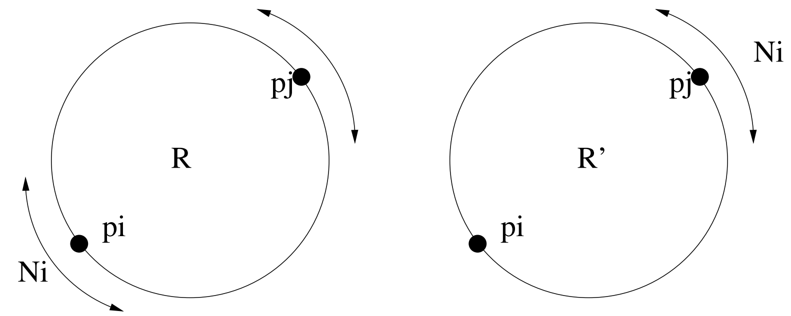
A processor in a synchronous algorithm can potentially learn something even in an inactive round, from the passage of time.

But in a comparison-based algorithm, it cannot learn about the order pattern of its ring in an inactive round.

Note that the k-th active round might be much much larger than the k-th round.

Order-Equivalent Means Similar Behavior

Proof Sketch for Lemma 3.17:



Construct R' such that

- p_j's k-neighborhood in R' *equals* p_i's in R
- ids in R' are unique
- R' and R are order-equivalent
- p_j in R' is matching to p_j in R

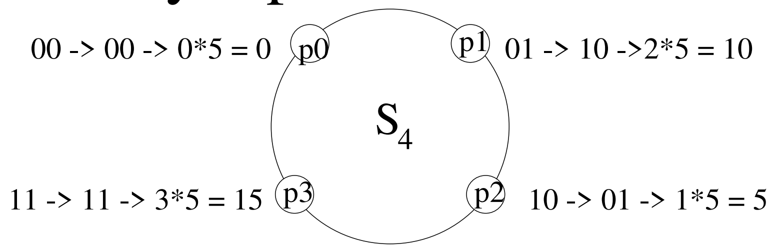
Can be done by spaced assumption (see text).

- p_i in *exec*(R) through k-th active round behaves *same* as p_j in *exec*(R'), since identical k-neighborhoods
- p_j in *exec*(R') through k-th active round behaves *similarly* to p_j in *exec*(R), since matching.

□

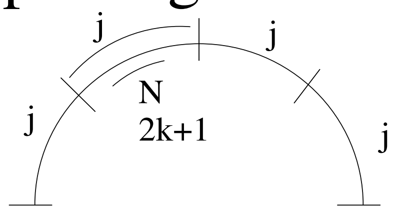
Highly Symmetric Ring S_n

p_i's id is $(n+1)\cdot$ *rev*(i), where *rev*(i) is the integer whose binary representation, using $\log n$ bits, is the reverse of i's binary representation. Example:



Lemma 3.21: For all $k < n/8$, for every k-neighborhood N of S_n, there are at least $\frac{n}{2(2k+1)}$ k-neighborhoods of S_n order-equivalent to N (including N).

Proof Sketch for Lemma 3.21: N is a sequence of $2k+1$ ids. Let j be smallest power of 2 that is larger than $2k+1$. Break S_n into $\frac{n}{j}$ segments of length j, with one segment encompassing N.



Claim: Each segment is order-equivalent to N.
Since $j < 2(2k+1)$, the number is more than $\frac{n}{2(2k+1)}$.
□

Synchronous Message Lower Bound (cont'd)

Lemma 3.20: Number of active rounds in $exec(S_n)$ is at least $n/8$.

Proof: Suppose T, number of active rounds, is less than $n/8$. Let p_i be elected leader.

- By Lemma 3.19, there are at least $\frac{n}{2(2T+1)}$ T-neighborhoods order-equivalent to p_i's.
- Since $n \geq 8$ and $T < n/8$, algebra shows that $\frac{n}{2(2T+1)} > 1$.
- Thus there exists $p_j \neq p_i$ whose T-neighborhood is order-equivalent to p_i's.
- By Lemma 3.17, p_j is also elected, contradiction.

$\square$

Synchronous Message Lower Bound (cont'd)

Lemma 3.21: At least $\frac{n}{2(2k+1)}$ messages are sent in k-th active round of *exec*(S_n), $1 \leq k \leq n/8$.

Proof: Since round is active, at least one processor, say p_i, sends a message.

- By Lemma 3.19, there are at least $\frac{n}{2(2k+1)}$ processors whose k-neighborhood is order-equivalent to p_i's.
- By Lemma 3.17, they each send a message in the k-th active round.

□

To finish proof of Theorem 3.18: number of messages sent in *exec*(S_n) is at least

$$\sum_{k=1}^{n/8} \frac{n}{2(2k+1)} \geq \frac{n}{6} \sum_{k=1}^{n/8} \frac{1}{k}$$

$$= \Omega(n \log n).$$

□

Randomized Leader Election

Leader election is impossible in anonymous rings, since no way to break symmetry: there is no *deterministic* algorithm, which works in *every* admissible execution.

Unique ids are a way to break symmetry, which works in every admissible execution.

Another way to break symmetry in *most*, but not all, situations, is to use **randomization**.

A **randomized algorithm** provides each processor with an additional input to its state transition function, a random number.

Weakened definition of the leader election problem:

- At most one leader is elected in every state of every admissible execution. (No change.)
- At least one leader is elected "with high probability". (Weaker than before.)

What does "with high probability" mean?

Randomized Leader Election Algorithm

Assume synchronous system.

Initially:

- Set id to 1 with probability $1 - \frac{1}{n}$ and to 2 with probability $\frac{1}{n}$
- Send id to the left.

When message M is received:

- if M contains n ids then
  - if your id is the unique maximum in M then elected
  - else not elected
- else append your id to M and send to the left

Observations about this algorithm:

- Uses $O(n^2)$ messages.
- There is never more than one leader.
- Sometimes there is no leader.

How often is there no leader?

Random Choices and Probabilities

Since the system is synchronous, to uniquely determine an admissible execution of the algorithm, all that is needed is to fix the random choices obtained initially.

Call this collection of random choices

$$R = \langle r_0, r_1, \ldots, r_{n-1} \rangle,$$

where each $r_i = 1$ or 2. Each R is an element of $\mathcal{R}$.

Denote the execution by *exec*(R).

Definition: For any predicate P on executions, $Pr[P]$ is the probability of $\{R \in \mathcal{R} : exec(R)$ satisfies $P\}$.

In words, $Pr[P]$ is the proportion of random choices resulting in an execution that satisfies P.

Probability of Electing a Leader

For the algorithm given above:

Let P be the predicate "there is a leader".

$Pr[P] =$ probability R contains exactly one 2

$= \binom{n}{1} \cdot \frac{1}{n} \cdot (1 - \frac{1}{n})^{n-1}$

$= (1 - \frac{1}{n})^{n-1}$

$\approx \frac{1}{e}$

$\approx .37$

Thus the probability of electing a leader is more than .37.

Improving the Probability of Electing a Leader

If processors notice there is no leader, they can try again. Each iteration of the basic algorithm is a **phase**.

Keep trying until success.

Random choices that define an execution consist of, for *each processor*, an infinite sequence of 1's and 2's.

In some situations the algorithm will not terminate, for instance, if every random number obtained is 1.

How likely is that to happen?

- Probability of terminating in a particular phase is $(1 - \frac{1}{n})^{n-1}$
- Probability of not terminating in a particular phase is $1 - (1 - \frac{1}{n})^{n-1}$
- Probability of not terminating in k phases is $(1 - (1 - \frac{1}{n})^{n-1})^k$, since phases are independent

The last expression goes to 0 as k increases.

Expected Number of Phases

Definition: The expected value of a random variable T, denoted $E[T]$, is

$$\sum_k k \cdot Pr[T = k]$$

Let T = number of phases until termination.

$Pr[T = k]$

$= Pr[$(first $k - 1$ phases fail)&(k-th phase succeeds)$]$

$= (1 - (1 - \frac{1}{n})^{n-1})^{k-1} \cdot (1 - \frac{1}{n})^{n-1}$

$= (1 - p)^{k-1} \cdot p$, where $p = (1 - \frac{1}{n})^{n-1}$

This is a geometric random variable with expectation $p^{-1} < e$.

Thus the expected number of phases until termination is less than 3.

Shared Memory

Processors communicate via a set of shared variables, instead of passing messages.

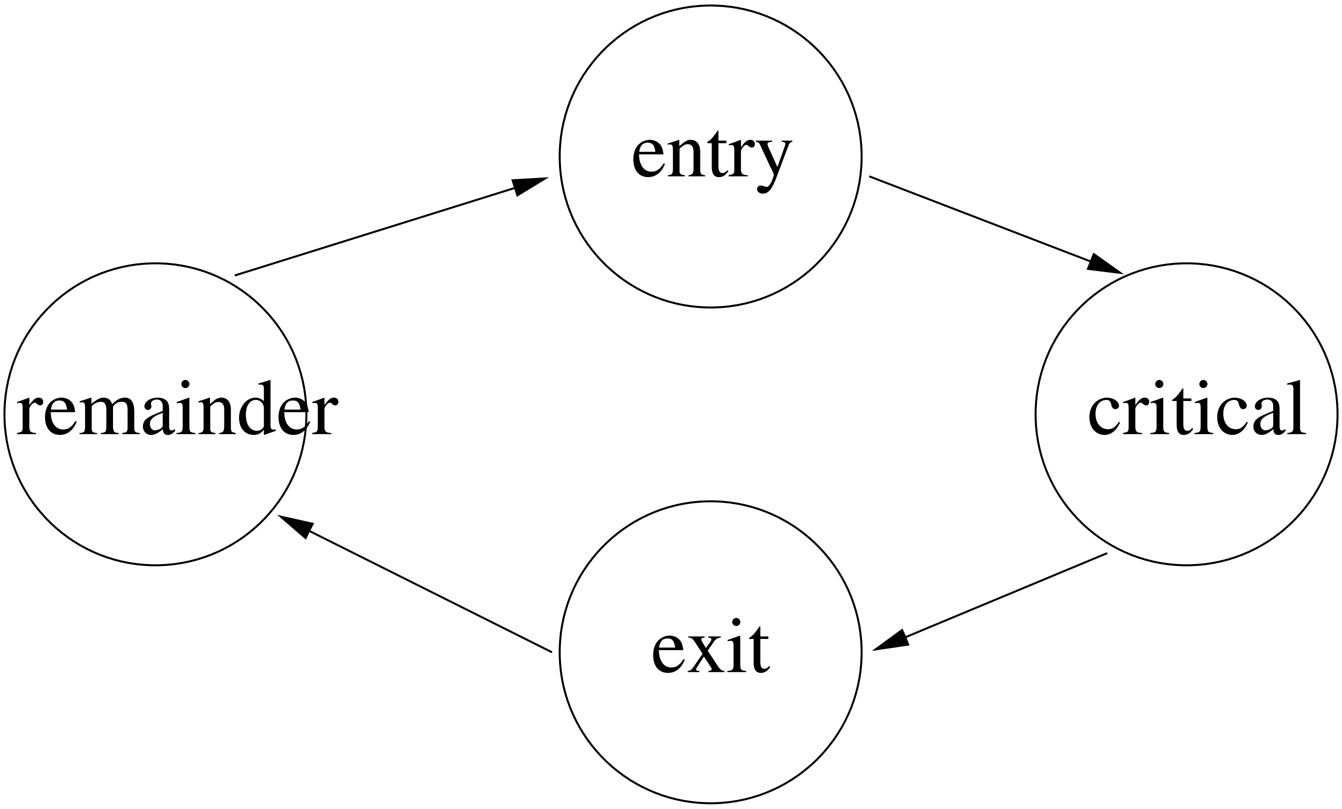
Each shared variable has a **type**, defining a set of operations that can be performed *atomically*.

