Lecture 13: Synchronizers

Outline: Synchronizers.

Sources: Sections 11.3, 18.1 and 18.3.

Recitation material: More details of synchronizer Zeta. It is also possible to cover the routing algorithm from Section 18.2.

Discuss the modeling issues (the two models, and the notions of local simulation).

Present the concept of 'safety' in a round.

Describe synchronizer Alpha.

It is possible to describe synchronizer Beta (in class, one student suggested using a 'synchronization server', which I turned into Beta, for a non-complete graph).

A good point to show here is that Beta keeps the whole graph synchronized while Alpha lets 'distant' nodes proceed at different rates.

Define clusters, and what it means for a collection of clusters to cover a graph.

Now assume we have a cover with the following property:

For each node there is a cluster including its 1-neighborhood.

Pick one of these clusters as the home cluster. Pick a cluster root for each cluster, and construct a cluster tree around it.

Now explain the algorithm.

Analyze its complexity as a function of the volume and the diameter of the cover.

Explain what coarsening is.

Show Theorem 18.1 and its proof.

Use Algorithm 14.3. This uses a reliable broadcast algorithm (with message diffusion).

Define the terminology: A processor prefers v in phase r if prefer$_i$ is v in Lines 1-7 of phase r; for phase 1 the preference is the input. If the assignment of the preference for the round r is in Line 8 then p_i deterministically prefers v in phase r.

First show that a common preference yields decision.

Lemma 14.4. If all processors reaching phase r prefer v in phase r, then all nonfaulty processors decide on v no later than phase r.

This gives validity, right away.

Then show the 'converse': if some processor decides, then all processors prefer the same value.

Lemma 14.5. If some processor decides on v in phase r, then all nonfaulty processors either decide on v in phase r or prefer v in phase $r + 1$.

We also show that the only way processors can disagree is by seeing different outcomes of the common coin (or by one using the coin and the other picking a preference deterministically).

Lemma 14.6. If some processor deterministically prefers v in phase $r + 1$, then no processor decides on $\bar{v}$ in phase r or deterministically prefers $\bar{v}$ in phase $r + 1$.

Now look at the earliest phase some processor decides. No nonfaulty processor decides on $\bar{v}$ in phase r_0 or in any earlier phase. All nonfaulty processors prefer v in round $r_0 + 1$, and finally, all nonfaulty processors decide on v no later than round $r_0 + 1$. Thus, the algorithm satisfies the agreement condition.

Discuss real termination (stopping to echo messages).

The probability of deciding depends on the bias of the common coin.

Lemma 14.8. The probability that all nonfaulty processors decide by phase r is at least ρ.

This bounds the expected time complexity of the algorithm...

12.3 The randomized common coin algorithm

Depending on the quality of the class, can either explain the common coin with exponential bias, or the more sophisticated algorithm giving a common coin with constant bias.

Lecture 12: Randomized consensus

Main points today:

- How randomization helps to 'solve' unsolvable problems and to improve the complexity 'despite' lower bounds (if we modify termination conditions carefully).
- For a key problem!

Outline: Randomized consensus for message passing (crash failures).

1. Overview (and definition of the problem).
2. The deterministic phase algorithm.
3. The randomized common coin algorithm.

Sources: Section 14.3.

Recitation material: Discuss the general structure of randomized consensus in shared memory systems (from [19]), with an exponential common coin.

12.1 Definition of the problem and overview

We want to achieve the following conditions:

Agreement: Nonfaulty processors do not decide on conflicting values.

Validity: If all the processors have the same input, then any value decided upon must be that common input.

Termination: All nonfaulty processors decide with some nonzero probability.

Agreement and validity are as before, but the termination condition was relaxed. We typically want to have termination with probability one. We also care about the expected time until termination.

We show an asynchronous randomized consensus algorithm for f crash failures, $n > 3f$. The algorithm terminates in constant expected time.

The algorithm has two parts: one part is a deterministic phase algorithm managing processors' individual preferences for decision to reach agreement (if possible), the second part is a randomized algorithm for a common coin, which is used to break ties.

12.2 The deterministic phase-based voting algorithm

In each phase, a processor votes on its preference for this phase, which is a binary value, and then obtains an estimate of the tally; it is possible that the tally is undecided. If the tally is unanimous, then the processor decides on the tally, otherwise the processor obtains its preference for the next phase by 'flipping' a common coin.

An f-resilient common coin with bias ρ is a procedure (with no input) that returns a binary output; For every admissible adversary and initial configuration, all nonfaulty processors executing the procedure output v with probability at least ρ (for v equal to 0 or 1).

- Increment sequence number, *seq*.
- Send a $\langle newval,v,seq \rangle$ to all processors.
- Wait for $(n+1)/2$ acks.

To read:

- Send $\langle request \rangle$ to all processors.
- Wait for $(n+1)/2$ $\langle value,v,seq \rangle$ messages.
- Pick the value with maximal sequence number.

Ping-pong protocol is needed to know which request is being answered.

(No need for ping-pong if the reader also sends sequence numbers with its request, and the answer is tagged with the sequence number of the request.)

Describe the ping-pong mechanism.

State and explain Lemma 10.19.

Prove Lemma 10.20 showing that a read returns the value of the latest preceding or overlapping write. This can be used to show that ordering operations by sequence numbers (as done in the previous lecture) is a linearization.

What to do if there are several readers?

- use the simulation of multi-reader from single-reader, OR
- imitate this simulation (and save an extra layer of sequence numbers).

Lecture 11: Wait-free atomic snapshots

Outline:

1. Bounded implementation of atomic snapshots.
2. Fault-tolerant implementation of read/write in message passing.

Sources: Sections 10.2 and 10.4.

No recitation this week.

11.1 Bounded snapshot implementation

Use handshake bits (and handshake procedures) to detect operations.

Explain the handshake procedures (Algorithm 10.4) and explain the handshaking properties.

Note Handshake Property 2: If it returns false than no tryHS by the other processor is completely enclosed between its own prior tryHS and checkHS.

Explain why toggle bits are needed.

Present Algorithm 10.5, and go over Lemmas 10.13 and 10.14.

Then explain the linearization; here we identify a point inside the interval of the operation, rather than creating a sequence.

- Updates are are linearized at their writes.
- Direct scans are linearized after the last read in the first collect in the successful double collect.
- Indirect scans are linearized at the same point as the direct scans.

By Lemma 10.14, the linearization points are inside the interval.

Prove Lemma 10.15.

Finally, we need to show the the implementation is wait-free. But notice that each 'failure' of a double collect can be attributed to (at least) one processor. After a failure is attributed three times to the same processor, the scan returns. So, at most $3n$ double collects.

This also proves the implementation takes $O(n^2)$ reads and writes.

11.2 Fault-tolerant simulation of read/write in message passing

The point is that we can port all the shared memory algorithms to work in message passing, if less than half the processors fail. If more processors fail, then can cause *partition* of the network and processors will not see updated values.

We will concentrate on one single-writer single-reader register. These simulations can be composed and combined.

A writer just sending a message to the reader is no good:

- May be lost...
- Cannot wait for an ACK...

We use the other processors as extra storage, and rely on *quorums.*

To write v:

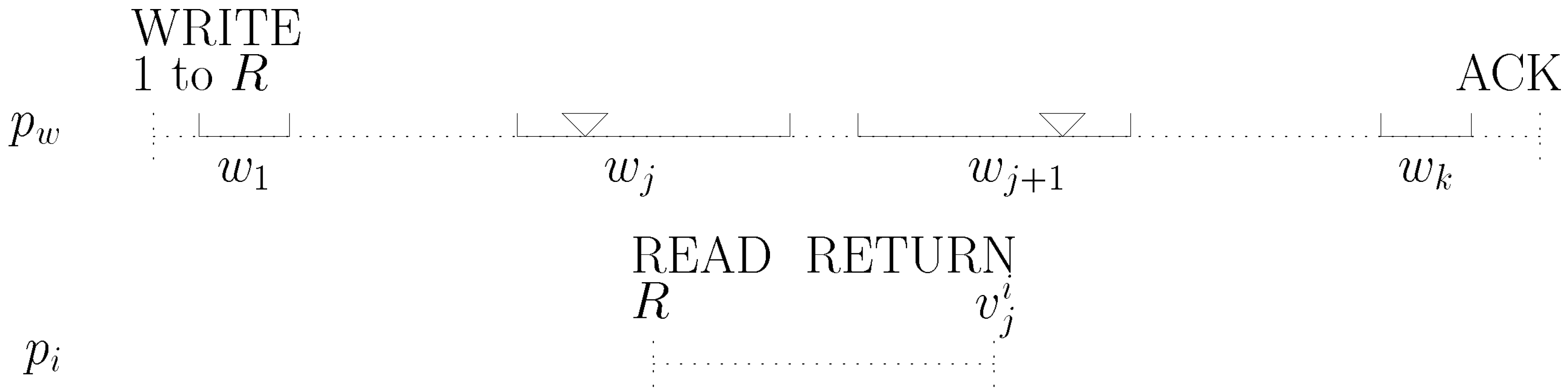


Figure 10.5

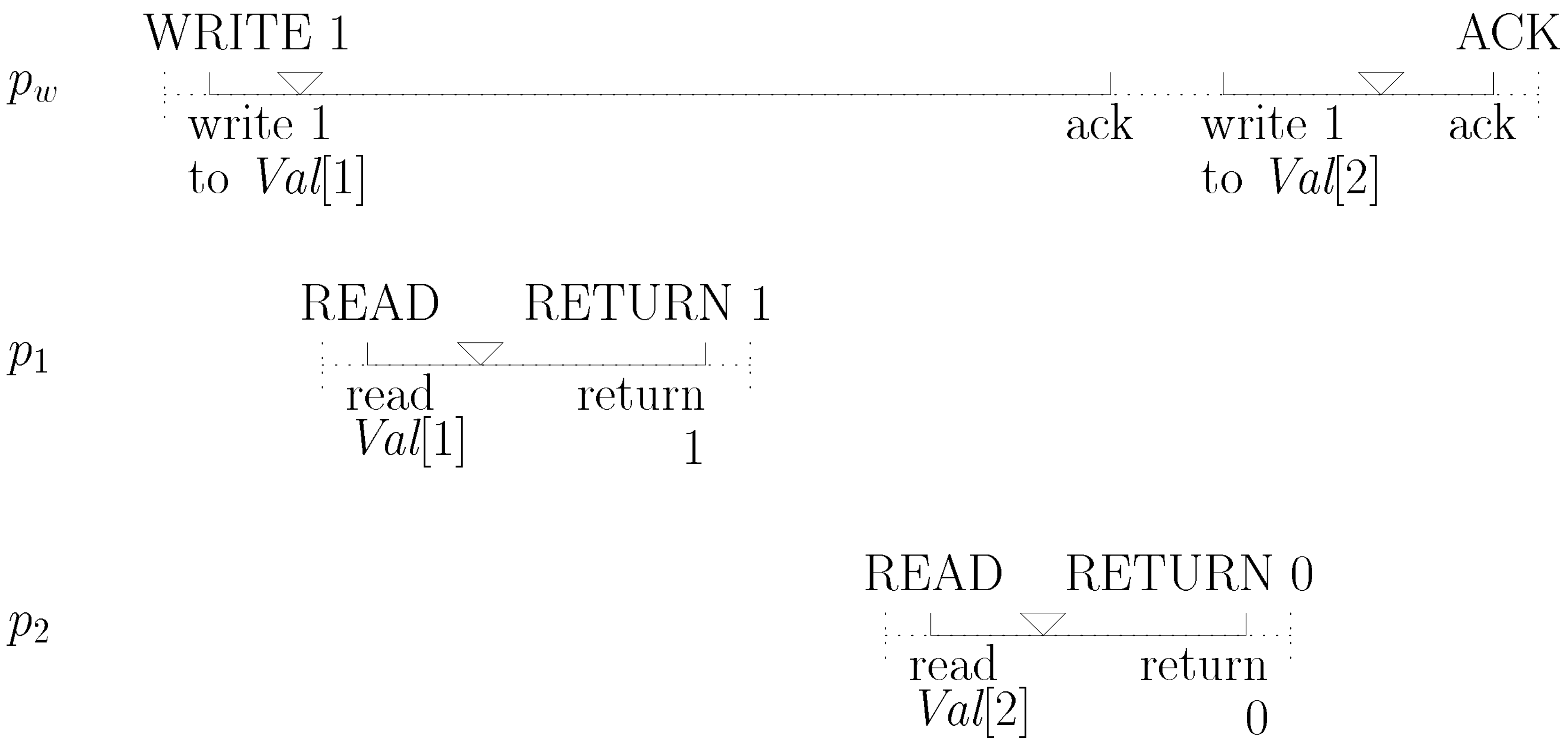

WRITE 1
ACK
p_w
write 1 to $Val[1]$
ack
write 1 to $Val[2]$
ack
READ
RETURN 1
p_1
read $Val[1]$
return 1
READ
RETURN 0
p_2
read $Val[2]$
return 0

Figure 10.4

Show Algorithm 10.3.

To prove its correctness, create the sequence explicitly according to vector timestamps.

10.5 Atomic snapshots

Specification: An object with n segments; a single processor updates each segment, but all processors can read all segments (at once!).

Show some correct and incorrect executions.

Requires ordering among the scanners (unlike distributed snapshots, some of you may be familiar with).

Idea: Collect (read all segments), and collect again. If not change observed, this was a snapshot. Otherwise, can 'blame' some updater. If many failures, some updater made many operations. Embed a scan in the update, and write it with the new value.

Show simple implementation with sequence numbers.

Lecture 10: Wait-free simulations of shared-memory objects

Today we concentrate on wait-free simulations of shared memory objects.

Sources: Chapter 10.

Recitation material: Lattice agreement and its relation to snapshots, based on:
H. Attiya, M. Herlihy and O. Rachman, "Atomic Snapshots Using Lattice Agreement," *Distributed Computing*, Vol. 8 (1995), No. 3, pp. 121–132.

10.1 Overview

We have seen several types of objects that can be read and written: with a single or many writers, with a single or many readers. Today, we justify why we can 'switch' between the various objects, by showing they simulate each other. We also show how to simulate a snapshot object. (By results of two lectures ago, we cannot expect to simulate much stronger objects.)

We'll define what it means to simulate in the model and will show how to:

1. Simulate multi-reader from single-reader.
2. Simulate multi-writer from single-writer.
3. Simulate reading many segments (but writing to a single segment), from multi-reader single-writer.

The first two simulations require unbounded size; the third does not.

10.2 Linearizability

- Processors interact correctly with the object implementation (one invocation at a time).
- A nonfaulty processor eventually gets a response.
- There is a way to linearize all completed operations (and some of the pending operations) so that the responses are according to the semantics of the implemented object and the order preserves the order of non-overlapping operations.

10.3 Multi-reader from single-reader

Simple idea is that the writer writes a value to each reader. Explain new-old inversions (Figure 10.4).

Prove that if readers do not write then implementation is not correct (Theorem 10.4, use Figure 10.5).

Show Algorithm 10.2.

To prove its correctness, create the sequence explicitly according to writer's timestamps.

10.4 Multi-writer from single-writer

Explain why writers need to order with respect to each other. Use multi-entry timestamps. (Could use single values, as in bakery algorithm, will be an exercise.)

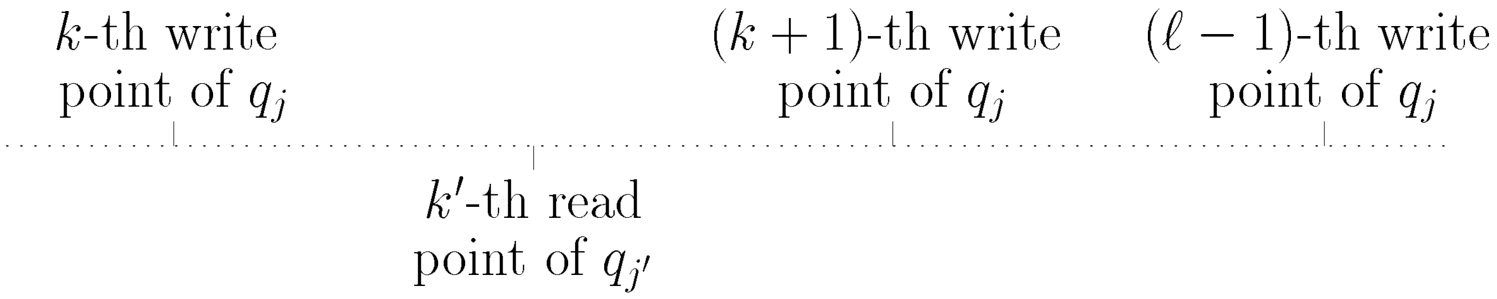

k-th write point of q_j
$(k+1)$-th write point of q_j
$(\ell-1)$-th write point of q_j
k'-th read point of $q_{j'}$

Figure 5.13

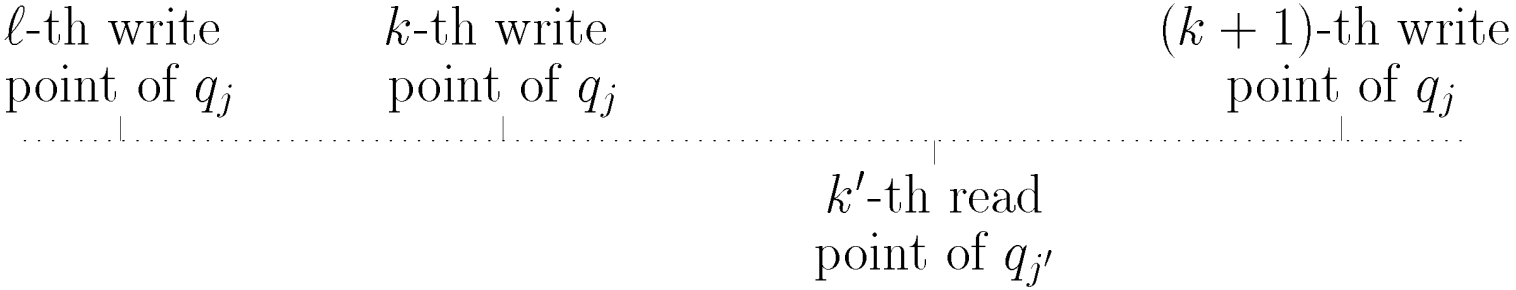

ℓ-th write point of q_j
k-th write point of q_j
$(k+1)$-th write point of q_j
k'-th read point of $q_{j'}$

Figure 5.12

is never true, it must be that p_{1-i} never sets its flag, i.e., it fails after Line 10 but before Line 11 or 12 of its simulation of q_{j_1}'s k_1-th pair. But this contradicts the fact that p_{1-i} fails during the simulation of q_{j_0}, not q_{j_1}. Q.E.D.

Comment: We are not done yet!!! This does not necessarily mean that the processor will eventually decide correctly, or even decide at all. This will follow only if we show that the codes are simulated *correctly*.

To show simulation, we need to construct an execution β of the simulated processors, $q_0, \ldots, q_{n-1}$, with at most one failure, and the same transitions as computed in α. This will suffice.

For every simulated processor q_j, and every $k \leq 1$, we identify two points in α:

- The *read point* is when the winner of the k-th pair of q_j returns from the last call to computed in Line 2 of get-read.
- The *write point* is when the winner of the k-th pair of q_j sets its flag to 1, or, if neither flag is ever set to 1, when the second simulating processor writes 0 to its flag.

Comment: Discuss atomicity problems and use of snapshots.

Let σ be the sequence of simulated processor indices corresponding to the read and write points.

Define an initial configuration C_0, in which the input value of each q_i is the input value of the simulating processor that is the winner for the first pair of q_i (use 0 if there's no winner).

β is the execution of $q_0, \ldots, q_{n-1}$ obtained by starting with C_0 and applying computation events in the order specified by σ. The next lemma is proved in the book using Figures 5.12 and 5.13.

Lemma 5.23. The values returned by get-read in α are consistent with the read and writes points defined.

Lemma 5.24. Let q_j be a processor that executes at least k pairs in β. Then in α,

1. eventually computed(j, k) is true, and, after that point,
2. the value of *Suggest*$[j, k, w]$, for the winner, is equal to the value of q_j's state (and shared register) after its k-th pair in β.

Putting the last two lemmas together, we get that the algorithm correctly simulates an n-processor consensus algorithm with two processors. Thus, if there is a 1-resilient consensus algorithm for n processors, there there is a 1-resilient consensus algorithm for two processors. But a 1-resilient consensus algorithm for two processors is wait-free, and thus, it cannot exist.

this step. Thus, it cannot be that it is stopping the progress on some other simulated processor's code! As we show in a minute, in this case p_0 will make progress on the other codes.

9.3 In more detail...

We make the following assumptions on the simulated algorithm (without loss of generality):

1. each processor q_j can write to a single shared register
2. the code of each processor q_j consists of alternating read steps and write steps, beginning with a read.
3. each write step of each processor q_j writes q_j's current state into q_j's shared register.

Thus, the simulated code works in 'super-steps' in which q_j reads the state of some other processor $q_{j'}$, changes its local state, based on its previous state and the value read, and writes its new state. We emphasize that super-steps are not atomic: The read and the write steps are separate.

Show the code (Algorithm 5.3).

9.4 Correctness proof

Since each processor first writes a suggestion, and then checks the other processor's suggestion we have:

Lemma 5.20. At most one flag is set, for each simulated step.

k is a *computed pair* of q_j if either both flags are set or one of them is not set and the other one equals 1. The *winner* of the k-th *computed* pair of q_j is the simulating processor whose flag is 1 or p_0, otherwise. By the above lemma, the winner is well-defined.

winner is only called for computed pairs, and returns the id of the winner; get-state and get-read return the value of the winner.

If one processor simulates a pair on its own (before the other processor executes line 10) then its flag will be set to 1. Thus, it will be the winner for this step, and can decide on the result of this pair even if the other processor never shows up.

We can argue progress:

Lemma 5.22. If p_i never fails or decides, then the values of its lastpair[j] variable grow without bound, for all j except possibly one.

Proof. p_i's lastpair[j_0] variable is stuck at k_0, for some j_0. Thus, p_i wrote 0 to Flag[j_0, k_0, i] and never finds Flag[$j_0, k_0, 1-i$] set. Thus, p_{1-i} crashes after writing Suggest[$j_0, k_0, 1-i$] (Line 10) and before writing Flag[$j_0, k_0, 1-i$] (Line 11 or 12).

Assume p_i also fails to make progress on q_{j_1}, for some $j_1 \neq j_0$, and let k_1 be the highest value reached by p_i's lastpair[j_1] variable.

Thus, computed(j_1, k_1) is never true, and neither Flag[j_1, k_1, i] nor Flag[$j_1, k_1, 1-i$] is ever set to 1. As a result, p_i executes Lines 6 through 11 of its simulation of q_{j_1}'s k_1-th pair. p_i executes Line 11 of its simulation of q_{j_1}'s k_1th pair after p_{1-i} executes Line 10 of its simulation of q_{j_1}'s k_1-th pair (or else p_i would set its flag to 1). Thus p_i finds the other processor's suggestion already set and sets its flag to 0. Since computed(j_1, k_1)

Lecture 9: Wait-free simulation of fault-tolerant algorithms

Today we prove how to simulate an n-processor algorithm by two processors, while preserving the number of failures.

Sources: Section 5.3.2.

Recitation material: More details, and extension to any number of processors k (assuming snapshots). The last topic is non-trivial, see [60].

9.1 The goals

There is a direct proof for this case (extending ideas presented for the wait-free case); this proof was presented in the recitation. However, there is a more interesting proof, by *reduction.*

- Assume (by way of contradiction!) there is an asynchronous consensus algorithm A for n processors, $q_0, \ldots, q_{n-1}$, and a single failure.
- From A we construct an algorithm A' which solves consensus among two processors, p_0 and p_1, and a single failure.
- By the result proved in the last lecture, A' cannot exist.
- Thus, A does not exist.

Why is this not simple? If p_0 simulates half of the processors, and p_1 simulates the other half of the processors, then a failure of a single simulating processor leads to a failure of a majority of simulated processors!

9.2 Overview

p_0 and p_1 go through the codes of $q_0, \ldots, q_{n-1}$, in round-robin order, and try to simulate their computation, one step at a time. They use their inputs as input for each simulated code. Once a decision is made by some simulated code, then this decision is taken as output by the simulating processor, which then stops the simulation. If they try to simulate the same step of the same simulated processor, then they need to agree on the result.

Comment: Agreement?! We have just showed it cannot be done... But we are not going to get the agreement to be wait-free for each simulated code, but to be wait-free overall.

Agreement is achieved by an algorithm very similar to the asymmetric mutual exclusion algorithm we have seen.

- A simulating processor writes a suggestion for a particular step, and checks to see if the other processor has written a suggestion for the same step.
- If the other processor has not yet written a suggestion, then the first processor declares itself as the winner (by setting a flag to 1) and its suggestion is used henceforth for this step.
- Otherwise, if the other processor has already written a suggestion, the first processor sets its flag to be 0.
- If both processors set their flags to 0, then the suggestion of p_0 is used.

Sometimes, it is not clear who wins, for example, if p_0's flag is 0 and p_1's flag is not set. But then, p_1 is stuck somewhere on

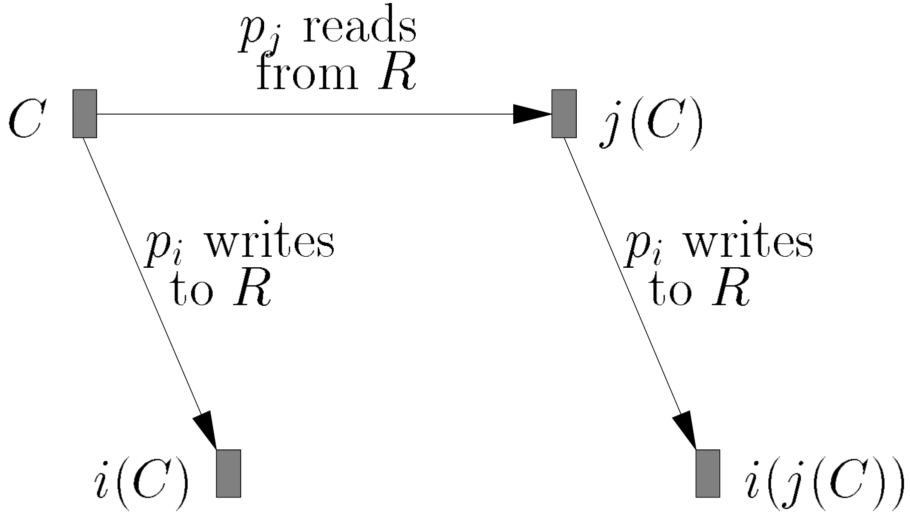


Figure 5.11

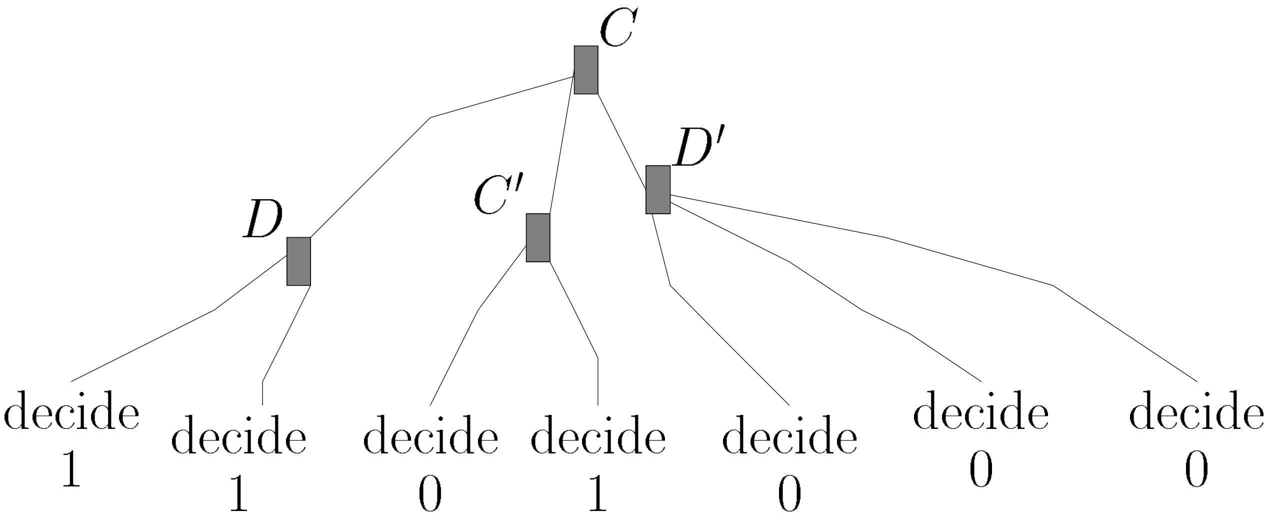

C
D'
C'
D
decide 1
decide 1
decide 0
decide 1
decide 0
decide 0
decide 0

Figure 5.10

Why does it work? Clearly, only one test&set returns 0, and the other (the later one) returns 1. The first processor takes its input, and the other processor takes the other input (which is already written).

This already means that we cannot implement test&set from read/write operations by a wait-free algorithm. Otherwise, we could simulate the above wait-free consensus algorithm for two processors using only read/write operations.

What about more than two processors?

Theorem . There is no wait-free consensus algorithm for three processors, using only read, write and test&set operations.

Proof. The same general structure as in the proof for read/write. The crucial part is to prove the equivalent of Lemma 5.18.

So, assume C is a bivalent configuration, and that all processors are critical from C. Without loss of generality, it must be that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent. (We do not care about p_2...)

We do a case analysis on the operations of p_0 and p_1 from C. If they commute (e.g., operations to different variables), then we are done as in Lemma 5.18.

Suppose they over-write each other (e.g., two test&set operations or two write operations). Then the state of the memory in $p_1(p_0(C))$ is the same as in $p_1(C)$. Thus, $p_1(p_0(C)) \overset{p_2}{\sim} p_1(C)$, and we can apply Lemma 5.16 to obtain a contradiction. Q.E.D.

So, test&set is not enough. What about other operations? For example, compare&swap suffices to solve consensus for any number of processors. (Whoever gets the variable first, sets the decision.)

```
$v$ := compare&swap(Dec,bot,input)
if $v$ = bot
    then decide input
    else decide $v$
```

What does it mean? That we cannot implement compare&swap for three processors from test&set, read, and write operations by a wait-free algorithm. Otherwise, we could simulate the above wait-free consensus algorithm for three processors using only test&set, read, and write operations.

By the way, cannot wait-free solve consensus for three processors also with queues and stacks, or fetch&add (!!!).

The set of values reachable from C contains either one value or two. In the first case, we say that C is *univalent*, meaning that the decision is already set (but perhaps the processors do not know it yet...). C is *0-valent* or *1-valent* according to the specific value. In the second case, we say that C is *bivalent*, meaning that the decision is still open. (Use Figure 5.10.)

Recall the notion of *similarity* we used for mutex lower bounds.

Suppose we have two univalent configurations, C_1 and C_2, such that $C_1 \overset{p_i}{\sim} C_2$, for some process p_i. Then, apply a p_i-only schedule to C_1. If C_1 is v-valent, p_i must decide on v. Apply the same schedule to C_2. Clearly, p_i decides on the same value v, and hence C_2 must also be v-valent. This shows:

Lemma 5.16. If C_1 and C_2 are univalent and $C_1 \overset{p_i}{\sim} C_2$, for some p_i, then C_1 and C_2 have the same valence.

8.3 The wait-free case

Our strategy would be to construct an infinite execution in which all configurations are bivalent. To start the construction, we prove:

Lemma 5.17. There is a bivalent initial configuration.

Proof. Consider D_{01}, the configuration in which p_0 starts with 0 and p_1 starts with 1. If it's bivalent, then we are done. Otherwise, assume it is 1-valent, and consider D_{00}, the configuration where both processors start with 0, which is 0-valent by the validity condition. Since $D_{00} \overset{p_0}{\sim} D_{01}$, we have a contradiction to Lemma 5.16. Q.E.D.

A processor p_i is *critical* in a configuration C if C is bivalent and $p_i(C)$ is univalent.

That is, p_i causes a decision to be made in C, which is why we want to avoid such 'critical' steps. Luckily by case analysis depending on the type of operations, we prove the following lemma (see the book and use Figure 5.11).

Lemma 5.18. If C is bivalent, then at least one processor is not critical in C.

Therefore, at each intermediate bivalent configuration, there is a processor which can take a step and still keep the bivalence. Therefore, we can construct an infinite admissible execution, in which all configurations are bivalent.

Remark: We have long lectures, so this does not fill a whole lecture. On the other hand, explaining the simulation of Section 5.3.2 requires a full lecture. I use the rest of the time to present some implications.

8.4 What about other operations?

This material covers examples to Theorem 15.5.

For example, test&set can be used to wait-free solve consensus between two processors.

```
Algorithm for $p_i$:
  write input to $R_i$
  if test&set(Dec) = 0
     then decide $R_i$
     else decide $R_{1-i}$
```

Lecture 8: Impossiblility of wait-free consensus

Main points today:

- Consensus is impossible in asynchronous systems (with read/write).
- One reason why this is perhaps the most important result in this course.

Outline:

1. Impossibility of wait-free consensus among two processors (shared memory).
2. Some implications.

Sources: Section 5.3 and parts of 15. This is also the reading assignment for this week.

Recitation material: A direct proof for message passing systems with any n and $f = 1$ ([109]).

8.1 Overview

What about asynchronous systems? Cannot solve consensus even if there is only a *single* crash failure (regardless of the number of processors). Holds for

- message passing
- shared memory with read/write operations

We will prove the result by two reductions:

- Prove the impossibility result for shared memory, $n-1$ failures (easy since there are a lot of failures) including two processors, one failure.

  Show that two processors can simulate n processors, without increasing the number of failures. $\Longrightarrow$
- Impossibility result for shared memory, 1 failure.