Changes to the model from the message-passing case:

- no *inbuf* and *outbuf* state components
- configuration includes values of shared variables
- only event type is a computation step by a processor. When p_i takes a step:
  - p_i's state in old config specifies which shared variable is to be accessed and with which operation
  - operation is done; variable's value in the new config changes according to operation's semantics
  - p_i's state in new config changes according to its old state and result of operation
- admissible execution: every processor takes an infinite number of steps

Mutual Exclusion Problem

Each processor's code is divided into four sections:



- *entry section:* synchronize with others to ensure mutually exclusive access to the...
- *critical section:* use some resource; when done, enter the...
- *exit section:* clean up, and then enter the...
- *remainder section:* not interested in the critical section

Mutual Exclusion Algorithms

A *mutual exclusion algorithm* specifies code for entry and exit sections to ensure that:

- **mutual exclusion:** at most one processor is in its critical section at any point, and
- either **no deadlock:** if a processor is in its entry section at some point, then later some processor is in its critical section,
- or **no lockout:** if a processor is in its entry section at some point, then later the *same* processor is in its critical section,
- or **bounded waiting:** no lockout + while a processor is in its entry section, other processors enter the critical section no more than a certain number of times.

Algorithm is allowed to assume:

- no processor stays in its critical section forever
- variables used in the entry and exit sections are not accessed during the critical and remainder sections

Overview of Mutual Exclusion Results

The main complexity measure of interest for shared memory mutual exclusion algorithms is the amount of shared space necessary, which is affected by:

- how powerful the type of the shared variables
- how strong the liveness condition to be satisfied

For most powerful shared variables (read-modify-write), number of different states of the shared memory is:

	upper bound	lower bound
ND	2 (T&S alg.)	2 (obvious)
NL	$\frac{n}{2} + c$ (Burns et al.)	$\frac{n}{2}$ (Burns et al.)
BW	n^2 (queue alg.)	n (Burns & Lynch)

Overview of Mutual Exclusion Results (cont'd)

If the shared variables are the weak read/write kind, we measure the number of *distinct* variables needed.

	upper bound	lower bound
ND		n (Burns & Lynch)
NL	$3n$ boolean (tournament alg.)	
BW	$2n$ unbounded (bakery alg.)	

Mutual Exclusion Algorithm Using Test-and-Set

A **test-and-set** variable V holds 2 values, 0 or 1, and supports the following operations:

- test&set(V):

```
temp := V
V := 1
return temp
```

- reset(V):

```
V := 0
```

<u>ME Algorithm Using One T&S Variable:</u>

```
entry:  wait until test&set(V) = 0
exit:   reset(V)
```

Convince yourself it guarantees mutual exclusion and no deadlock

However one processor could always grab V and starve the others, so it does not satisfy no lockout.

ME Algorithm Using Read-Modify-Write

A more powerful kind of shared variable than test-and-set is **read-modify-write**: its state can be anything of any size and it supports the operation

- RMW(V, f), where f is any function:

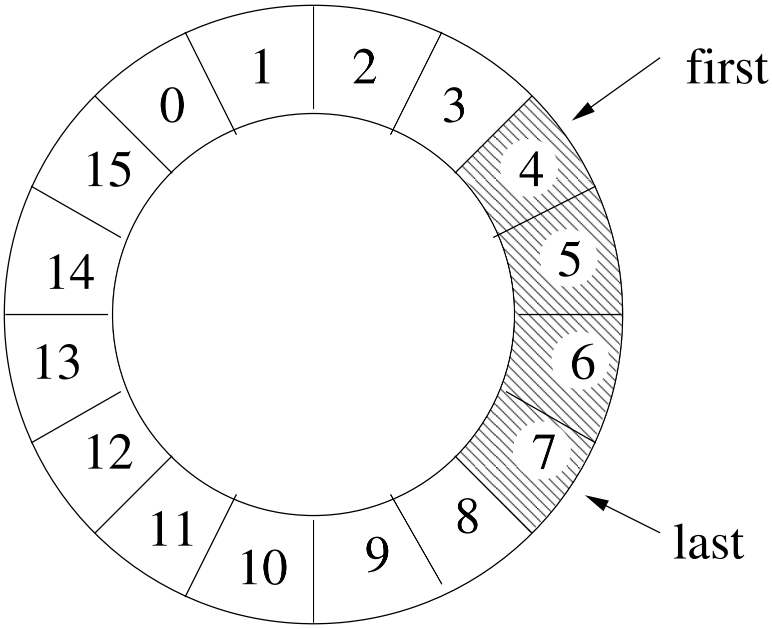
```
temp := V
V := f(V)
return temp
```

This type is so powerful there is no point in having multiple shared variables — instead just put all the shared information together into one.

ME Algorithm Using One RMW Variable:

- Conceptually the list of waiting processors is stored in a circular queue of length n
- Each waiting processor remembers in its local state its location in the queue, instead of keeping this information in the shared variable

ME Algorithm Using Read-Modify-Write (cont'd)



- Shared variable just keeps track of the active part of the queue with *first* and *last* pointers, which are indices into the queue (between 0 and $n-1$).
- Entry section: Increment *last* (mod n) to enqueue self. Wait until *first* equals this value.
- Exit section: Increment *first* to dequeue self.

Correctness: Satisfies ME and n-BW.
Space Complexity: n^2 values, or $O(\log n)$ bits, since all that is stored in the shared variable is *first* and *last* and each can take on n different values independently.

Lower Bound on Number of States of Memory

Theorem (4.3): Any mutual exclusion algorithm with k-bounded waiting must use at least n states of shared memory.

Proof: Assume in contradiction there is an algorithm that uses less than n states of shared memory.

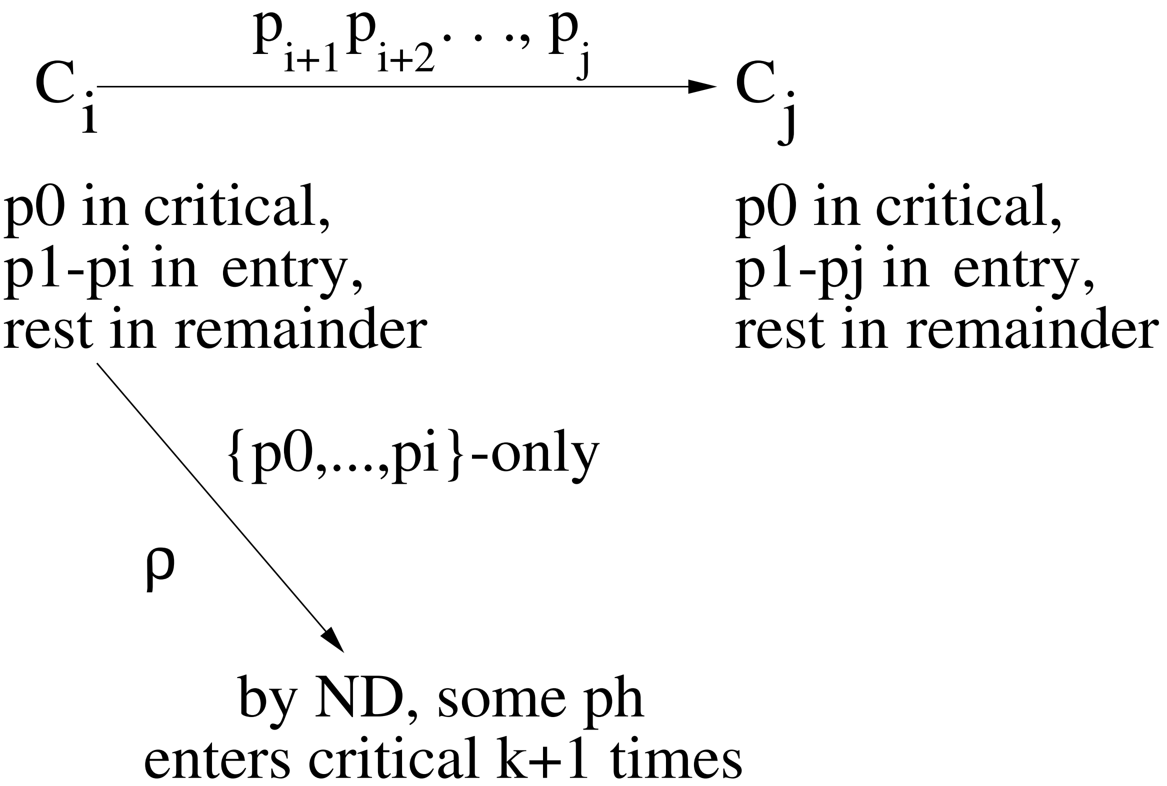
Consider the following execution of the algorithm:

$$C \xrightarrow{p_0\, p_0\, p_0 p_0} C_0 \xrightarrow{p_1} C_1 \xrightarrow{p_2} C_2 \cdots \xrightarrow{p_{n-1}} C_{n-1}$$

C_0	C_1	C_2	C_{n-1}
po in critical by ND	p1 in entry	p2 in entry	pn-1 in entry

There exist $i < j$ such that C_i and C_j have the same state of shared memory.

Lower Bound on No. of Memory States (cont'd)



Processors cannot tell difference between ρ starting at C_i and ρ starting at C_j since

- each processor that takes a step in ρ is in the same state in C_i as in C_j, and
- shared memory state is same in C_i and C_j.

Thus the processors do the same thing in the two executions, including p_h entering its critical section $k+1$ times. But in $exec(C_j, \rho)$, this violates k-bounded waiting w.r.t. p_j who is waiting. $\square$

“Beating” the Lower Bound with Randomization

The lower bound of n just shown for the number of states of shared memory, only holds for algorithms which must guarantee mutual exclusion and bounded waiting in *every* admissible execution.

Recall that even for the weaker liveness condition of no-lockout, the lower bound is $n/2$.

If we are willing to weaken the liveness condition for mutual exclusion to be:

- **Probabilistic No-Lockout:** Every processor has some non-zero probability of succeeding in each attempt to enter the critical section

then there is an algorithm that uses only a *constant* amount of shared memory. (See Section 14.2.)

The algorithm is fairly complicated and has an intricate analysis concerning the adversary.

Mutual Exclusion with Read/Write Variables

Now suppose that the shared variables are of read/write type: processors can atomically read *or* write each variable, but not both.

<u>Bakery Algorithm: NL ME Using R/W Vars</u>

$2n$ shared variables:

- `Choosing[i]`: initially 0; written by p_i, read by others
- `Number[i]`: initially 0; written by p_i, read by others

```
entry:
  Choosing[i] := true
  Number[i] := max(Number[0],...,Number[n-1]) + 1
  Choosing[i] := false
  for j := 0 to n-1 (except i) do
    wait until Choosing[j] = false
    wait until Number[j] = 0 or
               Number[j],j) < (Number[i],i

exit:
  Number[i] := 0
```

Analysis of Bakery Algorithm

Mutual Exclusion: This follows by showing that processor in critical section has the unique smallest `Number` (breaking ties with ids) among all contending processors.

No Lockout: This follows by observing that the contending processor p_i with the smallest `Number` (breaking ties with ids) will be next — every other processor that picks a `Number` while p_i is waiting will get a larger one.

In fact, the algorithm has n-bounded waiting.

Space Complexity: Number of shared variables is $2n$; `Choosing` variables are binary but `Number` variables are unbounded.