  Show that shared memory can simulate message passing, without increasing the number of failures. $\Longrightarrow$
- Impossibility result for message passing, 1 failure

An asynchronous system with n processors that tolerates $n-1$ failures is called *wait-free*, since a nonfaulty processor does not wait for other processors. Here, an execution is *admissible* if at least one processor takes an infinite number of steps (so the execution is infinite). **(Note how simple this condition is!)**

8.2 Definitions and a basic lemma

The key to the impossibility result is considering the decisions that are still possible from a particular (reachable) configuration.

So consider all admissible executions from a configuration C. In each of these executions either 0 or 1 is decided eventually, by the termination condition, and only one of them, by the agreement condition.

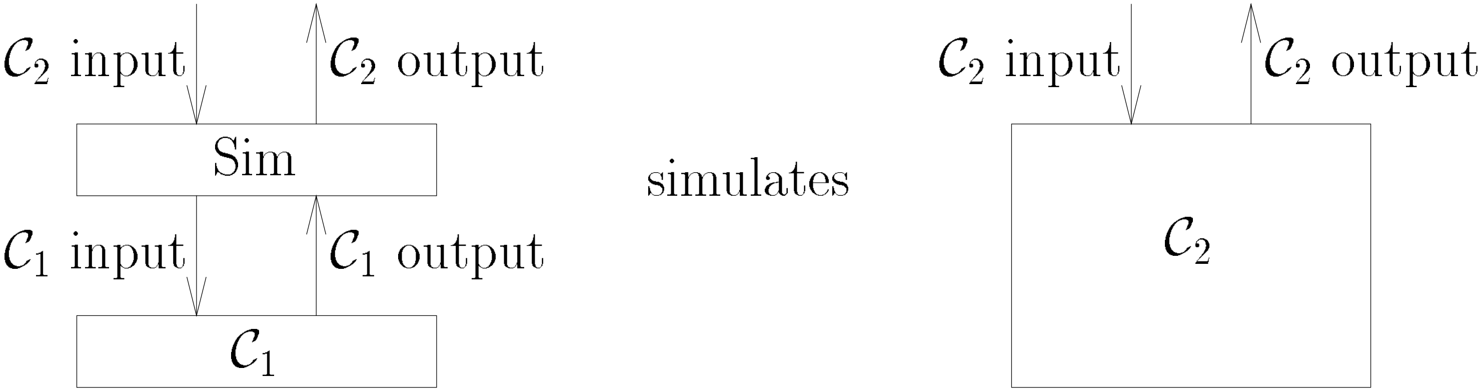

$\mathcal{C}_2$ input
$\mathcal{C}_2$ output
Sim
$\mathcal{C}_1$ input
$\mathcal{C}_1$ output
$\mathcal{C}_1$
simulates
$\mathcal{C}_2$ input
$\mathcal{C}_2$ output
$\mathcal{C}_2$

Figure 7.3

possible that p_j will not even know this has happened. To get around this problem, we require that $n > 2f$. In this case, if p_i decides not to crash itself, i.e., if it sees at least $n - f$ echoes of its own message, then, since $n - f > f$, at least one nonfaulty processor echoes p_i's message. Processors accept any echoed message they receive, even if they did not receive it directly.

Thus, each round k of the crash failure model translates into two rounds of the omission failure model, $2k - 1$ and $2k$, also denoted $(k, 1)$ and $(k, 2)$.

- In $(k, 1)$, a processor sends to all processors the message it is supposed to broadcast.
- In $(k, 2)$, processors echo the messages they have received.
- If a processor receives an echo of its own message from less than $n - f$ processors, then it crashes itself
- If a processor receives an echo of a message it did not receive directly, it accepts it.

The notion of simulation is similar to the previous simulation.

One claim to prove is that a nonfaulty never crashes itself.

The second claim to prove is that if a processor does not crash itself, then all nonfaulty processors receive its message.

should be received by all processors in the identical Byzantine model, although possibly with a one round lag. Thus, whenever a processor p_i sends a message m it also sends its *support* set, i.e., the messages that cause p_i to generate m. This set is used to check whether p_i's message is correct according to the algorithm; that is, inductively, if these messages are valid and p_i's message follows from them, then it is also valid.

This simulation requires knowledge of the application algorithm A.

I do not go into the details of the simulation (Algorithm 12.2).

The formal proof here has complications since the faulty processors in the real system C_1 (identical Byzantine) can still receive whatever they like, while in the simulated system C_2 (omission) we require integrity from all processors.

The key is the notion of *simulate with respect to nonfaulty processors*. That is, for each execution α of A with (S, C_1) there is an execution β of A on C_2, such that if a processor is nonfaulty in α (according to C_1) then it is nonfaulty in β (according to C_2). And all nonfaulty processors have the same view in α and β (when appropriately restricted to events of A).

Given some execution α, we define β, with the same set of nonfaulty processors.

The initial state of p_j in β is the content of any round 1 message from p_j that is validated by a nonfaulty processor in α. If no nonfaulty processor ever does so, then the state is arbitrary.

The state transitions of a processor in β are according to A.

In round k of β, processor p_j accepts message m from processor p_i if and only if, in α, some nonfaulty processor validates p_j's round $k+1$ message claiming that p_j accepted m from p_i in round k.

Although it may seem counter-intuitive, a message from p_j is received in β, only if some nonfaulty processor validates it in the next round of α. The key to the correctness of the construction is to prove, by induction on the rounds, that messages sent and received by nonfaulty processors are always received.

We do not go into the details of this proof.

7.5 Crash from omission

The properties we need to get are:

- Integrity.
- Nonfaulty Liveness.
- Faulty Liveness (if a nonfaulty processor does not receive at k, then no nonfaulty processor receives in $k+1$).

The idea is to make a processor 'crash' itself if it omits a message.

How can processor p_i detect it has omitted to send a message? We require processors to echo messages they receive; then, if some processor, say, p_j, does not echo p_i's message, then either p_i omitted to send this message, or p_j omitted to receive this message. If p_i receives less than $n - f$ echoes of its message, then it blames itself for not sending the message and crashes itself; otherwise, it blames p_j (and the other processors who did not echo the message), and continues.

Unfortunately, it is possible that p_i is faulty and omits to send a message only to a single nonfaulty processor, p_j; it is also

The key is to show for each execution of A on (S, C_1) a 'similar' execution of A on C_2. As we shall see, the definition of similar can be somewhat tricky.

7.3 Identical Byzantine from Byzantine

The properties we need to get are:

- Nonfaulty Integrity
- Faulty Integrity (Identical Contents)
- No Duplicates.
- Nonfaulty Liveness
- Faulty Liveness (Relay)

A nonfaulty processor receives (in the high-level sense) a message only when it us sure there are enough witnesses for this broadcast.

The simulation uses two rounds of the underlying Byzantine system to simulate each round of the identical Byzantine system. Round k of the identical Byzantine system is simulated by rounds $2k-1$ and $2k$ of the underlying Byzantine system, which are also denoted $(k, 1)$ and $(k, 2)$.

The simulation requires that $n > 4f$.

- To broadcast m in simulated round k, the sender, p_i, sends a message $\langle init,m,k \rangle$ to all processors in round $(k, 1)$.
- When a processor receives the first init message of p_i for round k, it acts as witness for this broadcast and sends a message $\langle echo,m,k,i \rangle$ to all processors in round $(k, 2)$.
- When a processor receives $n-2f$ echo messages in a single round, it becomes a witness to the broadcast and sends its own echo message to all processors if it has not already done so.
- When a processor receives $n-f$ echo messages in a single round, it accepts that message, if it has not already accepted a message from p_i for simulated round k.

(Can show Algorithm 12.1.)

Here we have the simplest notion of simulation: the same execution of A with (S, C_1) is also correct for C_2.

Theorem 12.2. The five conditions of identical Byzantine are satisfied.

(See proof in the book.)

Everybody sends to everybody in each round, so we only consider message size, which is something like n^2 bits (times the size of the message itself, round tag and processor id).

7.4 Omission from identical Byzantine

The properties we need to get are:

- Integrity (both faulty and nonfaulty) for all processors.
- Nonfaulty Liveness.

Allows both receive and send omissions.

We are almost there, only that the identical Byzantine model still allows a nonfaulty processor to send 'nonsense' messages, not according to the protocol. We validate messages before receiving them, exploiting the fact that the same messages

Lecture 7: Simulations between failure types

Main points today:

- Simulations between failure types.

Outline:

1. Problematics in going from Byzantine to crash, and outline.
2. Intuitive notion of simulation.
3. Identical Byzantine from Byzantine.
4. Omission from identical Byzantine.
5. Crash from omission.

No lower bounds today! Only synchronous systems.

Sources: Sections 12.1–12.5. This is also the reading assignment for this week, as well as Chapter 7.

Recitation material:

- Application to consensus
- Identical Byzantine in asynchronous systems.

7.1 Problematics in going from Byzantine to crash

Three factors make the Byzantine model different from the crash model:

A. Faulty processors do not send the same message to the nonfaulty processors.

B. Faulty processors may send messages that are not according to the algorithm.

C. Faulty processors may fail in more than a single round.

Our approach will be to handle A by simulating identical Byzantine (echoing messages and making sure all nonfaulty processors receive the same message), handle B by simulating omission failures (validating messages and making sure they are according to the algorithm), and handle C by simulating crash (forcing an omitting processor to crash itself).

One small problem is that validation requires to know which messages another processor received in the previous round, so as to know whether it has support for what it 'claims' in this round. Thus, identical Byzantine also include a *relay* property, forcing a processor to relay all messages it got.

In many cases, the relay property can be used to avoid validation altogether.

7.2 The notion of simulation

Intuitively it means that an algorithm A running on system C_1 with a simulation S, thinks that it is running on system C_2. (Use Figure 7.3.)

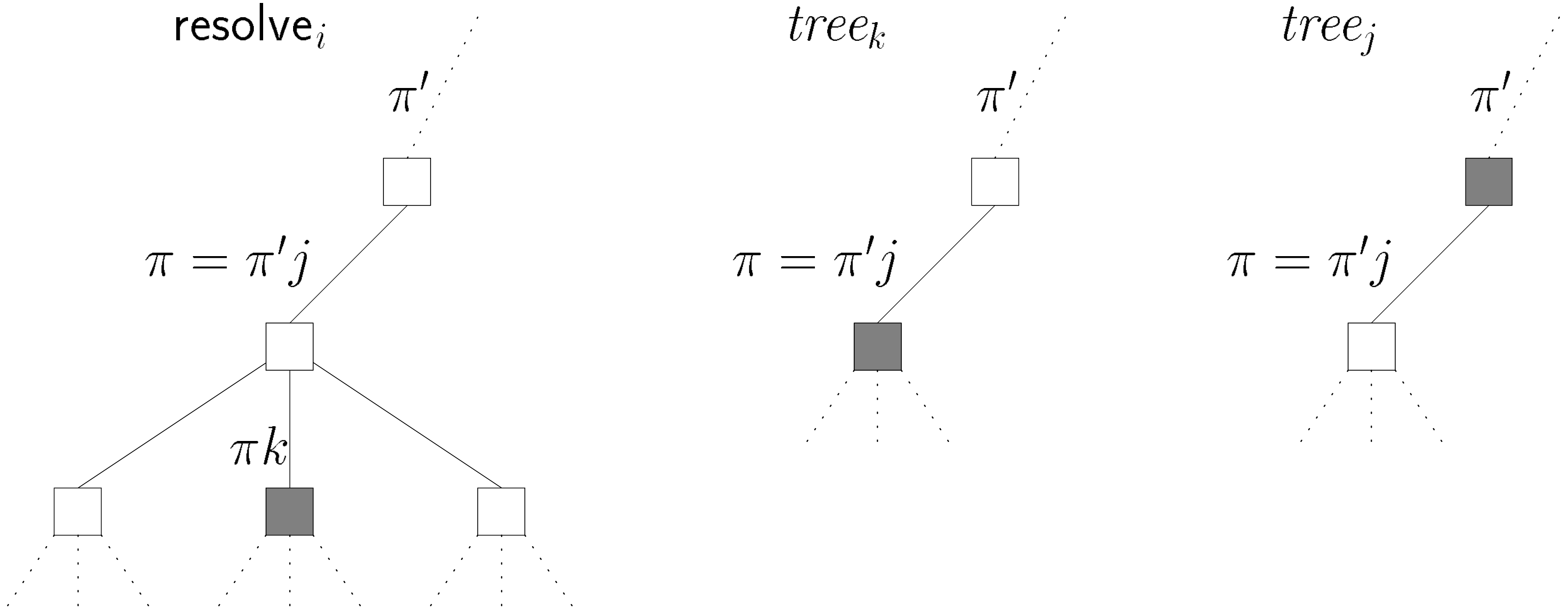


Figure 5.9

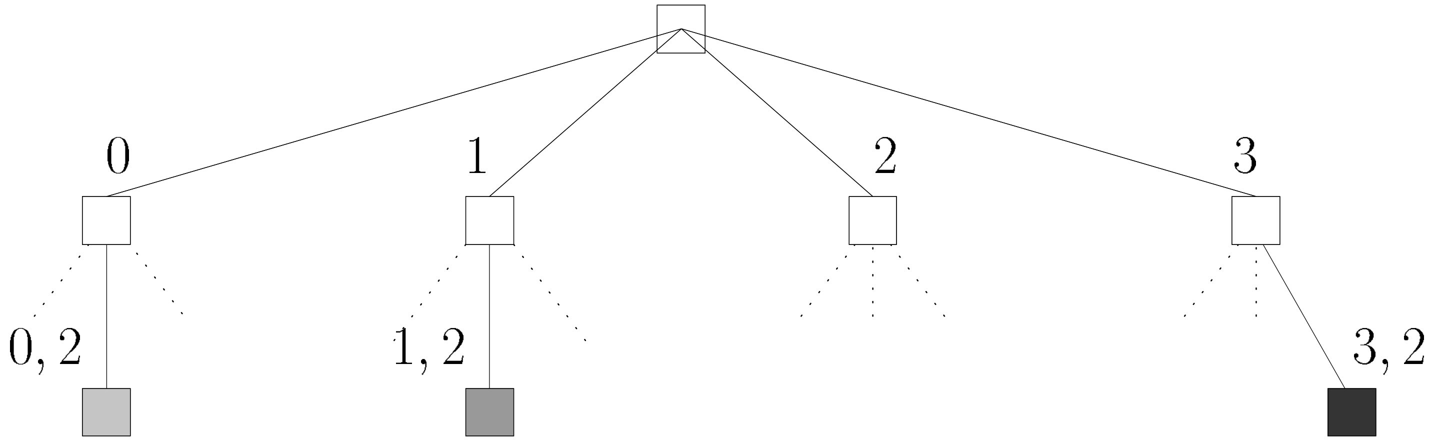


Figure 5.8

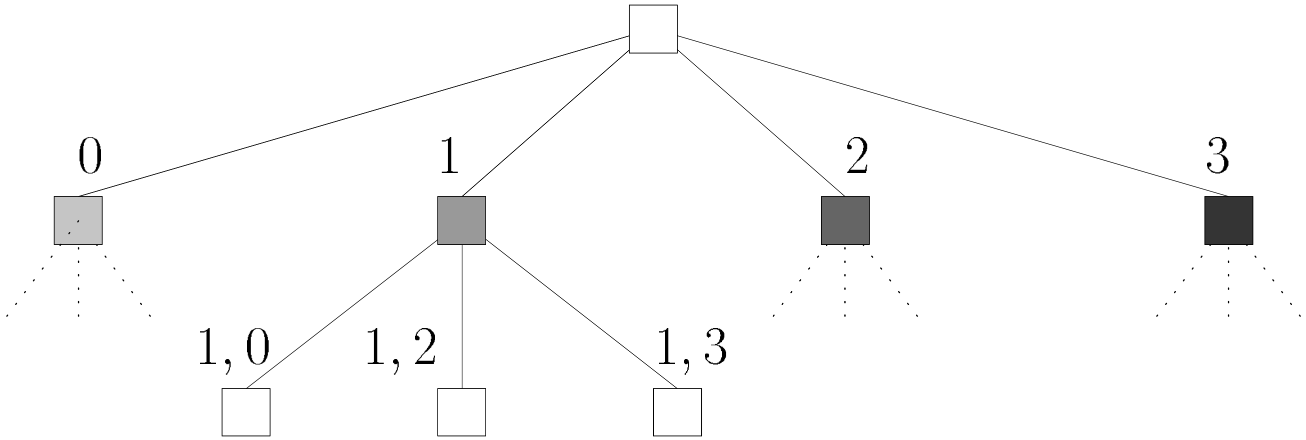
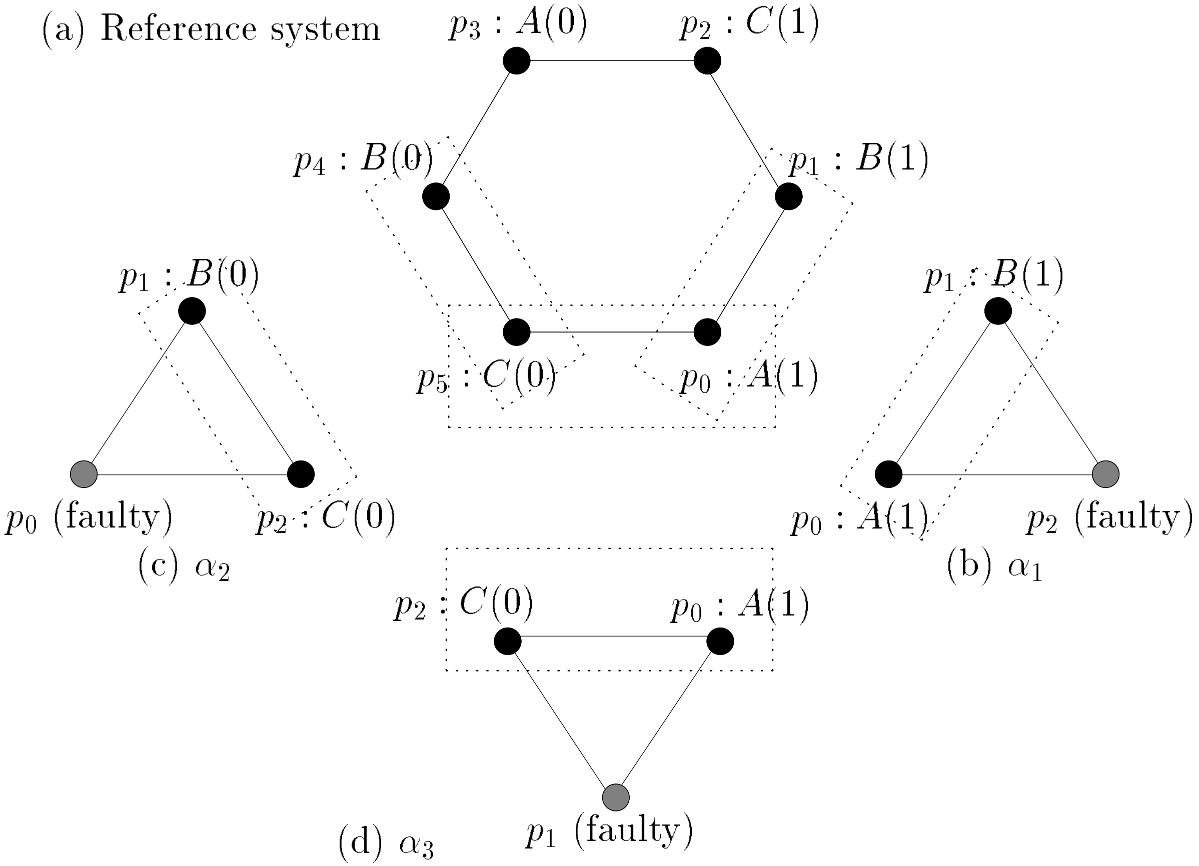