Bounded 2-Processor ME Algorithm

Uses 3 binary shared variables:

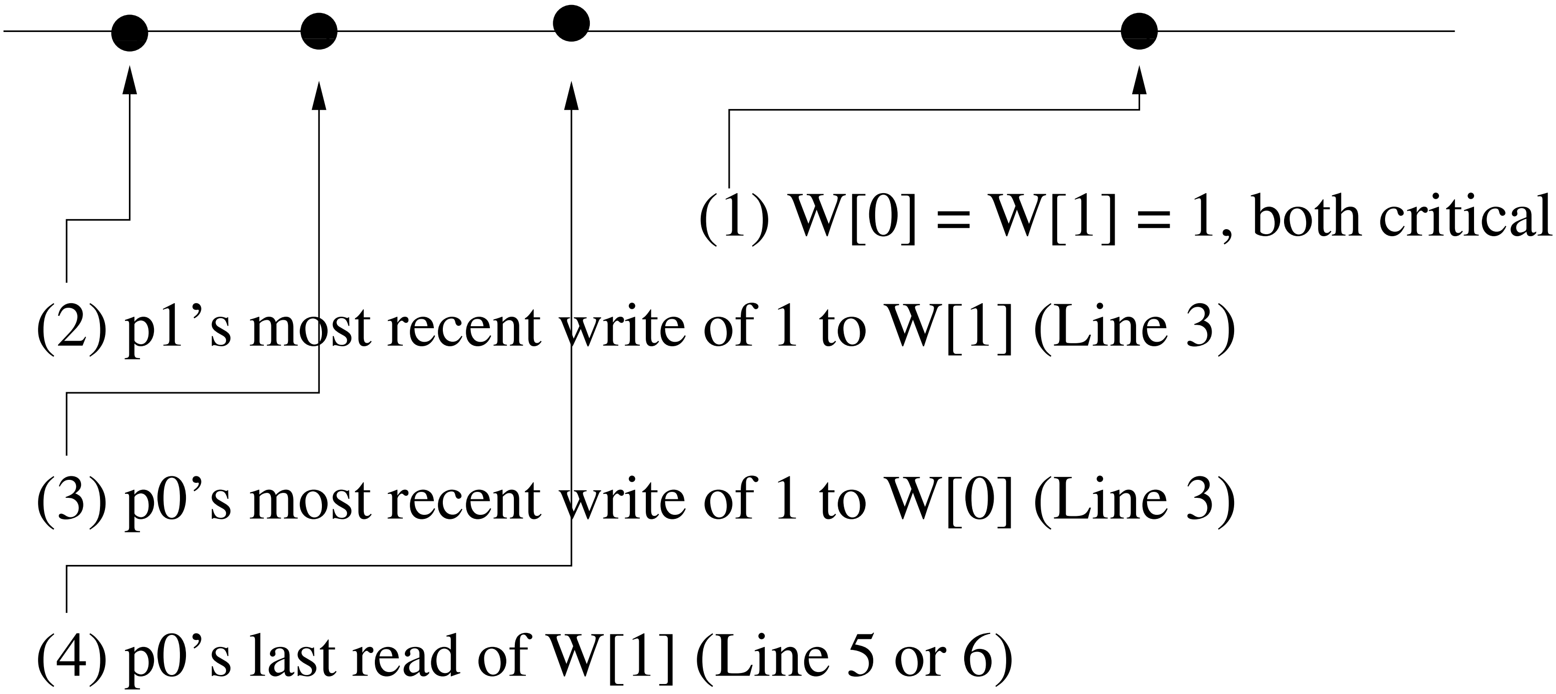
- `W[0]`: written by p_0 and read by p_1, initially 0
- `W[1]`: vice versa, initially 0
- `Priority`: written and read by both, initially 0

```
Entry:
1. W[i] := 0
2. wait until W[1-i] = 0 or
              Priority = i
3. W[i] := 1
4. if (Priority = 1-i) then
5.   if (W[1-i] = 1) then goto Line 1
6. else wait until (W[1-i] = 0)

Exit:
7. Priority := 1-i
8. W[i] := 0
```

Mutual Exclusion for 2-Processor Algorithm

Mutual Exclusion: Suppose in contradiction p_0 and p_1 are simultaneously critical.



WLOG suppose (2) precedes (3). But then in (4), p_0 reads 1, not 0, and thus p_0 cannot be critical at (1).

No Deadlock for 2-Processor Algorithm

(Useful for showing no lockout.)

If one processor (say p_1) ever enters its remainder section for good, then the other processor (say p_0) cannot be starved, since it will see keep seeing `W[1] = 0`.

So any deadlock would starve *both* processors.

WLOG, suppose `Priority` gets stuck at 0 after both processors are stuck in their entry sections.

Thus p_0 is not stuck in Line 2, skips Line 5, and is stuck in Line 6.

Thus p_1 skips Line 6 and is stuck in Line 2 with `W[1]` stuck at 0.

But then p_0 gets unstuck and enters the critical section.

No Lockout for 2-Processor Algorithm

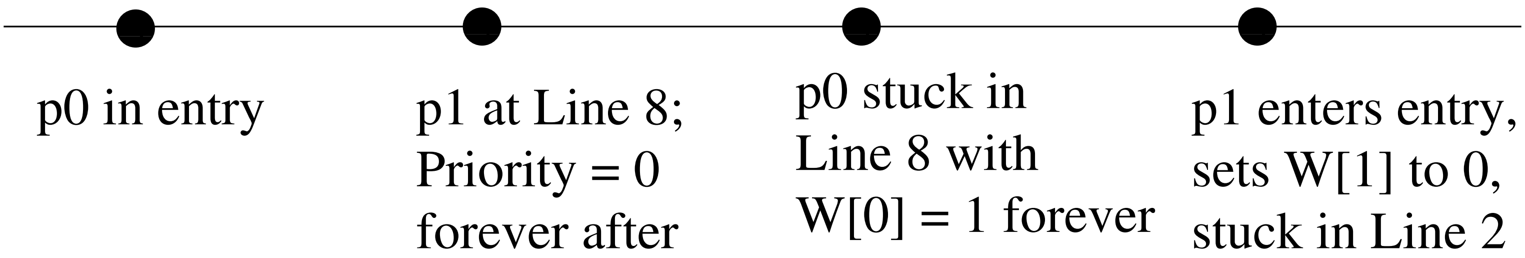
Suppose in contradiction p_0 (WLOG) is starved.

Since there is no deadlock, p_1 subsequently goes critical infinitely often.

The first time that p_1 executes Line 8 after p_0 gets stuck in its entry section, `Priority` gets stuck at 0.

Then p_0 is stuck in Line 6, waiting for `W[1]` to equal 0, with `W[0] = 1`.

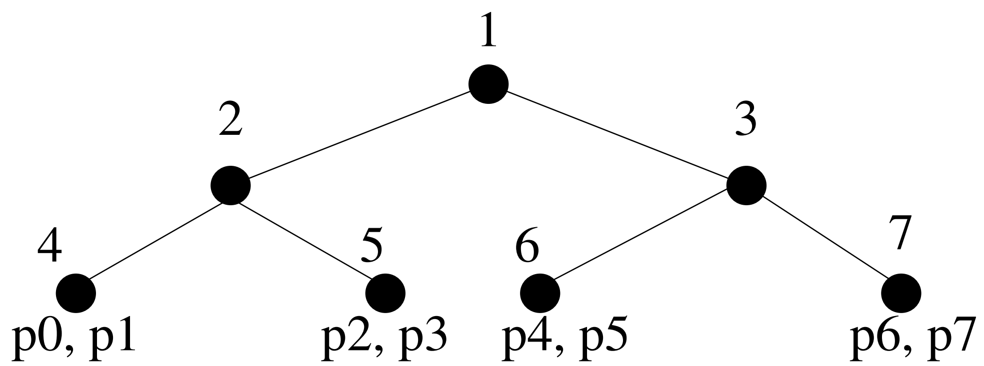
But the next time p_1 enters its entry section, it gets stuck in Line 2 with `W[1] = 0`. This contradicts no deadlock.



Tournament Algorithm

No lockout mutual exclusion algorithm for n processors using bounded size variables:

- based on a tournament tree, complete binary tree with $n-1$ nodes
- A copy of the 2-processor algorithm is associated with each node of the tree
- Each proc. begins at a specified leaf, two per leaf
- A proc. proceeds to next level in tree by winning the 2-processor competition for current node:
  - on left side, play role of p_0
  - on right side, play role of p_1
- when processor wins at root, it enters critical section



Tournament Algorithm (cont'd)

Pseudocode in book is recursive:

- p_i begins at node $2^k + \lfloor \frac{i}{2} \rfloor$, playing the role of $p_{i \bmod 2}$, where $k = \lceil \log n \rceil - 1$.
- After winning at node v, "critical section" for node v is competition for v's parent, node $\lfloor \frac{v}{2} \rfloor$, playing role of $p_{v \bmod 2}$ in 2-proc. algorithm.

Correctness: Based on the correctness of the 2-processor algorithm and the tournament structure.

- Projection of an admissible execution of tournament algorithm onto a particular node produces an admissible execution of 2-proc. algorithm.
- ME for tournament algorithm follows from ME for 2-proc. algorithm at the root node.
- NL for tournament algorithm follows from NL for the 2-proc. algorithms at all nodes of tree.

What about bounded waiting? No.

Space Complexity: $3n$ boolean read/write variables.

Lower Bound on Number of R/W Variables for ME

Theorem (4.18): Any no-deadlock mutual exclusion algorithm using read/write variables must use at least n shared variables.

Proof Strategy: Show by induction on n that there must be at least n variables.

These variables will satisfy a certain property, being **covered** in a configuration: means some processor is about to write it, according to its state in the configuration.

To make the induction go through, the configuration must satisfy the property of **appearing quiescent to a set of processors**: C is P-quiescent if there is a quiescent configuration D such that $C \stackrel{P}{\sim} D$.

Remember that **quiescent** means that all processors are in the remainder section.

Warm-up Lemma

This lemma says that before a processor can enter its critical section, it must write to some uncovered variable.

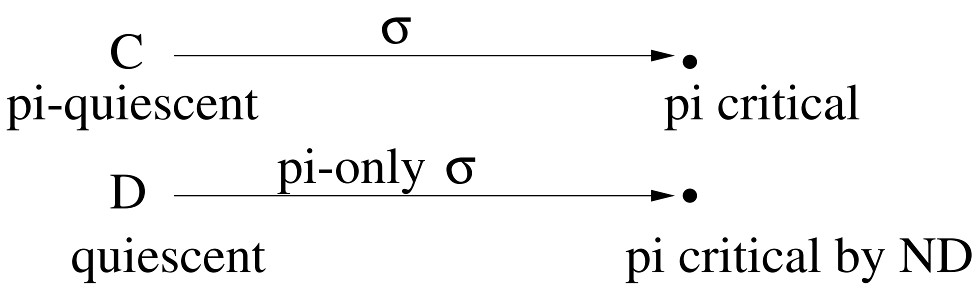
Lemma (4.16): If C is p_i-quiescent, then there is a p_i-only schedule σ such that

(a) p_i is critical in $\sigma(C)$ and

(b) during *exec*(C, σ), p_i writes to some variable that is not covered in C.

Proof: (a)

- Since C is p_i-quiescent, it looks the same to p_i as some quiescent D.
- By no deadlock, some p_i-only schedule σ exists from D that makes p_i critical.
- Applying σ to C also makes p_i critical.

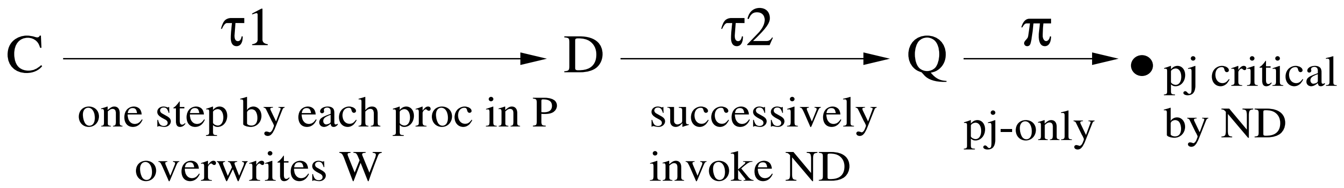


Warm-up Lemma (cont'd)

(b) Suppose during $exec(C, \sigma)$, p_i only writes to variables W, which are covered in C by processors P.