Figure 5.7



(a) Reference system
$p_3 : A(0)$
$p_2 : C(1)$
$p_4 : B(0)$
$p_1 : B(1)$
$p_5 : C(0)$
$p_0 : A(1)$
$p_1 : B(0)$
p_0 (faulty)
$p_2 : C(0)$
(c) α_2
$p_1 : B(1)$
$p_0 : A(1)$
p_2 (faulty)
(b) α_1
$p_2 : C(0)$
$p_0 : A(1)$
p_1 (faulty)
(d) α_3

Figure 5.6

Lemma 5.11. If there is a common frontier in the subtree rooted in a node v, then v is common as well.

Proof. Once again, the proof is by induction on the height of v in the tree. The base case is when v is a leaf, which is obvious.

In the induction step, consider a node v with a common frontier at height $k+1$. Each of its children is at height k, and clearly, is either common or has a common frontier. Thus all nonfaulty processors resolve the same value for all v's children, and thus the same value for v itself. Q.E.D.

In particular, the root is common, which means that all nodes decide on the same value.

Explain why this algorithm has long messages.

They store this information in a tree with height $f+1$ (each path from root to leaf has $f+2$ nodes).

Each node is labeled with a sequence of processor id's.

- The root is labeled with the empty sequence.
- If a node is labeled with $i_1, ..., i_r$, then its children are labeled with $i_1, ..., i_r, i$, for each processor id i not in this sequence (that is, for each $i \in \{0, .., n-1\} - \{i_1, ..., i_r\}$).

Construct an example for $n = 7$ and $f = 2$.

Each node corresponds to the processor p_i whose id appears last in sequence labeling the node; intuitively, a node corresponding to p_i in the tree held by p_j represents information held by p_i, as known to p_j. The third level of the tree in Figure 5.8 contains the second level of the tree of Figure 5.7.

In the first stage: Collect information ($f+1$ rounds), as follows:

- First round: Store input in the root of the tree, send to all. Store inputs received from other processors in level 2 of the tree (default if no legal value received).

  Note that in the first round we fill two levels of the tree.
- Round r: Send the r-th level of your tree to all. Store messages received from other processors in level $r+1$ of the tree. When receive a value labeled with $i_1, ..., i_r$ from p_i store it in the node labeled $i_1, ..., i_r, i$.

Intuitively, $i_1, ..., i_r, i$ is the value that
"p_i says that p_{i_r} says that .. that p_{i_2} says that p_{i_1} said".
This value is called tree$_i(i_1, ..., i_r, i)$.

As an example, fill in the information for the above tree.

In the second stage, decide. (After $f+2$ levels of the tree were filled in.)

Apply a recursive majority voting function resolve:

- For a leaf v, resolve$_i(v)$ =tree$_i(v)$ ($\perp$ if no value).
- For a non-leaf node v, resolve$_i(v)$ is the majority of resolve$_i(v\ j)$ over all v's children j; $\perp$ if no majority exists.

Apply the resolve function for the above tree, and decide on resolve$_i$ computed for the root of the tree. (Compute resolve for the above example.)

By induction on the height of v in the tree (use Figure 5.9), prove the next lemma.

Lemma 5.10. If a node v corresponds to nonfaulty processor p_j (is labeled σj) then for any nonfaulty processor p_i resolve$_i(\sigma j)$ = tree$_j(\sigma)$.

The lemma uses the fact that $n > 3f$.

We can already prove validity. Assume all processors have the same input v (and in particular, all nonfaulty processors have the same input). For every child of the root, corresponding to a nonfaulty processor, the lemma implies that we resolve to its input value. Then the majority is v and we decide on v.

A node is *common* if all nonfaulty processors compute the same resolve value for it. A subtree has a *common frontier* if there is a common node on each path from the root of the subtree to a leaf.

- One has to change, but not both.
- But what if p_1 is faulty? How can p_0 know?

What about three processors (and one failure)?

- p_0 has 0, p_1 has 1 and p_2 is faulty.
- p_2 is the tie breaker but can behave maliciously (double-faced to p_0 and to p_1).

These are not proofs, just indications of the complications. But we can prove:

Theorem 5.8. For $f = 1$ and $n = 3$ there is no consensus algorithm in the presence of Byzantine failures.

Proof. Assume such an algorithm exists, let

- A be the local algorithm for p_0,
- B be the local algorithm for p_1, and
- C be the local algorithm for p_2.

Buy two copies of the algorithm, and consider a synchronous ring with six processors, p_0 to p_5.

- p_0 and p_3 run A
- p_1 and p_4 run B
- p_2 and p_5 run C

(Consider Figure 5.6(a).)

Note that the algorithm is not supposed to work correctly in this situation, but it does produce some behavior which we use to specify other behaviors. We assign inputs as in Figure 5.6, and let β be the resulting execution.

Consider execution α_1 in Figure 5.6(b). p_0 and p_1 must decide 1.

Consider execution α_2 in Figure 5.6(c). p_1 and p_2 must decide 0.

Consider execution α_3 in Figure 5.6(d). p_0 and p_2 must agree... but also must decide as in α_1 and α_2... Q.E.D.

The lower bound can be extended to any number of processors to show that $n > 3f$ is necessary for solving consensus in the presence of f Byzantine failures.

The idea is that if such an algorithm exists, it can be simulated by three processors—each processor simulates all the steps (and the messages) of $n/3$ processors. If one processor fails, at most $n/3$ processors fail... (See Theorem 5.9 for more details; the students can read them at home.)

6.4 An exponential algorithm for consensus with Byzantine failures

This algorithm takes $f + 1$ rounds (which is optimal) and requires $n > 3f$ (optimal). It does send very long messages... In the recitation we cover an algorithm with constant-size messages (but twice the number of rounds).

This is a 'full information' algorithm in which processors try to tell each other all they know (and how they learned it).

Lecture 6: Consensus in the presence of Byzantine failures

Main points today:

- Modifying the model to allow Byzantine failures.
- Show that a larger ratio of processors is needed for consensus.
- With this ratio, there is a solution.

Outline:

1. Definition of the Byzantine failures.
2. Lower bound $n > 3f$ (for $f = 1$, $n = 3$).
3. Exponential algorithm.

Sources: Sections 5.2. This is also the reading assignment for this week.

Recitation material: The polynomial algorithm (Section 5.2.5).

6.1 Modifying the model to include Byzantine failures

In each admissible execution there exists a subset of at most f *faulty* processors. In a computation step of a faulty processor, the new state and the messages sent a completely unconstrained.

We assume that each processor (faulty or not) takes a step in each round, and that messages sent in a round are delivered in the same round.

Note that crashes can be 'simulated' by faulty processors not doing anything (and not sending messages).

We assume that messages sent by faulty processor look 'reasonable'; a faulty processor can send garbage, but this can easily be detected.

6.2 Modifying the problem definition

Has the same conditions as for crash failures (remind):

Termination: Nonfaulty processors eventually decide.

Agreement: Nonfaulty processors decide on the same value.

Validity: If all processors have the same input value v, then this is the value decided.

Here however, the validity condition is not equivalent to requiring that the decision value is the input of some processor.

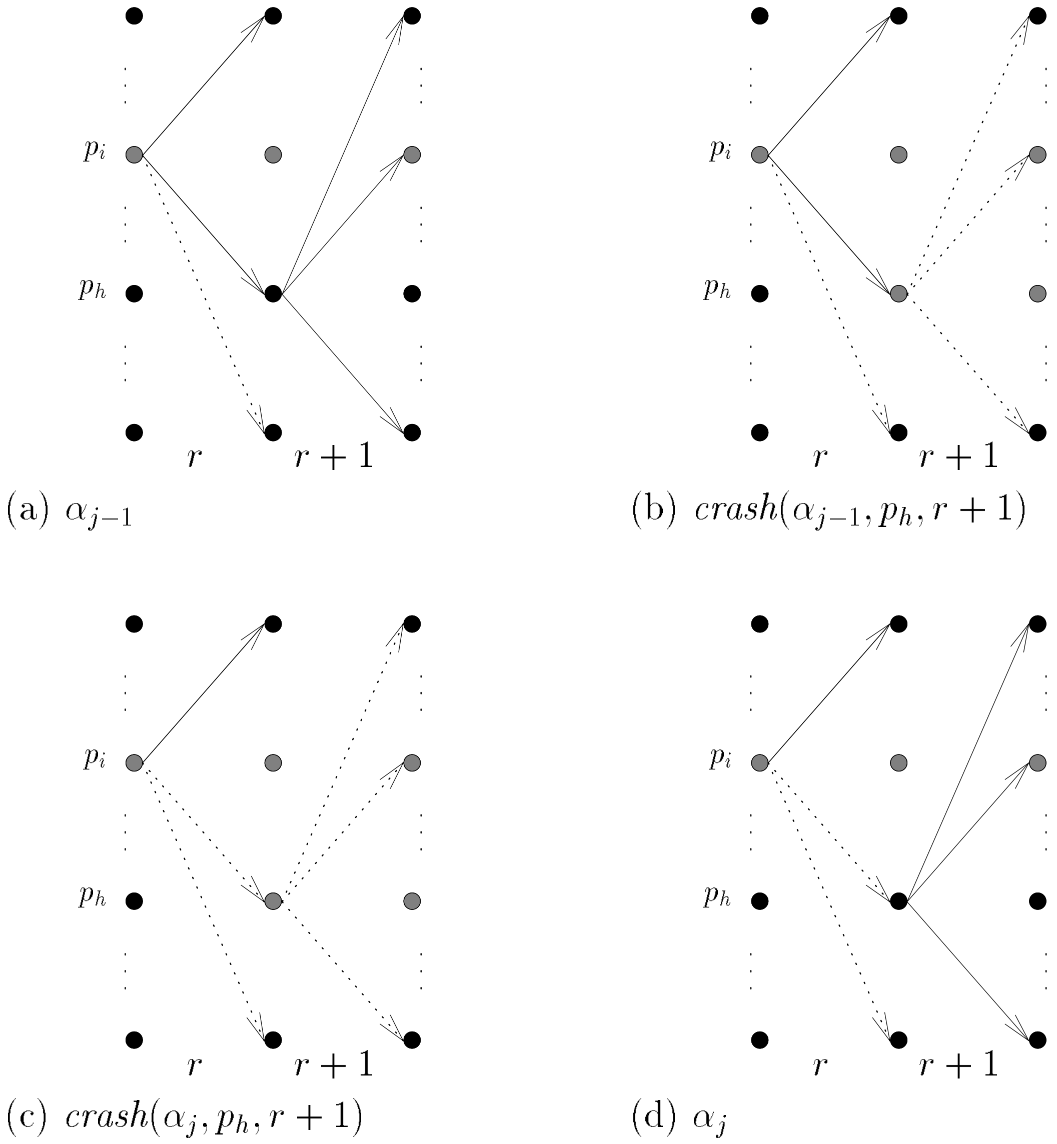
Again, this is a good point to challenge the students to suggest algorithms for this problem.

6.3 Lower bound on the ratio of faulty processors

Think about $f = 1$.

For crash failures, we could get consensus for two processors; what about Byzantine failures?

- p_0 has 0, p_1 has 1.