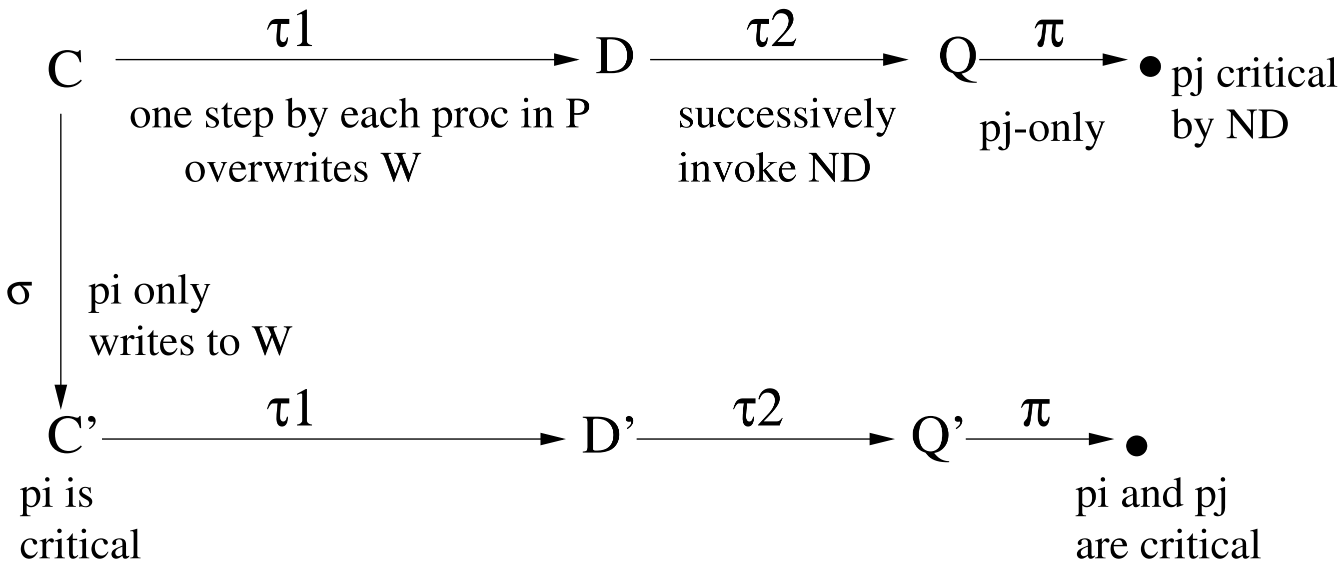
- Define schedule τ_1: starting at C, let each processor in P take one step (overwrites W), producing D
- Define schedule τ_2: starting at $\tau_1(C)$, successively invoke ND to obtain quiescent configuration Q.
- Define schedule π: starting at Q, let some $p_j \in P$ take steps alone until becoming critical (by ND).

Note that only processors in P, which does not include p_i, take steps in $\tau_1 \tau_2 \pi$.



Warm-up Lemma (cont'd)

- Apply σ to C, producing C'; p_i is critical in C'
- Apply τ_1 to $\sigma(C)$, producing D'. Since all steps in τ_1 are writes, and since the only writes in σ have been overwritten, D is P-similar to D'.
- Apply τ_2 to D', producing Q'. Note that Q is P-similar to Q'.
- Apply π to Q', which makes p_j critical. But p_i is still critical from back in C', contradicting mutual exclusion.



□

Main Lemma

Theorem follows from this lemma (part (b) w/ $k = n$):

Lemma (4.17): For all k between 1 and n, for all quiescent C, there exists D such that

(a) D is reachable from C by a $\{p_0, \ldots p_{k-1}\}$-only sched.

(b) $p_0, \ldots, p_{k-1}$ cover k distinct variables in D, and

(c) D is $\{p_k, \ldots, p_{n-1}\}$-quiescent.

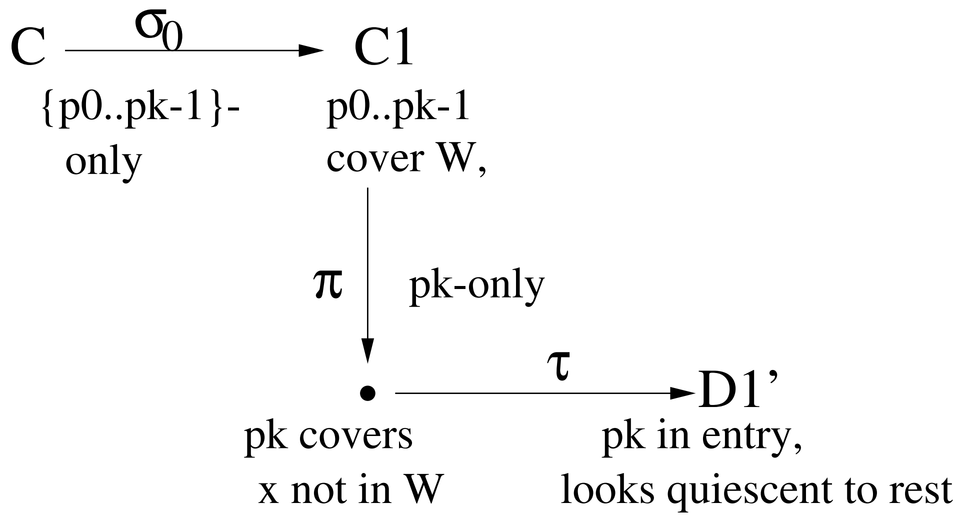
Proof: By induction on k.
Basis: $k = 1$. Since C is quiescent, no variable is covered in C. By warm-up lemma, if p_0 takes steps by itself, it eventually writes to some variable. The desired configuration D is just before p_0's first write:

- D is reachable from C by a p_0-only schedule,
- p_0 covers 1 distinct variable in D,
- D is $\{p_1, \ldots, p_{n-1}\}$-quiescent, since there is no change yet to any shared variable and only p_0 has taken steps.

Main Lemma (cont'd)

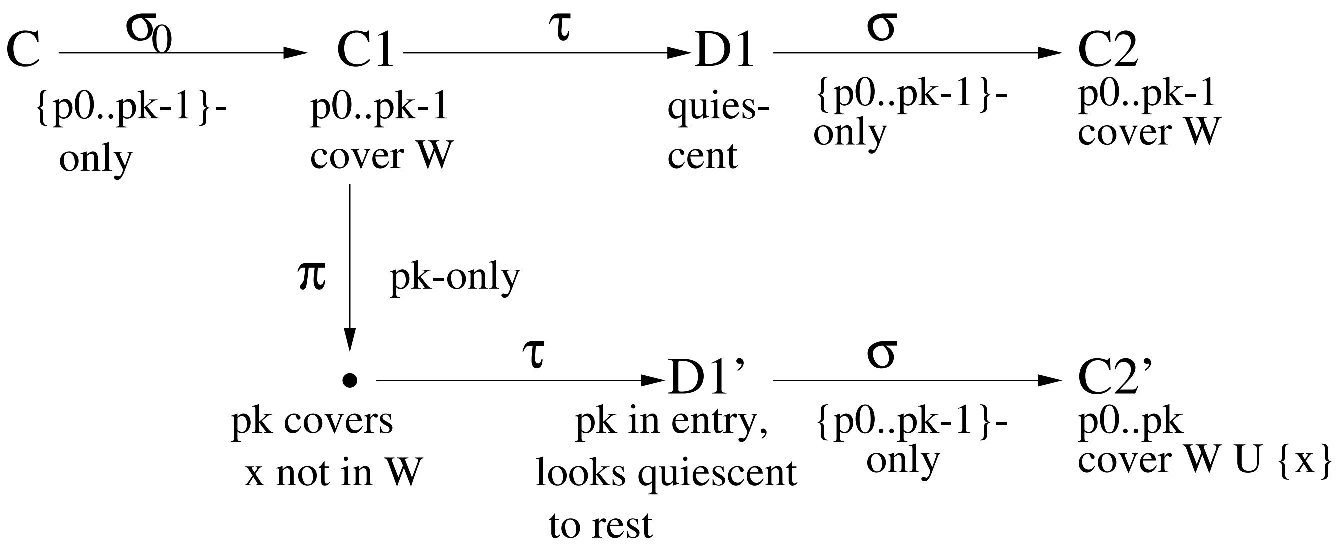
Induction: Assume for $k \geq 1$, show for $k+1$.

- By inductive hypothesis there is $\{p_0, \ldots, p_{k-1}\}$-only sched. σ_0 that takes C to C_1, where C_1 is $\{p_k, \ldots, p_{n-1}\}$-quiescent and p_0 through p_{k-1} cover a set W of k variables.
- By warm-up lemma, there is p_k-only schedule π from C_1 such that p_k covers some $x \notin W$ in $\pi(C_1)$.
- As in warm-up lemma proof, there is schedule τ ($= \tau_1\tau_2$) from $\pi(C_1)$ in which p_0 through p_{k-1} overwrite W and then those processors become quiescent. Let D' be the result, in which p_k is in entry section and rest are in remainder.



Main Lemma (cont'd)

- Applying τ to C_1 results in quiescent config. D_1.
- By induction, there is a $\{p_0, \ldots, p_{k-1}\}$-only sched. σ from D_1 that results in C_2 that is $\{p_k, \ldots, p_{n-1}\}$-quiescent and in which p_0 through p_{k-1} cover a set of k variables. *Assume W again.*
- Applying σ to D_1' produces desired config. C_2':
  - reachable from C by $\{p_0, \ldots, p_k\}$-only sched.
  - $p_0, \ldots, p_k$ cover $k+1$ distinct variables, $W \cup \{x\}$
  - $\{p_{k+1}, \ldots, p_{n-1}\}$-quiescent.



Main Lemma (cont'd)

However, the result of applying σ to D_1 might result in a *different* set of k variables being covered, W' instead of W.

It is possible that W' includes x, in which case only k variables are covered in C_2'.

To fix this problem, we repeatedly apply the inductive hypothesis to get

$C_1, D_1, C_2, D_2, C_3, D_3, \ldots$.

Since there are a finite number of shared variables, there exist distinct i and j such that in configurations C_i and C_j, the same set of k variables is covered.

Then apply essentially the same argument as above, replacing C_1 and C_2 with C_i and C_j.

$\square$

Fast Mutual Exclusion

The previous mutual exclusion algorithms require a processor to access $\Omega(n)$ variables even in the absence of contention.

It would be nice to have a **fast** algorithm: if there is no competition, then a processor enters the critical section in $O(1)$ steps.

Even nicer would be an **adaptive** algorithm: the number of steps (and/or other complexity measures) depends on the number of competing processors, *not* the total number of processors.

Fast Mutual Exclusion Algorithm

```
Initially FL = -1, // fast lock
          SL = -1, // slow lock
          W[i] = F for all i // want flag

Code for pi:

Entry:
 1. W[i] := T
 2. FL := i
 3. if SL <> -1 then
 4.    W[i] := F
 5.    wait until SL = -1
 6.    go to 1
 7. SL := i
 8. if FL <> i then // else enter via fast path
 9.    W[i] := F
10.    for all j, wait until W[j] = F
11.    if SL <> i then  // else enter via slow path
12.       wait until SL = -1
13.       go to 1

Exit:
14. SL := -1
15. W[i] := F
```

Analysis of Fast ME Algorithm

Fast: If no contention, then p_i sets FL to i, skips lines 4-6, sets SL to i, skips lines 9-13, and enters the critical section in $O(1)$ steps.

Mutual Exclusion: Suppose in contradiction p_i and p_j are simultaneously critical. One case is when p_i and p_j both enter via the fast path:

- p_i sets W[i] to true in Line 1
- p_i sets FL to i in Line 2
- p_i sets SL to i in Line 7
- p_i sees FL = i in Line 8
- p_i enters critical

Then events of p_j are interleaved as follows:

- p_j writes j to FL in Line 2 after p_i writes i to FL in Line 2 AND after p_i sees FL = i in Line 8
- But then p_j sees i in SL in Line 3 and waits in Line 5 instead of continuing.

Fault-Tolerant Consensus

Types of **processor failure:**

- *crash:* in middle of step, might only send a subset of messages
- *Byzantine:* take arbitrary actions

Consensus problem: Every processor has an input.

- *Termination:* Eventually every nonfaulty processor must decide on a value.
- *Agreement:* All nonfaulty decisions must be the same.
- *Validity:* If all inputs are the same, then the nonfaulty decision must be that input.

Validity ensures that outputs bear some relationship to inputs (but also rules out easy solutions!).

Background: Collection of armies, all on the same side. Each general begins with an opinion whether to attack. If all attack, they will win, otherwise they will lose. Some generals are *traitors* and will behave incorrectly.

Overview of Consensus Results

Let f be the maximum number of faulty processors.

Tight bounds for synchronous message passing:

	crash failures	Byzantine failures
number of rounds	$f+1$	$f+1$
total number of procs	$\geq f+1$	$\geq 3f+1$
message size	polynomial	polynomial

Asynchronous case: *impossible* in both shared memory and message passing, even if only one crash failure is to be tolerated.

Modeling Processor Failures

For an execution to be **admissible:**

Crash Failures:
All but a set of at most f processors (the **faulty** ones) take an infinite number of steps.

- In synchronous case: once a faulty processor fails to take a step in a round, it takes no more steps.
- In message passing case: In a faulty processor's last step, an arbitrary subset of the processor's outgoing messages make it into the channels. (This is where the difficulties lie.)

Byzantine Failures:
A set of at most f processors (the **faulty** ones) can send messages with arbitrary content and change state arbitrarily (not according to their transition functions).

Consensus Algorithm for Crash Failures

Code for p_i:

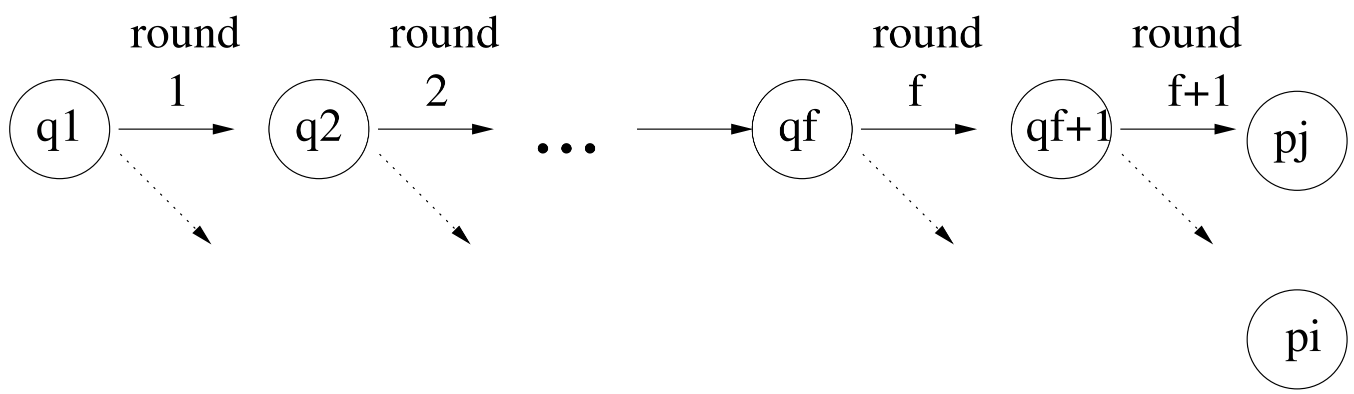
```
v := my input
at each round 1 through f+1:
   if I have not yet sent v then
      send v to all
   v := minimum among all received
      values and current value of v
   in round f+1, decide on v
```

Termination: By the code.
□

Validity: Holds since processors do not introduce spurious messages: if all inputs are the same, then that is the only value ever in circulation.
□

Analysis of Crash Consensus Algorithm

Agreement: Suppose p_j decides on a smaller value, x, than does p_i. Then x was hidden from p_i by a chain of faulty processors:



There are $f + 1$ faulty processors in this chain, a contradiction.

□

Performance:

- number of processors $n > f$
- $f + 1$ rounds
- $n^2 \cdot |V|$ messages each of size $\log |V|$, where V is the input set.

□

Lower Bound on Number of Rounds

Notice from the chain argument above that the algorithm cannot terminate before round $f+1$ in the worst case, since disagreement could result.

This kind of bad behavior can afflict *any* algorithm:

Theorem (5.7): Any consensus algorithm in synchronous MP that tolerates f crash failures takes at least $f+1$ rounds in the worst case, when $n \geq f+2$.

To prove this, we need some definitions:

Executions α_1 and α_2 are **similar to** p_i, denoted $\alpha_1 \stackrel{p_i}{\sim} \alpha_2$, if p_i is nonfaulty in both and has the same view in both.

Executions α_1 and α_2 are **indirectly similar**, denoted $\alpha_1 \approx \alpha_2$, if there is a chain of executions and processors such that

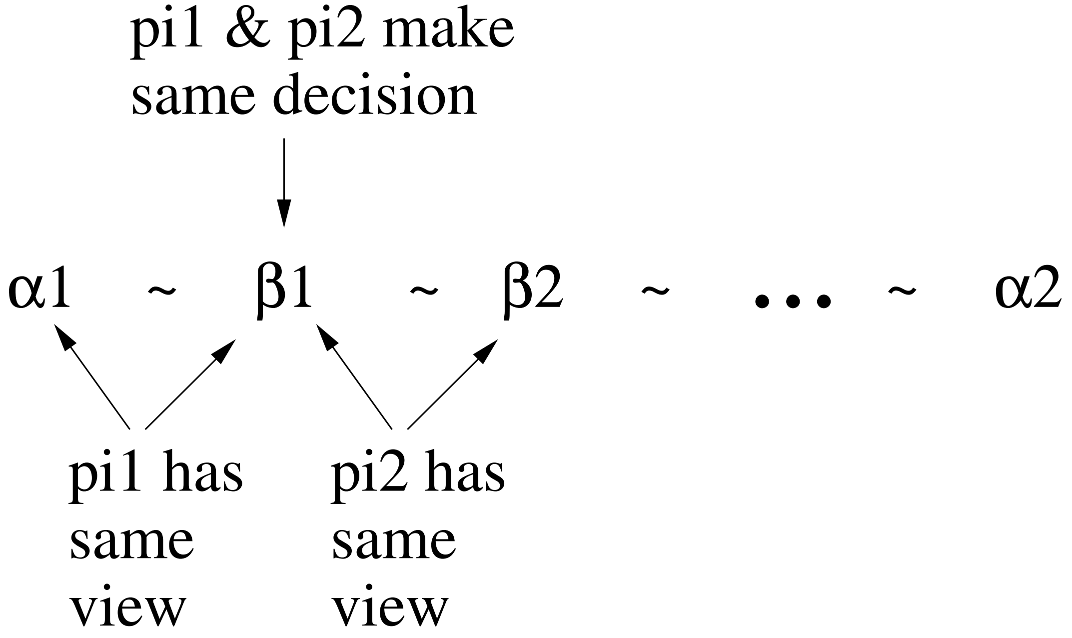
$$\alpha_1 \stackrel{p_{i_1}}{\sim} \beta_1 \stackrel{p_{i_2}}{\sim} \beta_2 \ldots \stackrel{p_{i_k}}{\sim} \alpha_2.$$

Round Lower Bound (cont'd)

Claim: If $\alpha_1 \approx \alpha_2$, then decision values are same.

Why?

- p_{i_1} has same view in α_1 and β_1.
- p_{i_1} decides same in α_1 and β_1.
- p_{i_2} decides same as p_{i_1} in β_1.
- p_{i_2} has same view in β_1 and β_2.
- p_{i_2} decides same in β_1 and β_2.
- Etc.



$\square$

Round Lower Bound (cont'd)

Start with the simple case of $f = 1$:

Theorem (5.5): Any consensus algorithm in synchronous MP that tolerates 1 crash failure takes at least 2 rounds in the worst case, when $n \geq 3$.

Proof: Assume in contradiction there is an algorithm that decides in one round. Show

$$\alpha^0 \approx \alpha^1 \approx \alpha^2 \ldots \approx \alpha^n$$

where α^i is the failure-free execution in which the first i processors have input 1 and the rest have input 0.

But α^0 must decide 0 while α^1 must decide 1, contradicting the claim.

How to show $\alpha^i \approx \alpha^{i+1}$?

Round Lower Bound (cont'd)

To show $\alpha^i \approx \alpha^{i+1}$:

1. Create a chain of executions, starting at α^i, in which the messages that p_i sends in the first round are removed, one by one.
   - Since there is only one round, there is no opportunity for processors to exchange information about whether or not they heard from p_i in round 1.
   - Each adjacent pair in this chain look similar to some nonfaulty processor.
2. At the end of this chain is an execution in which p_i sends no messages.
3. Change p_i's input to 1 instead of 0. Since p_i sends no messages, no processor can tell the difference.
4. Create a chain of executions in which the messages that p_i sends in the first round are added back, one by one. Analogous to step 1.
5. The execution at the end of this chain is α^{i+1}.

□

Round Lower Bound (cont'd)

Proving the general case, $f > 1$, uses the same approach as for the $f = 1$ case. However, proving that $\alpha^i \approx \alpha^{i+1}$ is more difficult, since there are more rounds. Strategy is to show:

$$\alpha^i \approx \text{crash}(\alpha^i, p_i, 1) \approx \text{crash}(\alpha^{i+1}, p_i, 1) \approx \alpha^{i+1}.$$

where crash(α, p_i, r) denotes the execution that has same inputs and same failure pattern as α except that p_i crashes at the beginning of round r.

The middle $\approx$ is easy to see, since the only difference in the two executions is the input to p_i, who never sends a message.

The first and third $\approx$ follow from the next lemma...

Round Lower Bound (cont'd)

Lemma (5.6): If α has no failures in round r or later and p_i is nonfaulty, then $\alpha \approx \text{crash}(\alpha, p_i, r)$.

Proof: By backward induction on r.

Basis: $r = f$. Show that α is indirectly similar to an execution that is the same as α except that p_i fails in the last round. Same as the $f = 1$ case.

Induction: Assume true for $r + 1$ and show for r. To use the inductive hypothesis, break the problem down to show:

$$\alpha \approx \text{crash}(\alpha, p_i, r+1) \approx \text{crash}(\alpha, p_i, r).$$

The first $\approx$ is true by the inductive hypothesis.

Round Lower Bound (cont'd)

To show crash$(\alpha, p_i, r+1) \approx$ crash(α, p_i, r):

- In crash$(\alpha, p_i, r+1)$, consider p_i as crashing at *end* of round r, instead of *beginning* of $r+1$.
- Create a chain of executions, starting with crash$(\alpha, p_i, r+1)$, in which the messages that p_i sends in round r are removed, one by one.
- Let α_j denote the execution in this chain in which p_i fails to send to the j highest numbered processors.
- Show that $\alpha_{j-1} \approx \alpha_j$.

Round Lower Bound (cont'd)

To show that $\alpha_{j-1} \approx \alpha_j$:

The first difference between α_{j-1} and α_j is that some processor p_h receives p_i's message in round r of α_{j-1} but not in round r of α_j.

Case 1: p_h is faulty in α. Since α has no failures in round r or later, p_h is already dead by round r. Thus no nonfaulty processor can tell whether p_h gets p_i's message or not, and so $\alpha_{j-1} \approx \alpha_j$.

Case 2: p_h is nonfaulty in α. Idea is to crash p_h in round $r+1$ so it cannot tell anyone whether or not it heard from p_i:
$\alpha_{j-1} \approx \text{crash}(\alpha_{j-1}, p_h, r+1) \approx \text{crash}(\alpha_j, p_h, r+1) \approx \alpha_j$

The first and third $\approx$ are by the inductive hypothesis. The second $\approx$ is because the only difference is whether p_h gets p_i's message in round r, but p_h never gets an opportunity to tell.
$\square$

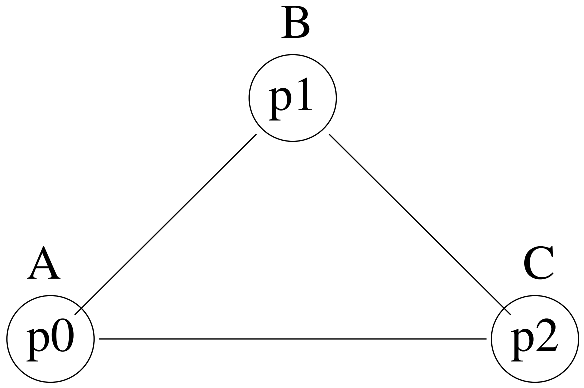
Byzantine Failures

How many processors total are needed to solve consensus when $f = 1$?

- Suppose $n = 2$. If p_0 starts with input 0 and p_1 starts with input 1, then someone has to change, but not both. What if one processor is faulty? How can the other one know?
- Suppose $n = 3$. If p_0 has input 0, p_1 has input 1, and p_2 is faulty, then a tie-breaker is needed, but p_2 might be malicious.