(a) α_{j-1}

(b) $crash(\alpha_{j-1}, p_h, r+1)$

(c) $crash(\alpha_j, p_h, r+1)$

(d) α_j

Figure 5.5

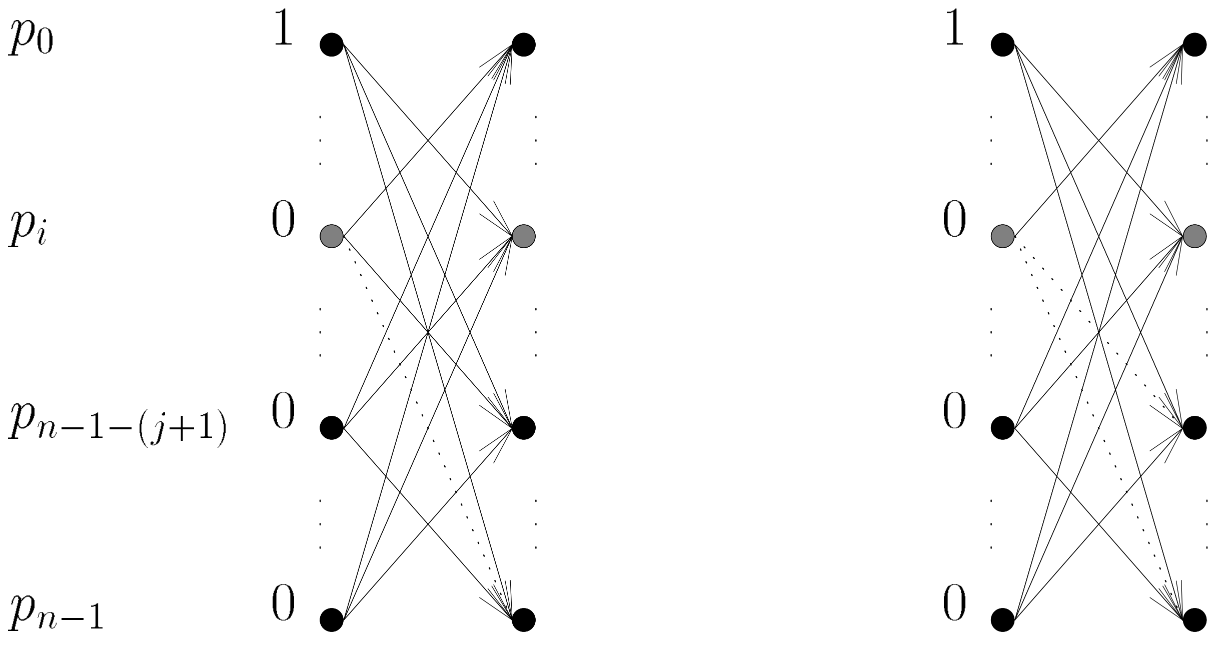


Figure 5.3

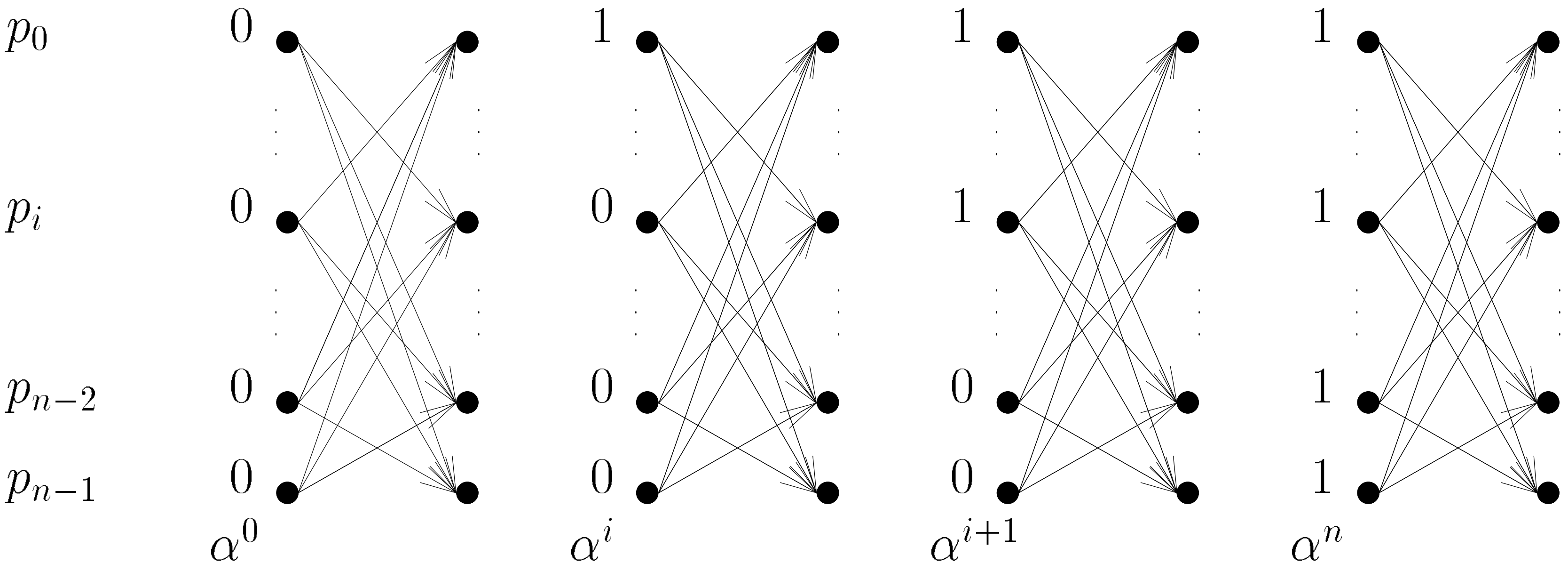

p_0
p_i
p_{n-2}
p_{n-1}
0
0
0
0
α^0
1
0
0
0
α^i
1
1
0
0
α^{i+1}
1
1
1
1
α^n

Figure 5.2

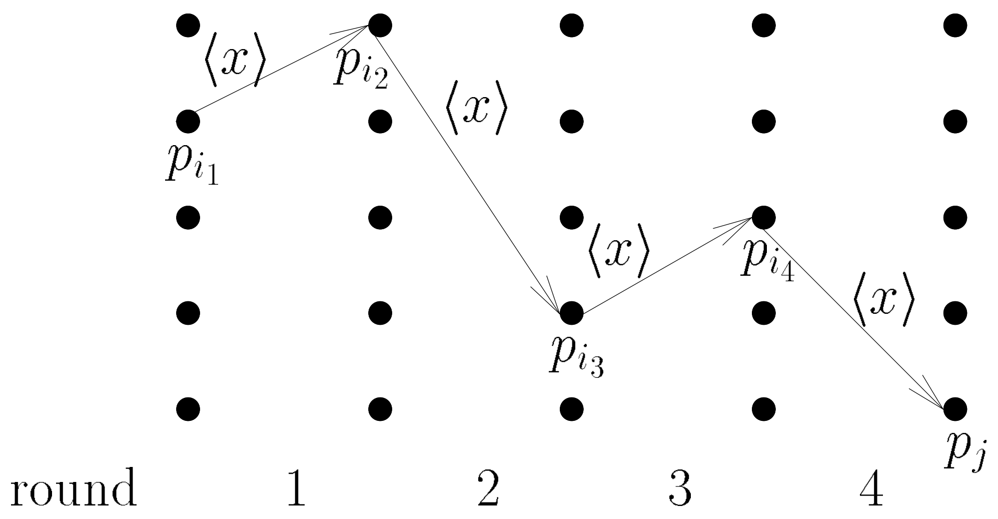

$\langle x \rangle$
p_{i_1}
p_{i_2}
$\langle x \rangle$
$\langle x \rangle$
p_{i_3}
p_{i_4}
$\langle x \rangle$
p_j
round
1
2
3
4

Figure 5.1

We already have a non-trivial result: More than one round is needed to solve consensus in the presence of a single failure.

5.4 $f+1$ rounds lower bound for f failures

Let's consider what happened in the above proof. There, the first round was also the last round, so there was no 'time' to report the omission of a message.

But this would not work if we have more than one round!

Remember that what we were doing is making p_i crash cleanly in round 1 (so we can change its input without anybody noticing). But what if there was some magical way to make an arbitrary processor p_j crash cleanly in round 2 (without using too many failures)?

Then we could follow the single round lower bound, only using this magical way on the execution in which p_i sends a message, and then using this magical way on the execution in which p_i does not send this specific message.

But what is this magical way? Well, it's the same thing we want to do in round 1 itself

$\Longrightarrow$ INDUCTION!!!

Present the definition of crash(α,p_i,r) (and its pre-conditions). (Right before Lemma 5.6.)

Lemma 5.6. For any execution α, processor p_i and round r, $\alpha \approx crash(\alpha, pi, r)$.

(This requires appropriate quantifications, I do not detail them at first, and then add them in retrospect.)

Proof. By backwards induction on r (the round number).

The base case, $r = f+1$, is similar to the single processor case studied before. If have time, can elaborate here and repeat the argument.

For the inductive step, assume the lemma holds for $r+1$, and prove it for round r. This is again similar to a single step in going from α^i to α^{i+1} in the single-failure case. (Use Figure 5.5.) Q.E.D.

A sticky thing in this proof is *accounting*: Making sure there aren't too many failures in the executions we construct.

This is a point where the exact definition of the model is important; we assume each round starts with sending all pending messages, then a computation step, which typically generates pending messages (to be delivered in the next round).

Termination and validity are trivial. The interesting property to prove is agreement (use Lemma 5.1).

If some nonfaulty processor p_i has x in V_i at the end of round $f+1$, then consider the first round in which x was inserted into V_i. If this is before $f+1$, then p_i sends x to all other processors, and all nonfaulty processors add it to their V.

Otherwise, consider the path along which x was transfered to p_i: $q_1, ..., q_{f+1}$ (use Figure 5.1). Since each processor sends a value at most once, these processors are distinct. At least one of these processors is nonfaulty, since there are at most f faulty processors. This processor sends x to all processors in this round, which means they all receive it.

Therefore, all nonfaulty processors have the same set of values at the end of round $f+1$, and pick the same minimum.

In the worst situation for the above algorithm, a value is "bumped" from one processor to another for $f+1$ rounds. Note that it is possible that this will happen, and thus, in our algorithm, a processor cannot halt before round $f+1$.

We next show this is the case for all algorithms solving this problem.

5.3 2 rounds lower bound for a single failure

Let's start with a special case. By way of contradiction, we assume there is a consensus algorithm for a single failure with one round.

The key is building a chain of similarities between the all-zeroes no-faults execution and the all-ones no-faults execution.

Two executions α and α' are *similar* with respect to a nonfaulty processor p_i if it has the same view in α and α'. This is denoted $\alpha \overset{p_i}{\sim} \alpha'$.

Clearly, in this case, p_i has the same decision in α and α', and by agreement, the same is true for all nonfaulty processors.

$\alpha \approx \alpha'$ is the transitive closure of $\overset{p_i}{\sim}$.

Clearly, if $\alpha \approx \alpha'$ then all nonfaulty processors decide on the same value in α and α'.

Consider the no-faults single-round executions α^i, $i = 0, .., n-1$, such that $p_0, ..., p_{n-i}$ start with 0 and others start with 1. (Figure 5.2.)

We want to show that $\alpha^0 \approx \alpha^n - 1$, so we show $\alpha^i \approx \alpha^{i+1}$. What we want to do is change p_i's input from 0 to 1. We cannot do it at once (since all other processors will notice that something happened and will, perhaps, change their decision value).

Instead, we omit p_i's messages in α^i one by one (use Figure 5.3). In each of these steps, there is at least one processor (other than p_i and the receiver of the message) that does not see any difference. This is because there is no additional round for the receiver to tell anything to anyone. If $n \geq 3$, this gives us the desired (direct) similarity.

Then in the resulting execution we flip p_i's input from 0 to 1. (Nobody sees a difference except p_i.)

Then, we do the same from α^{i+1}. We get the same execution!

Note that in each execution between α^i and α^{i+1} only p_i is faulty. So there is at most one failure in all executions.

Lecture 5: Consensus in the presence of crash failures

Main points today:

- Modifying the model to allow failures.
- Introduce the consensus problem.
- Show its inherent difficulty (already in this simple setting).

Outline:

1. Definition of the consensus problem (for crash failures).
2. Algorithm for crash failures (in $f + 1$ rounds).
3. 2-rounds lower bound for a single failure.
4. $f + 1$ rounds lower bound for f failures.

Can consider to start with the Byzantine story (Section 5.2).

Sources: Section 5.1. This is also the reading assignment for this week.

Recitation material: More details on the lower bound (including, as a special case, $f = 2$).

5.1 The consensus problem (crash failures)

We consider today only the synchronous message passing model, and modify it to allow simple failures, by *crashing*.

A processor crashes in round r if it sends all messages until round $r - 1$, in round r, it sends a subset of the messages it was supposed to send, and in any round later than r, it does not send any message.

There is a bound, denoted f (for *faulty*), on the number of crashes that can happen. It is not known in advance which processors will be faulty, nor even how many will be faulty.

Crashes are not necessary *clean*: In the round it crashes, a processor may send some messages to some processors, but not to others. With clean crashes, the problem is easy.

Each processor has an input value (x_i) and an output value (y_i), initially empty. A processor decides by writing to y_i.

The following properties are required, in every admissible execution (with fewer than f failures):

Termination. Nonfaulty processors eventually decide.

Agreement. Nonfaulty processors decide on the same value.

Validity. If all processors have the same input value v, then this is the value decided.

It is useful to open a discussion of possible consensus algorithms at this point.

5.2 A simple algorithm ($f + 1$ rounds)

(Use Algorithm 5.1.)

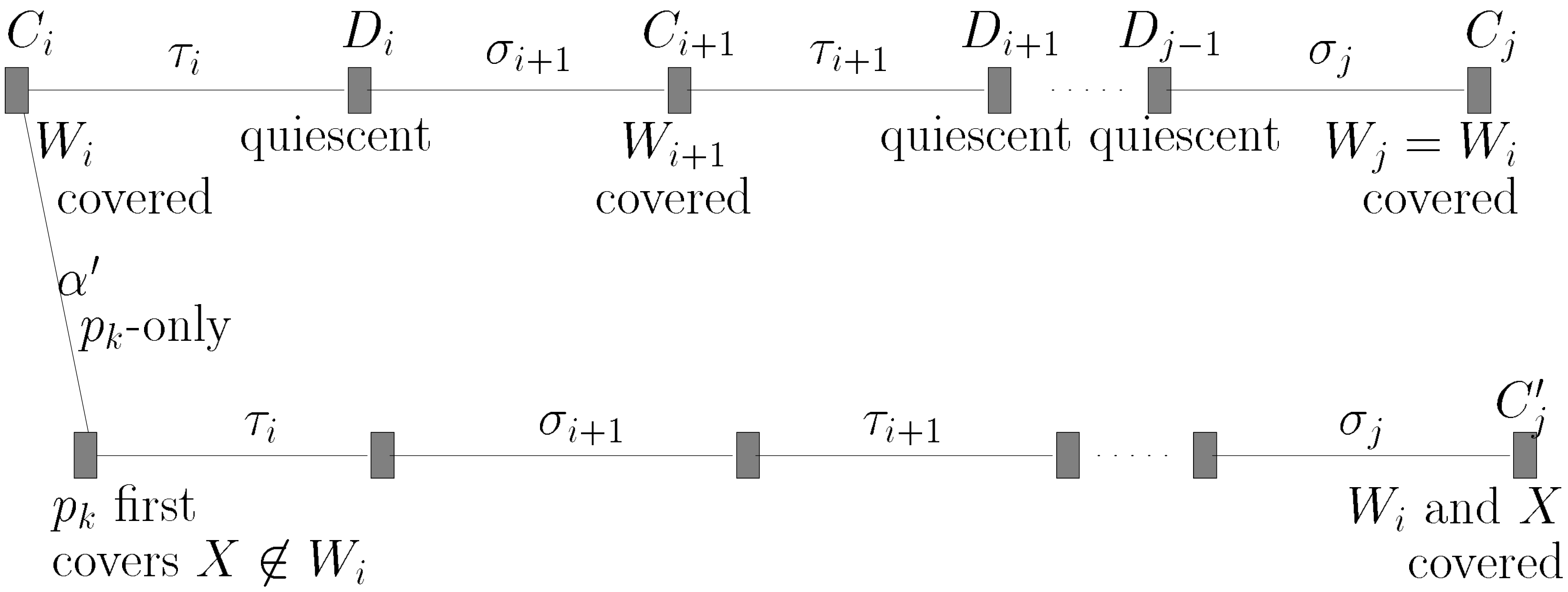

C_i
τ_i
D_i
σ_{i+1}
C_{i+1}
τ_{i+1}
D_{i+1}
D_{j-1}
σ_j
C_j
W_i covered
quiescent
W_{i+1} covered
quiescent
quiescent
$W_j = W_i$ covered
α'
p_k-only
τ_i
σ_{i+1}
τ_{i+1}
σ_j
C'_j
p_k first covers $X \notin W_i$
W_i and X covered

Figure 4.10

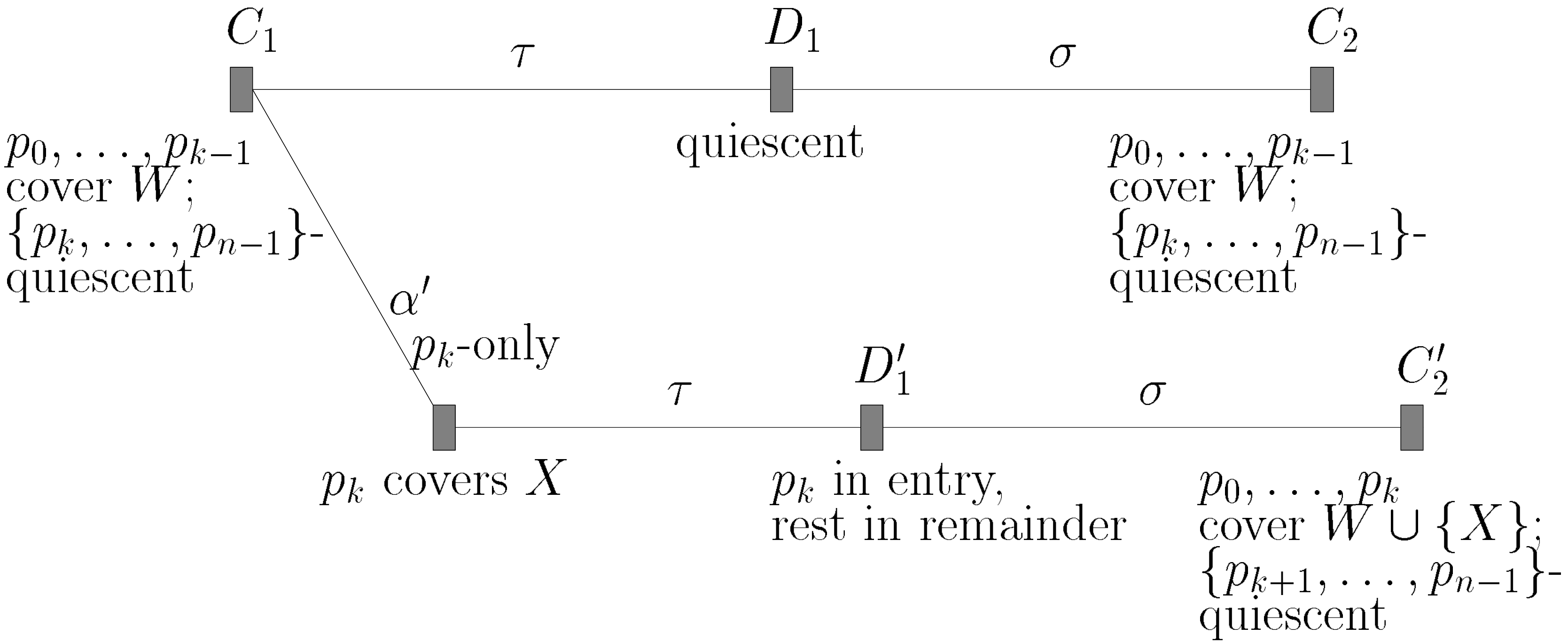

C_1
τ
D_1
σ
C_2
$p_0, \ldots, p_{k-1}$ cover W; $\{p_k, \ldots, p_{n-1}\}$-quiescent
quiescent
$p_0, \ldots, p_{k-1}$ cover W; $\{p_k, \ldots, p_{n-1}\}$-quiescent
α'
p_k-only
τ
D'_1
σ
C'_2
p_k covers X
p_k in entry, rest in remainder
$p_0, \ldots, p_k$ cover $W \cup \{X\}$; $\{p_{k+1}, \ldots, p_{n-1}\}$-quiescent