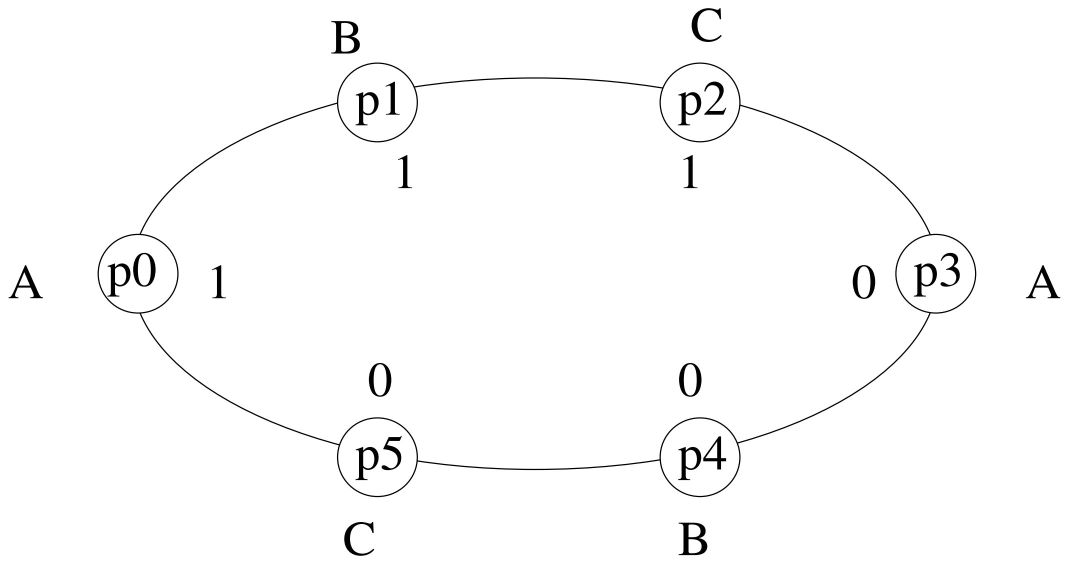
Theorem (5.8): Any consensus algorithm for message passing that tolerates 1 Byzantine failure must have at least 4 processors total.

Proof: Suppose there is a consensus algorithm $\mathcal{A} = (A, B, C)$ for 3 processors and 1 Byzantine failure.



Processor Lower Bound for Byzantine Case

Now consider a ring of six processors running components of $\mathcal{A}$ in this fashion:

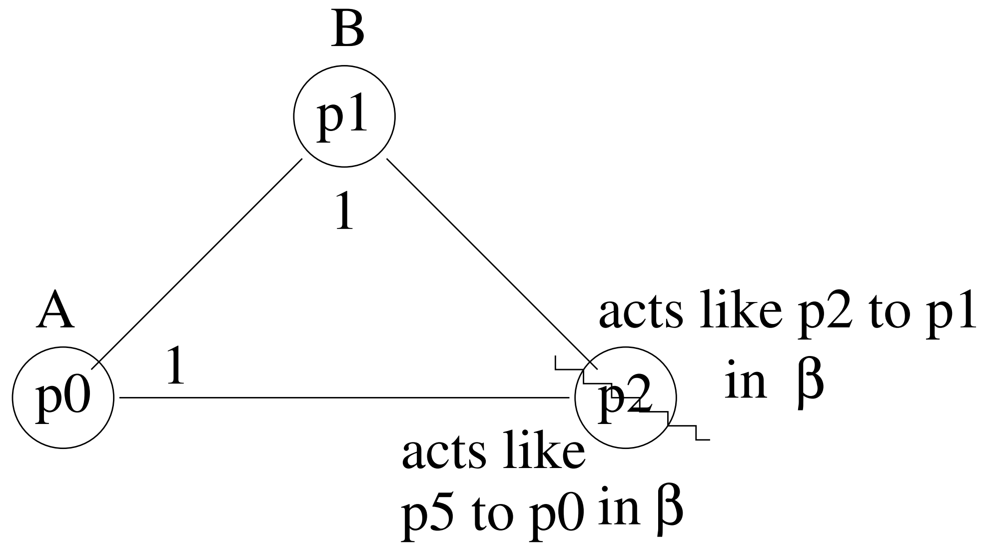


Give each processor the indicated input and let the ring execute. Call the resulting execution β.

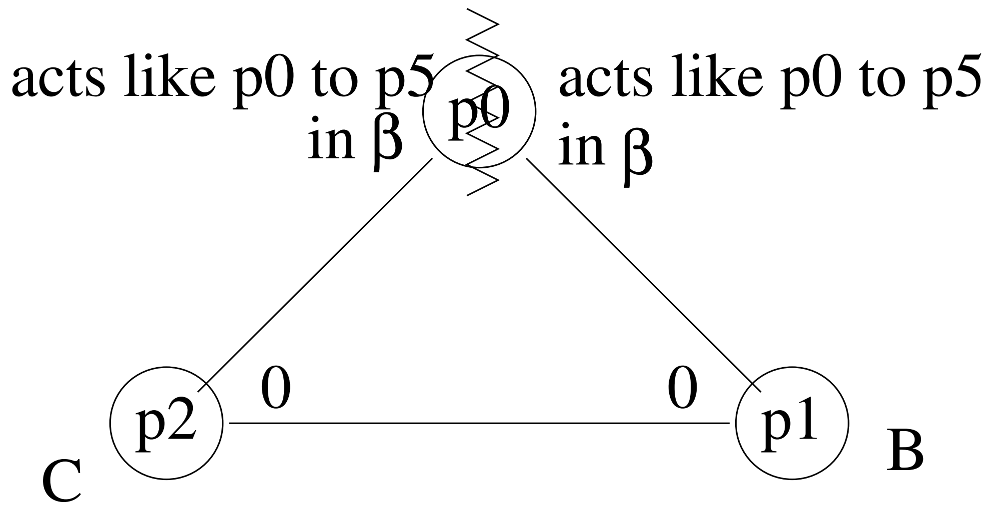
- β does not necessarily solve consensus: it doesn't have to, since the assumptions under which $\mathcal{A}$ is supposed to work do not hold.
- However, the processors do something. This behavior will be used to specify the behavior of the faulty processors in certain particularly adversarial executions of $\mathcal{A}$ on the triangle.

Processor Lower Bound (cont'd)

Let α_1 be this execution, in which 1 is decided:

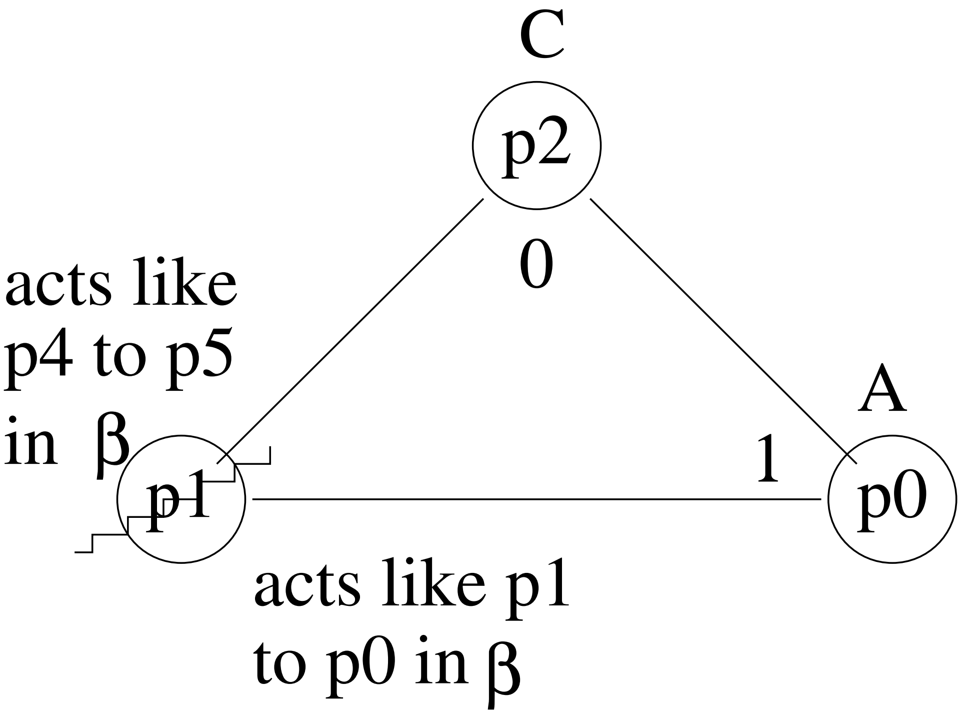


Let α_2 be this execution, in which 0 is decided:



Processor Lower Bound (cont'd)

Let α_3 be this execution:



What is decided in α_3?

- p_0's view in α_3 equals p_0's view in β, which equals p_0's view in α_1. Thus p_0 decides 1 in α_3.
- p_2's view in α_3 equals p_2's view in β, which equals p_2's view in α_2. Thus p_2 decides 0 in α_3.
- But this contradicts agreement.

□

Read reduction in textbook to show $n = 3f$ is impossible for $f > 1$.

Consensus Algorithms for Byzantine Failures

Minimum number of rounds is $f + 1$, since crash failures are a special case of Byzantine failures.

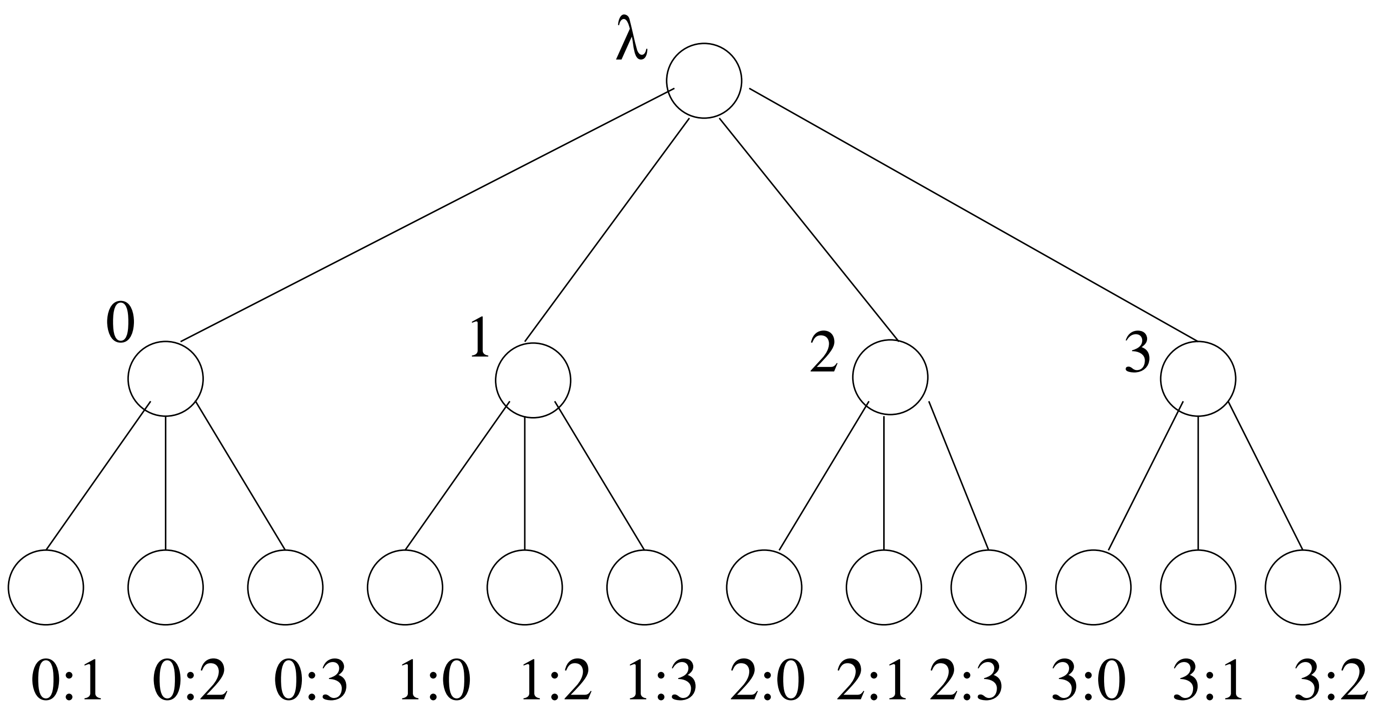
Exponential Tree Algorithm

Each processor maintains a tree data structure in its local state. Each node of the tree is labeled with a sequence of processor indices with no repeats:

- root's label is empty sequence λ (root has level 0)
- root has n children labeled 0 through $n - 1$
- child node labeled i has $n - 1$ children labeled $i : 0$ through $i : n - 1$, skipping $i : i$
- in general, node at level d with label v has $n - d$ children labeled $v : 0$ through $v : n - 1$, skipping any index appearing in v
- nodes at level $f + 1$ are the leaves.