Figure 4.8

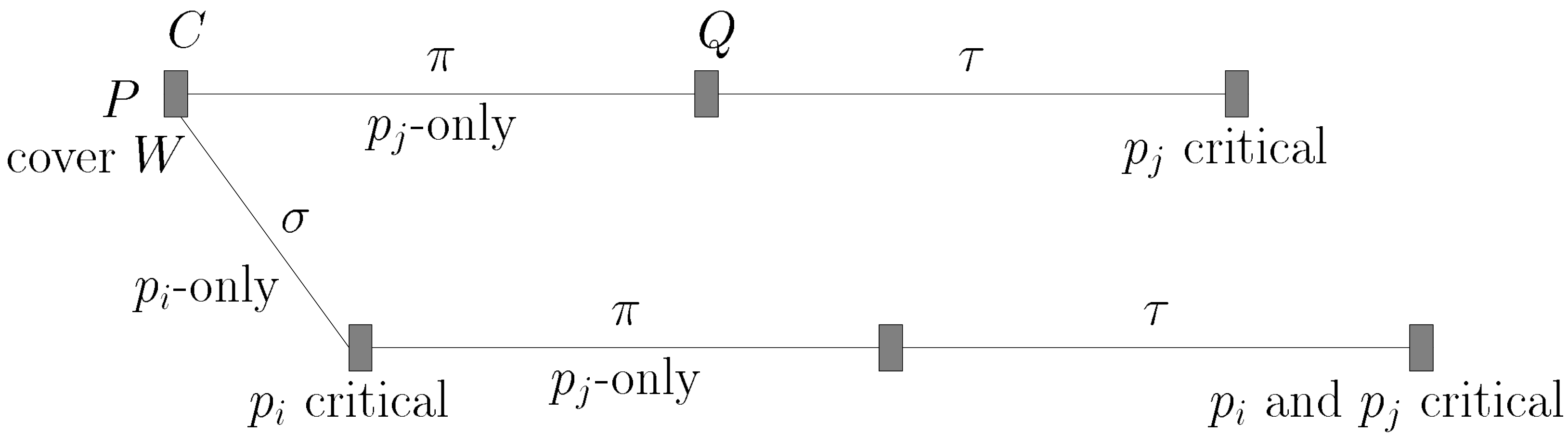

C
π
Q
τ
P
p_j-only
cover W
p_j critical
σ
p_i-only
π
τ
p_i critical
p_j-only
p_i and p_j critical

Figure 4.7

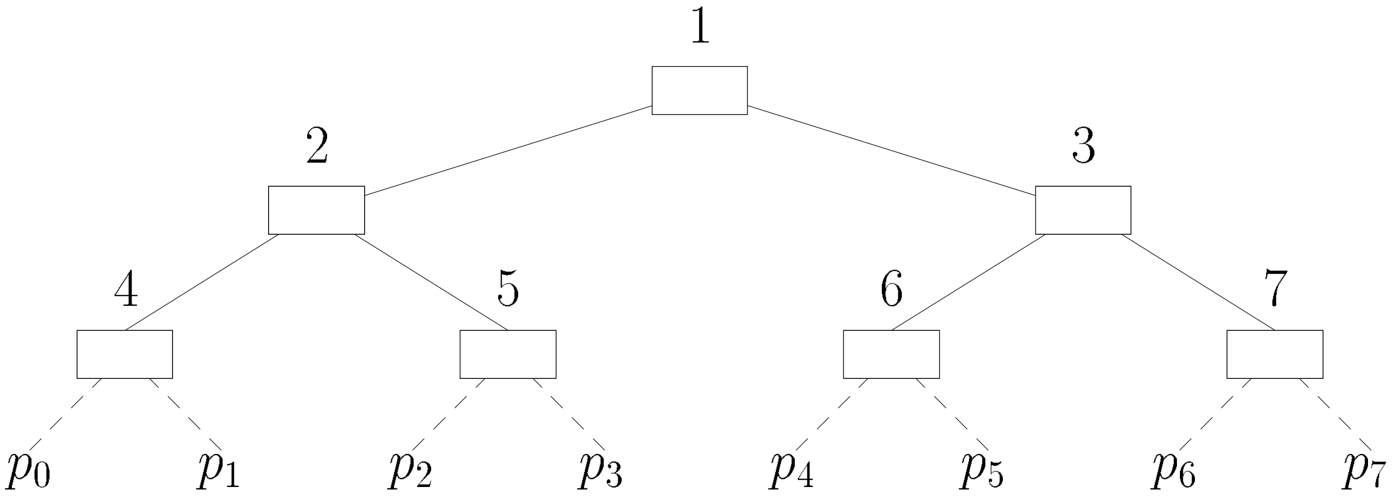


Figure 4.6

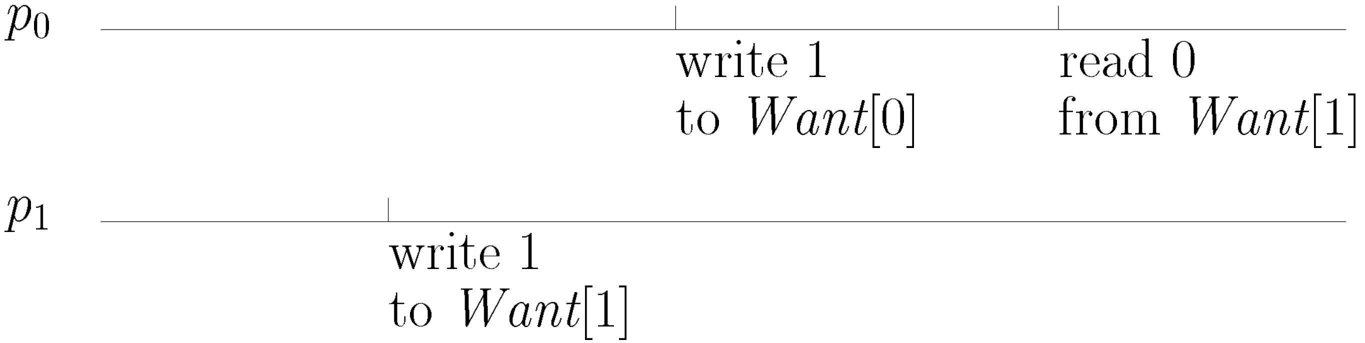


Figure 4.5

We do not show in detail how the real proof works (this can be found in the book). The idea is that we repeatedly create executions like C_2, until the same set of variables is repeated twice.

From these two executions, we can follow essentially the same argument as above (Figure 4.10).

such that
$p_0, \ldots, p_k$ cover $k+1$ different variables and
D is $\{p_{k+1}, \ldots, p_{n-1}\}$-quiescent, for any $0 \leq k < n$.

Clearly, this will give us the lower bound when taking $k = n-1$.

But first we prove:

Lemma 4.16. If C is p_i-quiescent, then there exists a p_i-only schedule which brings p_i into the critical section, and in which p_i writes to some variable that is not covered in C.

Proof. (Use Figure 4.7.) Consider a p_i-only schedule from C. By the no deadlock property, p_i must eventually enter the critical section (*draw* σ).

Assume, by way of contradiction, that p_i does not write to any variable not covered in C.

Then let the covering processors write to all variables (*draw* τ).

Now apply τ to C (*draw* Q). And then apply a p_j-only schedule to Q (*draw* π). By the no deadlock property, p_j must eventually enter the critical section.

Now apply π above (*draw*) and obtain a contradiction since p_i and p_j are critical (contradicting mutual exclusion). Q.E.D.

Proof of Lemma 4.17. By induction on k. For the basis, $k = 0$, consider the schedule guaranteed by Lemma 4.16 and 'truncate' it just before the first write of p_i to the additional register.

For the inductive step: Assume the lemma holds for k and prove it for $k+1$.

To understand what is going on, we first assume that each time we apply the lemma for k, $p_0, \ldots, p_{k-1}$ cover the same set of variables. (Regardless of the configuration we start from.) In general, this is not the case!

(Use Figure 4.8.)

We can apply the inductive hypothesis to get a configuration C_1 which is $\{p_0, \ldots, p_{k-1}\}$-quiescent and in which $p_0, \ldots, p_{k-1}$ cover $X_0, \ldots, X_{k-1}$. (*Draw* initial point.)

Now apply Lemma 4.16 to get P_k to cover another variable, X_k. (*Draw* α').

The problem is that P_k may have written in the process to variables $X_0, \ldots, X_{k-1}$. Therefore, the configuration is no longer $\{p_k, ..., p_{n-1}\}$-quiescent.

So, we let $p_0, \ldots, p_{k-1}$ write to $X_0, \ldots, X_{k-1}$. But then, they no longer cover them! So, we have to clean the situation (using the fact this is an 'on-going' problem).

Make $p_0, \ldots, p_{k-1}$ (in turn) go into the critical section and out of it. (*Draw* τ and D_1'.)

Ideally, we want now to apply the inductive hypothesis to D_1'. But this is not possible (since it is not necessarily quiescent).

However, apply τ to C_1 and get D_1 (*draw*), which is quiescent.

Then, we can apply the inductive hypothesis to get C_2 which is $\{p_k, ..., p_{n-1}\}$-quiescent and in which $p_0, \ldots, p_{k-1}$ cover $X_0, \ldots, X_{k-1}$ (this is where the simplifying assumption kicks in!). (*Draw.*)

Now, apply σ also to D_1', and get C_2' with the desired properties. (*Draw.*)

Proof. The interesting case is when both processors are in the entry section.

Clearly, the value of Priority does not change, without loss of generality assume it is 0.

But then p_0 waits in Line 6 with Want[0]=1, while p_1 waits in Line 2 with Want[1]=0.

Eventually, p_0 will see Want[1]=0 and will move into the critical section. Q.E.D.

To prove no lockout, assume p_0 is in the entry section.

By the no deadlock property, either p_0 goes into the critical section (and we are done) or p_1 goes into the critical section. When p_1 goes out, it sets Priority to be 0. At this point, p_0 will go in, since Want[0]=1 and then p_1 will not set Want[1].

4.3 n-Processor mutual exclusion (the tournament algorithm)

Only outline the idea. Use Figure 4.6.

Each processor starts at some leaf, and then moves to the next level to perform a 2-processor mutual exclusion algorithm (we use the above algorithm).

The critical section of internal (non-root) nodes is to go up to the next level. The critical section of the root is the real critical section.

The proof is by showing that an (appropriately defined) projection of the execution at each node is an execution of the 2-processor algorithm. Then we can derive the properties of the n-processor algorithm from the 2-processor algorithm.

More details are in the recitation.

4.4 n shared variables are needed

The next lower bound does not assume that:

- The size of registers is limited.
- Only a single processor writes to each register.
- The algorithm provides no lockout.

Note that if registers can be written by only a single processor, then the lower bound is almost obvious (need to announce your presence).

A good question (possible project?) is whether the assumption of no lockout simplifies the proof (or increases the bound—by a constant factor only).

First, we need some definitions.

Processor p_i *covers* a variable x in configuration C if the next step of p_i from C is to write to x (according to its state in this configuration).

Note that if p_i is in the same state in another configuration D, then it will also write to x.

A configuration is *quiescent* if all processors are in the remainder.

A configuration C is P*-quiescent* if there is a quiescent configuration D that looks similar to processors in P.

That is, as far as the processors in P 'know', C is quiescent.

The main lemma is:

Lemma 4.17. If C is quiescent, then there exists a configuration D, reachable from C with a $\{p_0, \ldots, p_k\}$-only schedule,

Lecture 4: Mutual exclusion using read/write operations

Main points today:

- Mutual exclusion can be solved with read/write operations, even with bounded values.
- But needs a lot of registers.

The bakery algorithm is not covered here, since it is typically covered in our Operating Systems course.

Outline:

1. 2-Processor mutual exclusion, with lockout.
2. 2-Processor mutual exclusion, without lockout.
3. The tournament algorithm.
4. The lower bound (n registers are necessary).

Sources: Section 4.4. This is also the reading assignment for this week.

Recitation material: The tournament algorithm (if needed, 2-processor algorithm).

4.1 2-Processor mutual exclusion (lockout possible)

Start with the asymmetric algorithm (Algorithm 4.4), to see how we get mutual exclusion.

Lemma . The algorithm satisfies mutual exclusion.

Proof. (Use Figure 4.5.) Suppose p_0 and p_1 are simultaneously Critical. Then, at this point (*draw*), Want[0] = Want[1] = 1.

One of them writes to Want last, say p_0 (*draw*). So the write of p_1 to Want[1] is before it (*draw*).

Then the read of p_0 from Want[1] follows its write (*draw*), and hence follows p_1's write to Want[1], and returns 1, a contradiction. Q.E.D.

This also guarantees no deadlock (will not prove that, it follows as part of next algorithm), but not no lockout.

4.2 2-Processor mutual exclusion (without lockout)

To get also no lockout, use the shared variable Priority. Present Algorithm 4.5.

Mutual exclusion follows along the same lines as the previous algorithm, see proof in the book.

Lemma . The algorithm satisfies no deadlock.

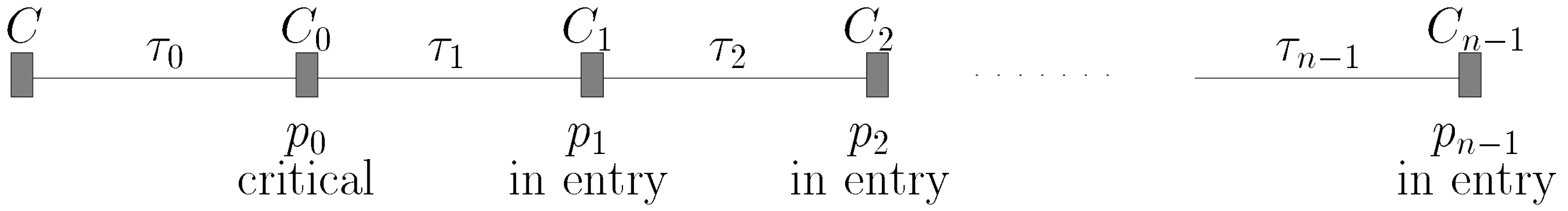

C
τ_0
C_0
τ_1
C_1
τ_2
C_2
τ_{n-1}
C_{n-1}
p_0
critical
p_1
in entry
p_2
in entry
p_{n-1}
in entry

Figure 4.4

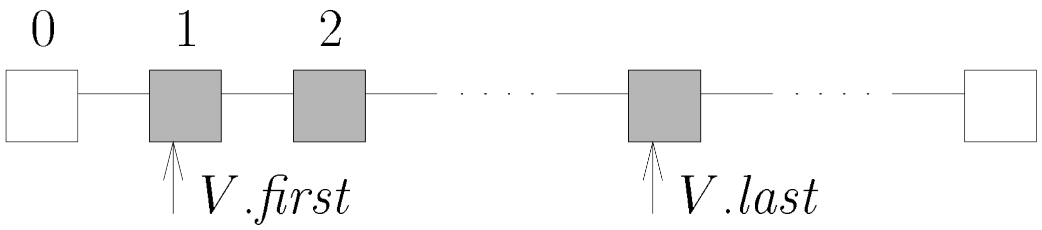


Figure 4.2

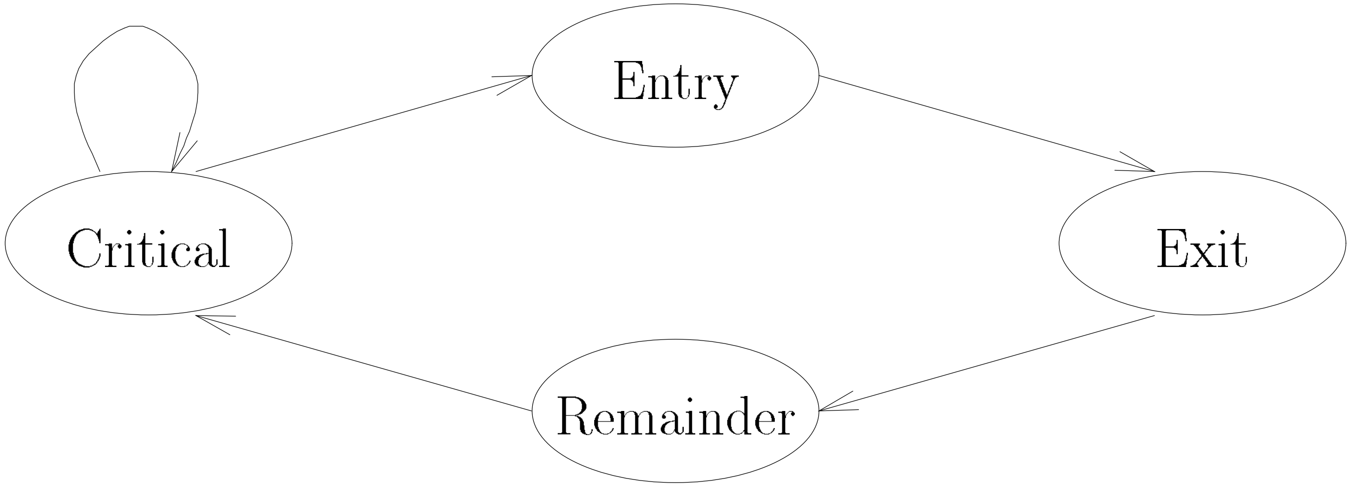


Figure 4.1

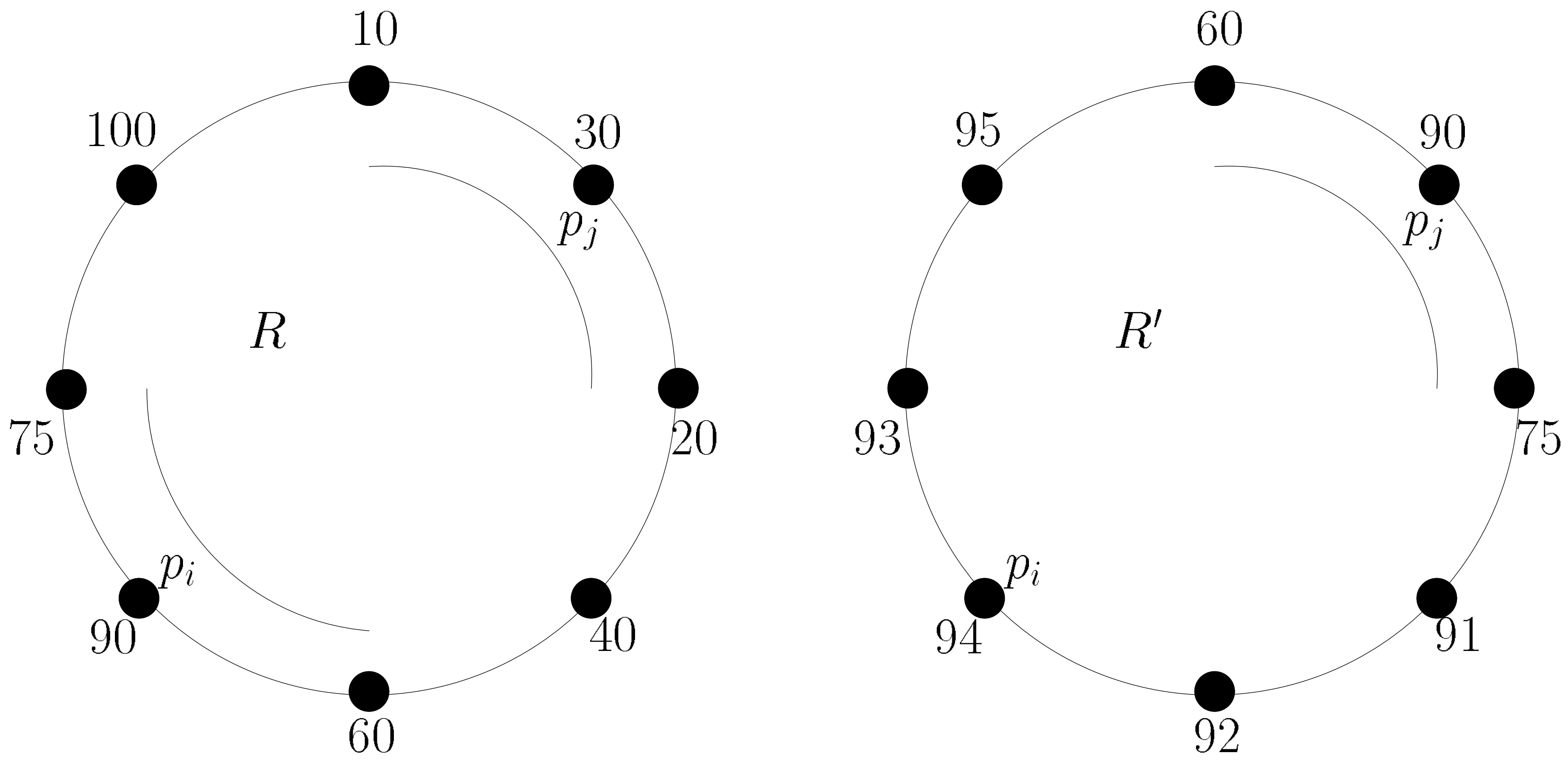


Figure 3.7

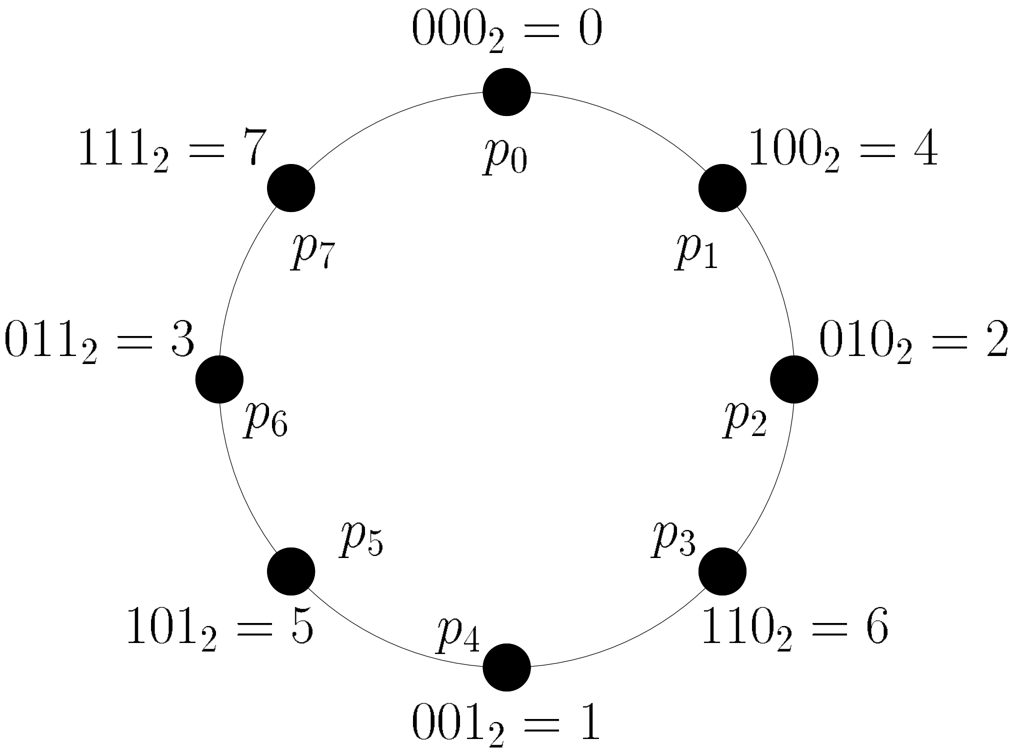

$000_2 = 0$
p_0
$100_2 = 4$
p_1
$010_2 = 2$
p_2
$110_2 = 6$
p_3
$001_2 = 1$
p_4
$101_2 = 5$
p_5
$011_2 = 3$
p_6
$111_2 = 7$
p_7

Figure 3.8

is Critical. In C_j, $p_1, \ldots, p_i$, ..., p_j is in Entry and p_0 is Critical.

From C_i consider a run in which p_0 leaves the critical section, and then $p_1, \ldots, p_i$ run alone forever.

Some processor p enters the critical section at least $k+1$ times (within finite schedule).

Then apply the same schedule to C_j, and p_j is overtaken $k+1$ times.

Exit: Clean up.

Remainder: Everything else.

We need to design the Entry and Exit sections, that should work with almost any Critical and Remainder sections (as black boxes), only assuming:

- No processor stays in Critical section forever.
- Variables used in Entry/Exit not used elsewhere.

Must guarantee the following properties.

Mutual exclusion: No two processors together in Critical.

No deadlock: If processors are in Entry then eventually *one* is in Critical.

No lockout: If a processor is in Entry then eventually *it* is in Critical.

Last two properties should hold only in admissible executions (where each processor takes an infinite number of steps). Also, *unobstructed exit* property guarantee that a process leaves the exit section.

Here, give an overview of results.

3.4 Solutions using Test&Set, Read-Modify-Write

```
Test&Set($v$): atomically
    temp := $v$
    $v$ := 1
    return temp
Reset($v$):
    $v$ := 0
```

The following simple algorithm works in this case:

```
Entry:
    wait until Test&Set(L) = 1
Exit
    Reset(L)
```

Clearly, it guarantees mutual exclusion and no deadlock. But what about no lockout? Use a more powerful primitive.

```
Read-Modify-Write($v$,func): atomically
    temp := $v$
    $v$ := func(temp)
    return temp
```

Use a shared circular queue (use Figure 4.2).

Each processor gets a number between 0 and $n-1$. This number indicates when you are supposed to become Critical. A component indicates which number is now in the critical section, and is incremented when a processor exits the critical section.

The clearly guarantees mutual exclusion, no lockout and clearly, also bounded waiting (really, FIFO).

Memory: One variable, with $O(\log n)$ bits.

3.5 Lower bound on the memory size

If we want to guarantee k-Bounded waiting (a processor is not overtaken more than k times), then we need variables of size $\Omega(\log n)$. (Use Figure 4.4 and Theorem 4.3.)

Get two configurations C_i and C_j which have the same memory state ($i < j$). In C_i, $p_1, \ldots, p_i$ are in Entry and p_0

Proof. To prove this, we need to assume identifiers are *spaced* away from each other (at least n identifiers between them).

The problem is that p_i and p_j are not matching in their own ring. So we need to work through an additional ring.

Consider Figure 3.7.

Given two processors p_i and p_j in ring R, consider another ring R' with a processor p'_j, such that:

- R' is order-equivalent to R, and
- p'_j in R' is matching to p_i in R, and
- p'_j in R' has the same neighborhood as p_j in R.

(To do this, just place p_i's neighborhood and add identifiers around it to get the required order pattern.)

We argue (by induction on k) that p_i's behavior in R is equal to p'_j's behavior in R' (during the first k active rounds). This is the somewhat difficult argument!

Then since the algorithm is comparison-based, p'_j's behavior in R' is similar to p_j's behavior in R (by definition).

It follows that p_i's behavior in R is similar to p_j's behavior in R. Q.E.D.

Before round $n/8$, to any processor there is another processor with order-equivalent k-neighborhood. So, if the first processor halts as leader, the other one halts as leader too. This proves:

Lemma . In the symmetric ring there are at least $n/8$ active rounds.

Since there are many processors with order-equivalent neighborhoods:

Lemma . In every active round, at least $n/4k$ messages are sent.

Wrap-up with the arithmetics...

Final remarks: In the book, can see how the time-bounded case can be reduced to this case (using Ramsey's theorem, it's really neat!).

3.2 The Shared Memory Model

Similar to message passing, but:

- Only asynchronous.
- Configurations include states of processors and values of shared registers.
- Each step changes state of a processor and of a single shared variable.
- Assume each processor takes infinite number of steps.

3.3 The Mutual Exclusion Problem

Partition the code into four parts (use Figure 4.1).

Critical: The actual code that should be protected.

Entry/Trying: Where processor coordinate who would enter the critical section.

Lecture 3: Mutual exclusion (using strong operations)

Main points today:

- Limitations of synchrony (complete from last time).
- Introduce the shared memory model.
- Various atomic operations on shared variables, and their implication on the complexity of solving a particular problem.

Outline:

1. Complete the lower bound of $\Omega(n \log n)$ messages for restricted synchronous algorithms.
2. Define the shared memory model.
3. Define the mutual exclusion problem.
4. Solutions using Test&Set, Read-Modify-Write.
5. Lower bound on the memory size.

Sources: Sections 3.4, 4.1, 4.2 and 4.3. This is also the reading assignment for this week.

Recitation material:

- Repeat the lower bound for synchronous rings.

3.1 An $\Omega(n \log n)$ lower bound for restricted synchronous algorithms (complete)

We have already seen the general structure of this lower bound. Repeat the important definitions:

- Two id sequences are *order-equivalent* if they have the same order pattern.
- Rings are *order-equivalent* if the have the same order pattern when indices are taken relative to the minimum identifier.
- Two processors are *matching* if they are at the same position in the id sequence.
- An algorithm is *comparison-based* if matching processors, in order equivalent rings have similar behavior, when the behavior is whether a processor sends messages or not and whether it halts or not.
- k-neighborhood.
- *active* round.

Using Figure 3.8 as an example, we argue:

Lemma 1. There exists a highly symmetric ring: at least $n/4k$ processors with order-equivalent k-neighborhoods, for any $k < n/8$.

The next lemma is the main thing to emphasize in the proof.

Lemma . If the k-neighborhoods of two processors (in the same ring) are order equivalent then they have the same behavior through the k-th *active* round.

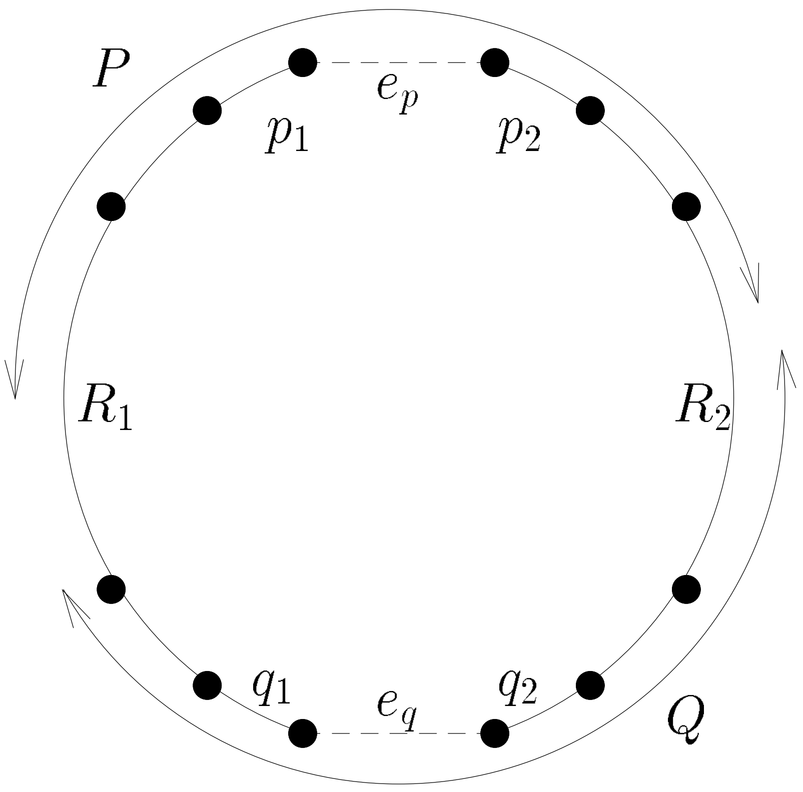


Figure 3.6

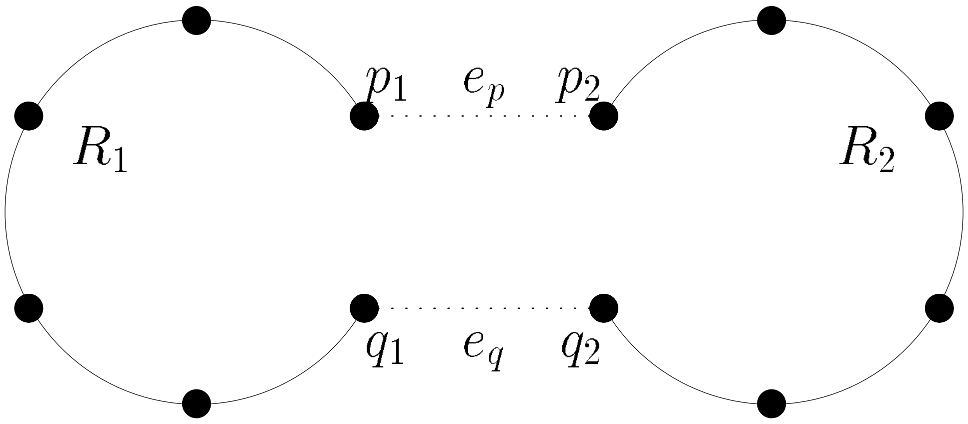


Figure 3.5

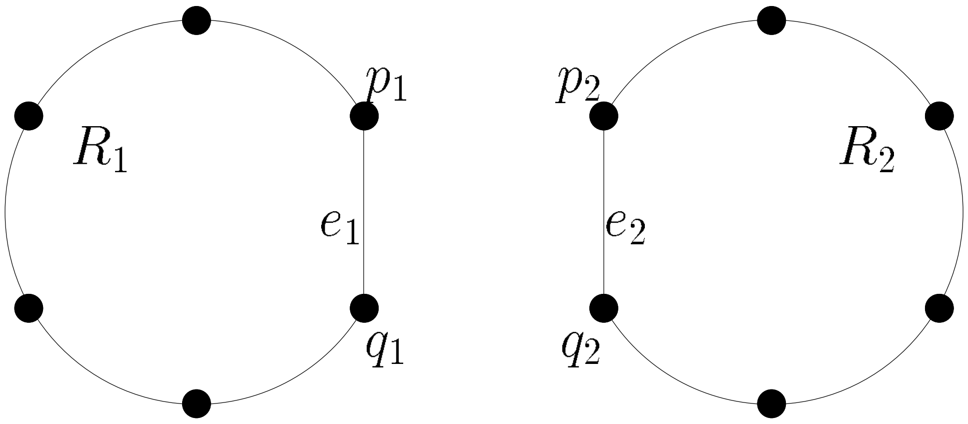


Figure 3.4

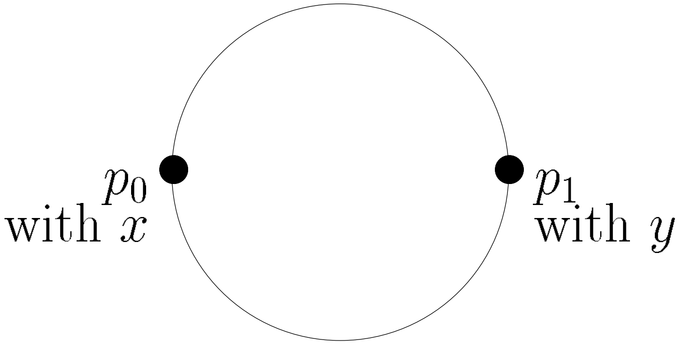


Figure 3.3

This covers Lemma 3.16 and 3.17, and will be proved in the next lecture.