Example of Exponential Tree

The tree when $n = 4$ and $f = 1$:



Exponential Tree Algorithm

Each processor fills in the tree nodes with values as the rounds go by:

- initially, store your input in the root (level 0)
- round 1: send level 0 of your tree (the root);
  store value received from p_j in node j (level 1)
  (default if none)
- round 2: send level 1 of your tree;
  store value received from p_j for node k in node $k : j$
  (level 2) ("the value that p_j told me that p_k told p_j")
  (default if none)
- continue for $f + 1$ rounds

In the last round, each processor uses the values in its tree to compute its decision. The decision is resolve(λ), where resolve(π) equals

- value in tree node labeled π if it is a leaf
- majority$\{$resolve$(\pi') : \pi'$ is a child of $\pi\}$ otherwise (default if none).

Proof of Exponential Tree Algorithm

Lemma (5.10): Nonfaulty processor p_i's resolved value for node $\pi = \pi' j$, what p_j reports for π', equals what p_j has stored for π'.

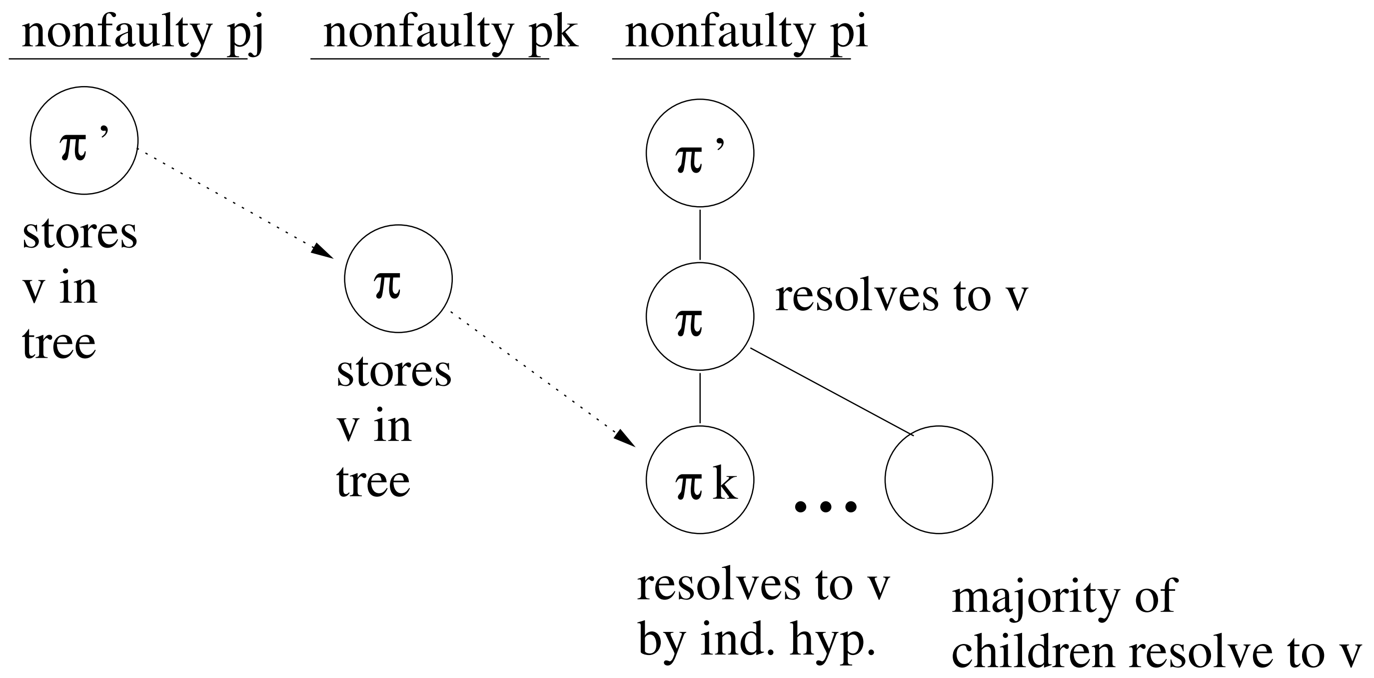
Proof: By induction on the height of π.

Basis: π is a leaf. Then p_i stores in node π what p_j sends it for π' in the last round.. For leaves, the resolved value is the tree value.

Induction: π is not a leaf.

- By tree definition, π has at least $n - f$ children.
- Since $n > 3f$, π has majority of nonfaulty children.
- Let πk be a child of π such that p_k is nonfaulty.
- Since p_j is nonfaulty, p_j correctly reports to p_k that it has some value v in node π'; thus p_k stores v in node $\pi = \pi' j$.
- By induction, p_i's resolved value for πk equals the value v that p_k has in its tree node π.
- So all of π's nonfaulty children resolve to v in p_i's tree, and thus π resolves to v in p_i's tree.

Proof of Exponential Tree Algorithm



□

Validity: Suppose all inputs are v.

- Nonfaulty processor p_i decides on resolve(λ), which is the majority among resolve(j), $0 \leq j \leq n-1$.
- The previous lemma implies that for each nonfaulty p_j, resolve(j) is the value stored at the root of p_j's tree, which is p_j's input v.
- Thus p_i decides v.

□

Proof of Exponential Tree Algorithm (cont'd)

Agreement: Show that all nonfaulty processors resolve the same value for their tree roots.

A node is **common** if all nonfaulty processors resolve the same value for it. We will show the root is common. Strategy:

1. Show that every node with a certain property is common.
2. Show that the root has the property.

Lemma (5.11): If every π-to-leaf path has a common node, then π is common.
Proof: By induction on the height of π.
Basis: π is a leaf. Then every π-to-leaf path consists solely of π, and since the path is assumed to contain a common node, that node is π.

Proof of Exponential Tree Algorithm (cont'd)

Induction: π is not a leaf. Suppose in contradiction π is not common.

- Then every child π' of π has the property that every pi'-to-leaf path has a common node.
- Since the height of π' is smaller than the height of π, the inductive hypothesis implies that π' is common.
- Therefore all nonfaulty processors compute the same resolved value for π, and thus π is common.

□

Show every root-to-leaf path has a common node.

- There are $f + 2$ nodes on a root-to-leaf path.
- The label of each non-root node on a root-to-leaf path ends in a distinct processor index: $i_1, i_2, \ldots, i_{f+1}$.
- At least one of these indices is that of a nonfaulty processor, say i_k.
- Lemma 5.10 implies that the node whose label ends in i_k is common.

□

Proof of Exponential Tree Algorithm (cont'd)

Complexity:

- $n > 3f$ processors
- $f + 1$ rounds
- Messages in round r contain $n(n-1)(n-2)\cdots(n-(r-2))$ values.

  When $r = f + 1$, this is exponential if f is more than constant relative to n.

A Polynomial Algorithm for Byzantine Agreement

We can reduce the message size with a simple algorithm that increases the number of processors to $n > 4f$ and number of rounds to $2(f+1)$.

Phase King Algorithm

Uses $f+1$ phases, each taking two rounds.
Code for p_i:

```
pref := my input

first round of phase k:
  send pref to all
  receive prefs of others
  let maj be the value that occurs > n/2 times
      among all prefs (0 if none)
  let mult be number of times maj occurs

second round of phase k:
  if i = k then send maj // I am the phase king
  receive tie-breaker from pk (0 if none)
  if mult > n/2 + f
     then pref := maj
     else pref := tie-breaker
  if k = f+1 then decide pref
```

Proof of Phase King Algorithm

Lemma (5.13): If all nonfaulty processors prefer v at start of phase k, then all do at end of phase k.
Proof:

- Each nonfaulty processor receives at least $n - f$ preferences (including its own) for v in the first round of phase k.
- Since $n > 4f$, $n - f > n/2 + f$.
- Thus the processor still prefers v.

□

Validity: Follows from Lemma 5.13.

□

Proof of Phase King Algorithm (cont'd)

Lemma (5.14): If the king of phase k is nonfaulty, then all nonfaulty processors have the same preference at the end of phase k.

Proof: Consider two nonfaulty processors p_i and p_j.

Case 1: p_i and p_j both use p_k's tie-breaker. Since p_k is nonfaulty, they agree.

Case 2: p_i uses its majority value and p_j uses the king's tie-breaker.

- p_i's majority value is v.
- p_i receives more than $n/2 + f$ preferences for v.
- p_k receives more than $n/2$ preferences for v.
- p_k's tie-breaker is v.

Case 3: p_i and p_j both use their own majority values.

- p_i's majority value is v.
- p_i receives more than $n/2 + f$ preferences for v.
- p_j receives more than $n/2$ preferences for v.
- p_j's majority value is also v.

$\square$

Proof of Phase King Algorithm (cont'd)

Agreement:

- Since there are $f + 1$ phases, at least one has a nonfaulty king.
- Lemma 5.14 implies that at the end of that phase, all nonfaulty processors have the same preference.
- Lemma 5.13 implies that from that phase onward, the nonfaulty preferences stay the same.
- Thus the decisions are the same.

$\square$

Performance:

- number of processors $n > 2f$
- $2(f + 1)$ rounds
- $O(n^2 f)$ messages, each of size $\log |V|$.

Impossibility of Asynchronous Consensus

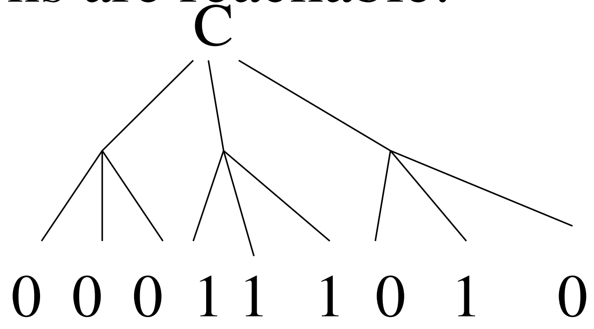
1. Show impossible in shared memory with $n-1$ faults. Prove directly: not hard since lots of faults.
2. Step 1 implies there is no 2-proc., 1-resilient alg.
3. Show impossible in shared memory with 1 fault. Prove by reduction:
   - Show how to use a hypothetical n-proc., 1-resilient alg. to design a 2-proc., 1-resilient alg.
   - Since step 2 implies there is no 2-proc., 1-resilient alg., the hypothetical n-proc., 1-resilient alg. cannot exist.
4. Show impossible in message passing with 1 fault. Prove by reduction:
   - Show how to use a hypothetical message passing alg. to design a shared memory alg.
   - Since step 3 implies there is no shared memory alg., the hypothetical message passing alg. cannot exist.

Impossibility of Wait-Free Consensus

An algorithm for n processors is **wait-free** if it can tolerate $n-1$ failures: nonfaulty processor does not *wait* for others.

Theorem (5.19): There is no wait-free consensus algorithm in an asynchronous shared memory system using read/write registers.

Proof: Assume in contradiction there is one. Classify configurations of the algorithm according to how many different decisions are reachable:



bivalent: both 0 and 1 are reachable decisions

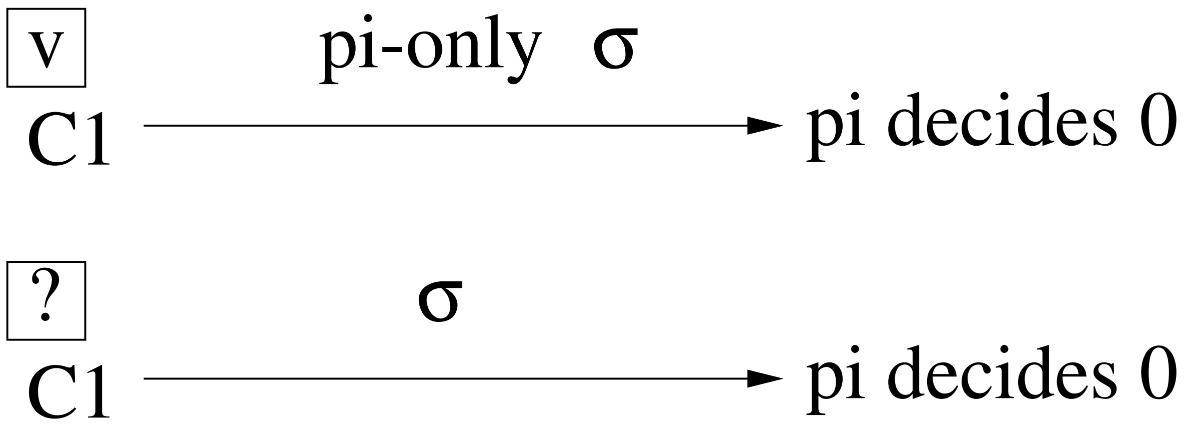
univalent: only one decision is reachable; **0-valent** or **1-valent**.

Strategy: Construct an admissible execution in which all configurations are bivalent — processors never decide.

Impossibility of Wait-Free Consensus (cont'd)

Claim (5.16): If C_1 and C_2 are univalent and $C_1 \overset{p_i}{\sim} C_2$, then C_1 and C_2 have the same valency.
Proof: Suppose C_1 is v-valent.

- *Since the algorithm is wait-free*, there is a schedule σ in which only p_i takes steps that causes p_i to decide v.
- Since p_i cannot tell the difference between C_1 and C_2, if σ is applied to C_2, p_i also decides v.
- Thus C_2 is also v-valent.



□

Impossibility of Wait-Free Consensus (cont'd)

Lemma (5.17): There is a bivalent initial configuration.

Proof:

- I_0 is initial configuration with all inputs 0. It is 0-valent.
- I_{01} is initial configuration in which p_0's input is 0 and the rest are 1.
- I_0 and I_{01} look the same to p_0, so Claim 5.16 implies I_{01} cannot be 1-valent.
- I_1 is initial configuration with all inputs 1. It is 1-valent.
- I_{01} and I_1 look the same to p_1, so Claim 5.16 implies I_{01} cannot be 0-valent.
- Thus I_{01} is bivalent.

$$I_0 \overset{p0}{\sim} I_{01} \overset{p1}{\sim} I_1$$

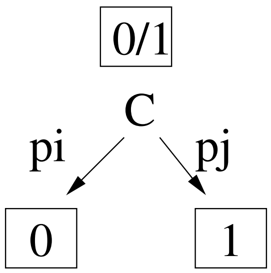
□

Impossibility of Wait-Free Consensus (cont'd)

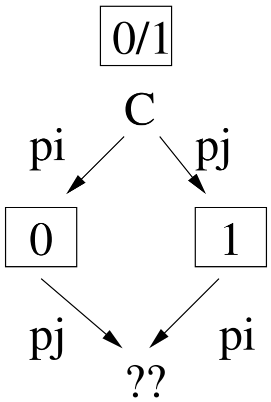
p_i is **critical** in bivalent C if p_i's step in C results in a univalent configuration.

Lemma (5.18): If C is bivalent, then at least one processor is *not* critical in C. I.e., there is a processor that can take a step and keep the system bivalent.

Proof: Suppose in contradiction all processors are critical in C. Then there exist p_i and p_j such that



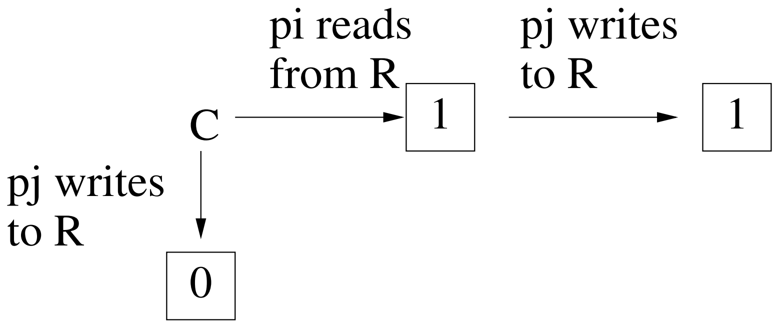
Case 1: p_i and p_j access different registers or read the same register.



The operations commute. Contradiction.

Impossibility of Wait-Free Consensus (cont'd)

Case 2: P_i writes to and p_j reads from the same register R.



But $i(C) \overset{p_i}{\sim} i(j(C))$, contradicting Claim 5.16.

□

Now construct an admissible execution in which all configurations are bivalent:

1. Start with the bivalent initial configuration from Lemma 5.17.
2. Use Lemma 5.18 to get next bivalent configuration by having a non-critical processor take a step.
3. Go to step 2.

□

Impossibility of 1-Resilient Consensus

Even if the ratio of nonfaulty processors becomes overwhelming, consensus still cannot be solved in asynchronous SM (with read/write registers).

1. Assume there exists an algorithm A for n processors and 1 failure.
2. Use A as a subroutine to design algorithm A' for 2 processors and 1 failure.
3. Previous result shows A' cannot exist.
4. Thus A cannot exist.

WLOG, assume that algorithm A for $q_0, q_1, \ldots, q_{n-1}$, satisfies:

- each q_j has a single register R_j which is writes and others read, initially empty
- code of each q_j alternates reads and writes, beginning with a read
- each write step of each q_j writes q_j's entire current state into R_j

Impossibility of 1-Resilient Consensus (cont'd)

Idea of algorithm A' for p_0 and p_1:

Each p_i goes through the q_j's in round robin order, trying to simulate their steps. Steps are grouped into **pairs**: a read and the following write.

When p_i begins the simulation for q_j, it uses its own input as the input for q_j. If p_i ever simulates a decision step by some q_j, it decides the same thing.

How do p_0 and p_1 keep their simulations consistent? They need to "agree" on the value of each q_j's local state after each pair of steps by q_j.

Impossibility of 1-Resilient Consensus (cont'd)

For q_j's k-th pair, p_0 and p_1 each have a flag variable:

- Assume q_j's $(k-1)$-st pair has been computed.
- p_i calculates its suggestion for q_j's state after the k-th pair (see below).
- p_i checks if p_{1-i} has made a suggestion for this state of q_j.
- If not, then p_i sets its flag to 1.
- If so, then p_i sets its flag to 0.

Interpretation of flags:

1. If p_i's flag is 1, then p_i is the **winner**.
2. If both are 0, then consider p_0 the **winner**.
3. If one is 0 and the other is not yet set, then winner is not determined.
4. If neither is set yet, then winner is not determined.
5. Not possible for both to be 1.

In situations 1 and 2, q_j's k-th pair is said to be **computed**; otherwise not.

Impossibility of 1-Resilient Consensus (cont'd)

How does p_i calculate its suggestion for q_j's state after q_j's k-th pair?

- p_i gets q_j's state after its $(k-1)$-st pair:
  - If $k-1=0$, then use q_j's initial state with p_i's input.
  - Otherwise get the suggestion of the winner for q_j's $(k-1)$-st pair.
- Consult q_j's state (just obtained) to determine which q_r's register is to be read in its k-th pair.
- Get current value of q_r's register by finding largest m such that q_r's m-th pair has been computed and get the winning suggestion.
- Apply q_j's transition function to get the value of q_j's state after its k-th pair.

Impossibility of 1-Resilient Consensus (cont'd)

Each admissible execution of A' (by p_0 and p_1) simulates an execution of A (by q_0 through q_{n-1}). If p_i observes a q_j to make a decision in the simulated execution, then p_i makes the same decision.

If the simulated execution of A is admissible, then it satisfies

- termination: eventually the q_j's decide
- agreement: the q_j's make the same decision
- validity: if all q_j's have input v, then the decision is v. Recall that the simulated execution's inputs are based on the real execution's inputs.

And thus the actual execution of A' would be correct.

Impossibility of 1-Resilient Consensus (cont'd)

Why is the simulated execution admissible? We must show that at least $n - 1$ processors take an infinite number of steps in it.

How can a simulation of a q_j be blocked? If p_0 or p_1 crashes during its simulation of q_j's k-th pair, for some k. For example:

- p_0 writes a suggestion and crashes.
- p_1 sees p_0's suggestion and writes 0 to its flag.
- p_0's flag remains unset forever.
- Therefore q_j's k-th pair is never computed.

But the crash of one p_i can only block the simulation of one q_j! In the example, p_1 will be able to progress on its own with simulating the steps of all processors other than q_j.

$\square$

Impossibility of Consensus in Message Passing

1. Assume there exists an n-processor, 1-resilient consensus algorithm A for message passing.
2. Use A as a subroutine to design an n-processors 1-resilient consensus algorithm A' for shared memory with read/write registers.
3. Previous result shows A' cannot exist.
4. Thus A cannot exist.

Idea of A': Simulate message channels with read/write registers. Then run A on top of these simulated channels to get A'.

For channel from p_i to p_j, have a register written by p_i and read by p_j:

- p_i "sends" a message m by writing the old value of the the register with m appended.
- p_j "receives" a message by reading the register and checking for new values at the end.

Randomized Consensus

To try to get around some of the negative results presented so far for consensus, we can

- weaken the termination condition: nonfaulty processors must decide with some nonzero probability;
- keep the same agreement and validity conditions.

This version of consensus is solvable

- in asynchronous message passing
- with Byzantine failures
- when $n > 3f$ (optimal)

We will show a weaker result, an algorithm:

- in asynchronous message passing
- with crash failures
- when $n > 3f$ (however, $n > 2f$ is optimal)
- constant *expected* time to termination

Common Coin

Similar to phase king algorithm, but instead of using the king's majority value as the tie-breaker, use a **common coin**.

Abstract properties of a common coin subroutine:

- no inputs
- output is 0 or 1 at each processor
- for every admissible adversary and initial configuration, the probability that all nonfaulty processors get the same output is at least ρ (the **bias** of the coin).

One way to implement a common coin subroutine is: each processor flips an independent coin. The bias of this method is $\rho = \frac{1}{2^n}$.

The bias of the common coin affects the expected time of the entire algorithm: bigger bias gives faster algorithm.

Randomized Consensus Schema

```
initially r = 1 and pref = input
while true do
   send <vote,pref,r> to all
   wait for >= n-f messages of the
                   form <vote,*,r>
   let v be the majority of phase r
                             votes
   if all phase r votes are v then
      decide v
   send <outcome,v,r> to all
   wait for >= n-f messages of the
               form <outcome,*,r>
   if all phase r outcomes are w then
      pref := w
   else pref := common-coin()
   r := r+1
endwhile
```

Proving agreement and validity is similar to phase king.

Termination of Randomized Consensus

Lemma (14.8): For each $r \geq 1$, the probability that all nonfaulty processors decide in phase r is at least ρ.

Proof Sketch: Fix any admissible adversary and initial configuration.
Case 1: All nonfaulty processors set their preferences in phase $r - 1$ using the common coin.

- With probability $\geq \rho$, they all get 0 and with probability $\geq \rho$, they all get 1.
- With probability $\geq 2\rho$, all preferences are same at the end of phase $r - 1$.
- Results on agreement and validity show they all decide in the next phase, r.

Termination of Randomized Consensus (cont'd)

Case 2: Some nonfaulty processor sets its preference in phase $r - 1$ without using the common coin.

- That processor sees all outcomes the same, v.
- Results on agreement and validity show no other nonfaulty processor sets its preference to a value other than v based on outcome messages in phase $r - 1$.
- With probability $\geq \rho$, a processor that sets its preference using common coin in phase $r - 1$ will get v.
- With probability $\geq \rho$, all preferences are the same at the end of phase $r - 1$.
- Results on agreement and validity show they all decide in the next phase, r.

□

Termination of Randomized Consensus (cont'd)

- Expected number of phases until termination is $\frac{1}{\rho}$, since number of phases is a geometric random variable.
- If T is the time per common coin execution, the expected time is $O(T/\rho)$.
- With the simple common coin, where $\rho = \frac{1}{2^n}$ and $T = O(1)$, the expected time is $O(2^n)$. Not good.
- Can use a more complicated common coin with $\rho = 1/8$ and $T = O(1)$, giving an expected time of $O(1)$.
- However for this common coin to work, we must assume a **weak** adversary, which cannot see the *contents* of messages.