- After $n * 2^x$ rounds p_i gets its message back, at this point no other message is being forwarded.
- How many times can a message of some other processor p_j, with identifier y, be forwarded?

  $$\leq (n * 2^x)/2^y = n * 2^{x-y}.$$

  Remember that $y < x$.
- So what is the total?

  $$n * \sum_{j \neq i} 2^{x - id_j}$$

Since id_j are different, the maximum is achieved with $x - id_j$ being $1, 2, \ldots$, which yields a geometric sum.

In addition, p_i's message is forwarded n times. This yields a total of $O(n)$ messages.

Point to the book to see how to remove the assumption that processors start together. The idea is that messages travel in a fast rate to wake-up processors, and then switch to the variable rate described above. Correctness remains the same, but message complexity analysis is slightly more complicated.

This is a strange algorithm:

- Uses identifiers in non-standard manner (to determine rate).
- Takes a (very!) long time (number of rounds).

We next see that if identifiers are used only for comparisons, then $\Omega(n \log n)$ is a lower bound on the number of messages.

2.4 An $\Omega(n \log n)$ lower bound for restricted synchronous algorithms (start)

The restriction is that algorithm use identifiers only for comparisons.

Specifically, consider a ring, and list all identifiers in clockwise order, starting from the minimum identifier.

Two rings are *order-equivalent* if they have the same order pattern (if $x_i < x_j$ in one ring then $y_i < y_j$ in the second ring, when indexes are taken relative to the minimum identifier).

Two processors are *matching* if they are at the same distance from the minimum identifier.

An algorithm is *comparison-based* if matching processors, in order equivalent rings have similar behavior, when the behavior is whether a processor sends messages or not (for simplicity assume if send, then send to both directions) and whether it halts or not.

Define *k-neighborhood.*

Assume we have a ring which is highly symmetric and has lots of processors with order equivalent neighborhoods (more specific requirements appear below).

A round is *active* (in a ring R) if a message is sent somewhere in the ring. Note that active rounds are the same in order-equivalent rings.

The idea is that only in active rounds processors acquire information. That is:

Lemma . If the k-neighborhoods of two processors (in the same ring) are order equivalent then they have the same behavior through the k-th *active* round.

Otherwise, patch them (Figure 3.5), and 'release' the two edges between them and let messages flow between them; since the max is in one side, a message must arrive to each processor on the other side, yielding $n/2$ additional messages.

The problem is that if we release both edges, we won't be able to 'patch' the ring in the next level of the induction. We need to keep the execution *open*, i.e., have at least one edge on which no message is delivered (in either direction).

We should be more careful and first do an 'experiment'. *(Experiments in theory?)*

We release both edges, and look at the propagation of messages (from the edges onwards). Note that messages form two consecutive chains centered at the two edges. We make sure to stop just before the two chains 'meet' at a processor, and hence the chains are disjoint. A message must reach each processor in R_2 and hence at least one of the chains is at least $n/4$ long. Let's say this is the chain centered in e_1. (Use Figure 3.6.)

Now we construct a different execution in which we only release e_1; since the chains were disjoint, the processors take the same steps (and send the same messages) as in the experiment execution.

We get $n/4$ additional messages, and hence:

$$M(n) \geq 2M(n/2) + n/4$$

$$\Longrightarrow M(n) = \Omega(n \log n)$$

2.2 The $O(n)$ non-uniform synchronous algorithm

This and the next item cover Section 3.4.1.

Simply, partition into phases, each with n rounds. In the k-th phase discover if the identifier k is in the ring.

To discover if k is in the ring:

1. If you have k (and not halted before) then send a message (possibly without any content).
2. Forward the first message you get, unless you have sent a message.
3. After n rounds, if no message then continue to next phase, otherwise, halt.

Complexity is obvious.

2.3 The $O(n)$ uniform synchronous algorithm

Also assumes that processors start together, which is part of our formal model, anyway.

The idea is that a message with small identifier x travels faster than a message with large identifier y. This way, the message x will catch up with the message y before it has been forwarded many times.

As always, processors 'swallow' (and do not forward) messages with identifiers larger than the minimal identifier seen so far (including of course their own).

Specifically, a message with identifier x travels at 'rate' 2^x; that is, it is delayed for $2^x - 1$ rounds at each processor it arrives to. (See Algorithm 3.2.)

Clearly, only the processor with minimum identifier gets its message back; so, we have only a single leader.

Message complexity:

- Assume p_i has the minimum identifier, x.

Lecture 2: Leader election in rings

Main points today:

- There are inherent bounds on message complexity of leader election algorithms.
- Synchrony can help.
- But only to a limited degree!

This is also the first presentation of how lower bounds on distributed algorithms are proved; use of similarity and symmetry arguments.

Outline:

1. Lower bound of $\Omega(n \log n)$ message on asynchronous algorithms.
2. A synchronous algorithm with $O(n)$ messages (uniform, synchronous start).
3. A synchronous algorithm with $O(n)$ messages (non-uniform, synchronous start).
4. Lower bound of $\Omega(n \log n)$ messages on restricted synchronous algorithms.

Sources: Sections 3.3.3 and 3.4. This is also the reading assignment for this week.

Recitation material:

- Why need to assume processors have id's (Section 3.2).
- A randomized leader election algorithm for anonymous rings, with analysis (Section 14.1).

2.1 The $\Omega(n \log n)$ lower bound

This covers Section 3.3.3.

Think only about the restricted problem of getting all processors to know the minimum (an exercise asks you to show a reduction from the general problem to this restricted problem). The algorithm should be uniform (work regardless of ring size).

The proof is by an inductive construction, assuming n is a power of 2, showing an execution of n processors in which $M(n)$ messages are sent (for some function M with appropriate growth).

The base case is when $n = 2$ (use Figure 3.3).

Then consider two processors, p_0 (with identifier x) and p_1 (with identifier y), and assume $x < y$. Clearly, p_1 should get a message from p_0 with x inside it. Then consider the execution in which this message is sent. For this execution, $M(2) = 1$.

For the inductive step: Consider two sub-rings R_1 and R_2, and the edges between them, e_1 and e_2; assume the minimum is in R_2. (Use Figure 3.4)

First cause many messages, $M(n/2)$, be sent in each 'half' of the ring, without any communication between the two halves (using the induction hypothesis). Let each half quiesce and see how many messages are sent; if already get $n/4$ additional messages then we are done.

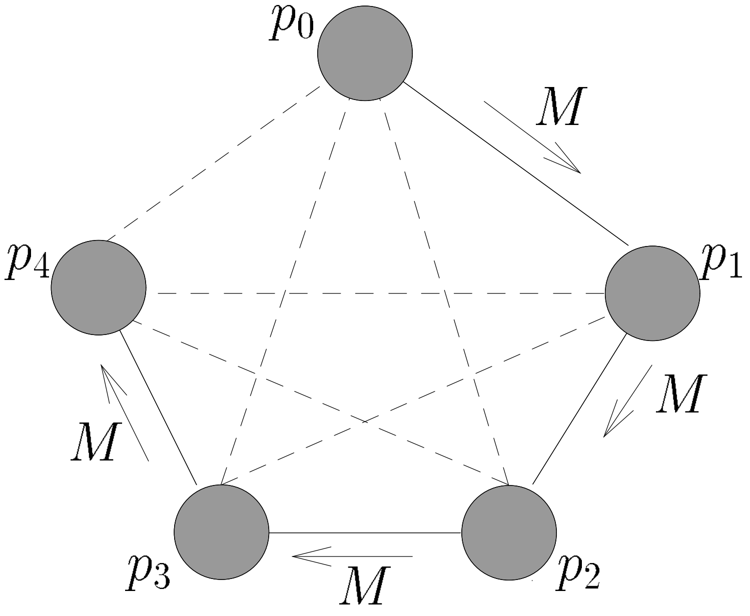


Figure 2.6

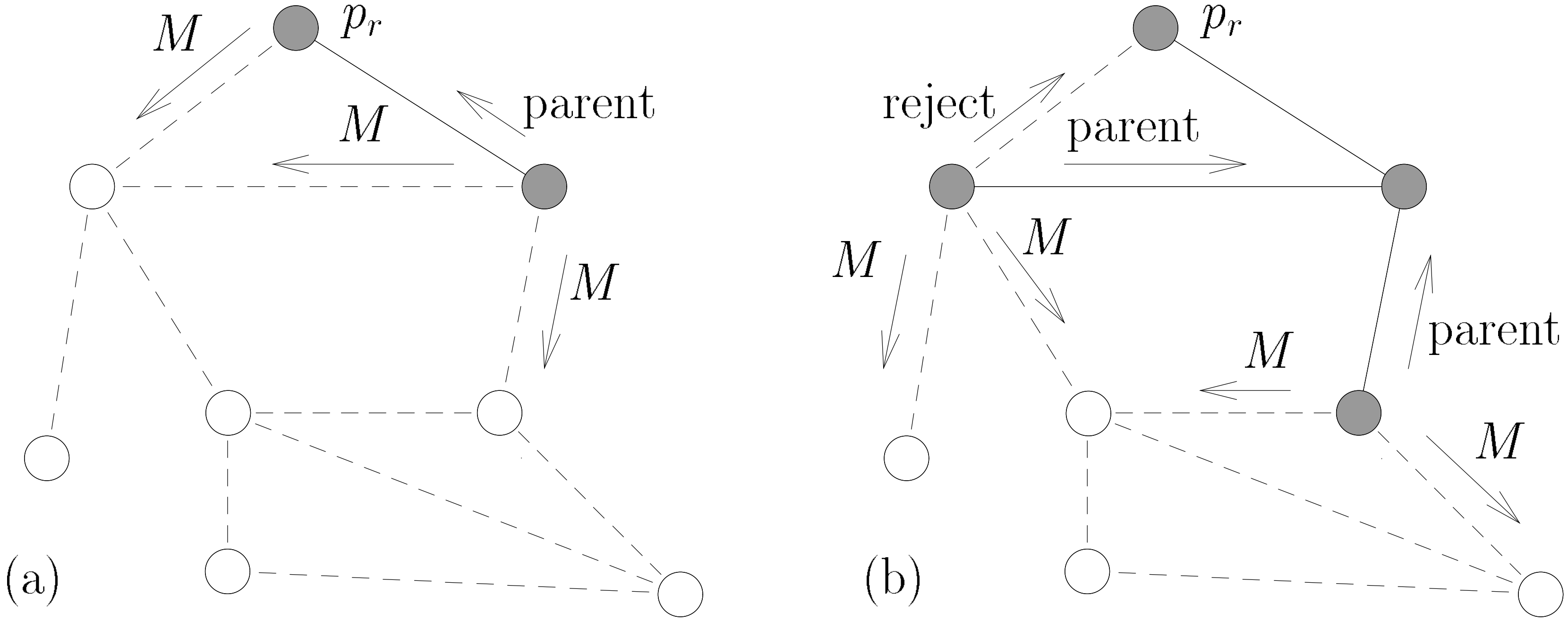


Figure 2.5

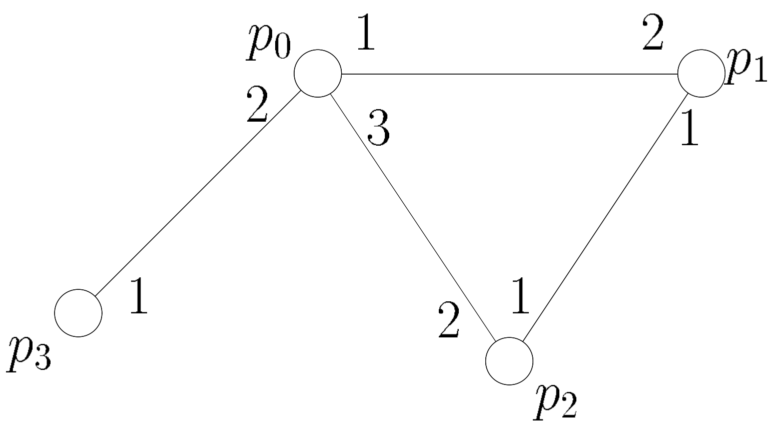


Figure 2.1

1.4 What if there's no root? (Section 2.5)

High-level explanation of how to conduct several such explorations concurrently (Algorithm 2.4):

- Each processor starts its own 'DFS exploration' computation, carrying its identifier.
- When getting a message with an identifier larger than the current root's identifier: Join the DFS exploration of this root. Otherwise, do not respond (thus stalling this DFS exploration).

Complexity: messages–n times DFS $= O(n * m)$; $O(m)$ time.

Pretty bad; we'll see what to do in a particular network, rings.

But first, let's re-do the algorithm in a ring (Section 3.3.1).

1.5 Improving the message complexity in a ring

This covers Section 3.3.2.

Elects a leader (to be the root), by a simplified version of Algorithm 3.1 for known ring size $n = 2^r$.

In each asynchronous 'phase' k, try to be the leader for a neighborhood with 2^k processors around you.

If in phase k, you still have a chance to become a leader, you are a *temporary leader* in phase k. if you are a temporary leader in phase k then:

- Send a message with your identifier to distance 2^k in both directions and wait for acknowledgment.
- Acknowledgment is returned only if no other processor with higher id in the 2^k-neighborhood.
- If received acknowledgment, be a temporary leader in phase k+1.
- If $k = r$ ($2^k = n$), terminate.

Correctness and analysis:

- 2^k-neighborhoods of temporary leaders in phase k cannot overlap

  $\Longrightarrow O(n)$ messages in each phase (total)
- Hence, no more than $n/2^k$ temporary leaders in phases $k + 1$

  $\Longrightarrow$ a single leader in phase $\log n$
- Hence, at most $\log n$ phases

  $\Longrightarrow O(n \log n)$ messages total.

We have improved from $O(n^2)$ messages to $O(n \log n)$ messages. Perhaps we can do even better?

But what if the tree is unknown? Can use a simple modification of the same algorithm to construct a tree from a given root (Algorithm 2.2):

- The root sends to its neighbors.
- A node receiving a message for the first time, marks the sender as parent and sends a 'parent' message. Send the message to all its other neighbors.
- When receiving a message (not for the first time), send a 'reject' message.
- Wait to hear answers from all neighbors; create a list of children (with those who sent 'parent'); terminate.

(Figure 2.5 shows an example execution.)

If processors work in lock-step, then this is a BFS tree; otherwise, it is **not** (Figure 2.6).

1.2 Formal model (asynchronous and asynchronous)

This covers Section 2.1.

Processors are state machines; events are either message delivery events or computation events.

Executions are alternating sequences of configurations and events.

Synchronous model: Processors take steps in rounds.

Complexity measures (worst-case):

- Number of messages sent since the algorithm starts until it terminates.
- Time is the number of rounds.

Asynchronous model: No real restrictions.

Message complexity is defined essentially as for the synchronous model.

Asynchronous time is measured using an arbitrary assignment of times to events under the assumption that messages are delivered within 1 time unit.

1.3 Analysis of flooding and DFS

This covers Section 2.4.

Re-do the previous algorithm and analyze it in both models.

Show modifications to collect back information (convergecast).

Show the modification to do DFS exploration (Algorithm 2.3).

- The root sends to one of its neighbors.
- A node receiving a message for the first time, marks the sender as its parent and sends the message to one of its neighbors.
- When receiving a message (not for the first time), send a 'reject' message.
- When receiving a response (either reject or 'parent') from a neighbor, try another neighbor.
- When all neighbors were explored, send 'parent' message to the parent processor.

For both models: Message complexity is $O(m)$, and time complexity can be as bad as $O(m)$.

Lecture 1: Introduction and basic message passing algorithms

Main points today:

- What are distributed systems?
- How to model them and what are the complexity measures.
- How to state algorithms and analyze them.
- Optimizing the message complexity.

This first lecture already introduces the emphasis of the course on formal modeling and analysis of complexity measures, and the important role of lower bounds. It also starts to indicate differences between the synchronous and asynchronous models.

Outline:

1. Description of message-passing systems and a simple algorithm for flooding on a tree.
2. Synchronous and asynchronous systems and their modeling; definition of message and time complexity.
3. Analysis of flooding algorithm (given a root) in both models; and some of its variations (DFS).
4. Finding a root; complexity analysis. Yields $O(n^2)$-messages algorithm for ring networks.
5. Improving the message complexity to $O(n \log n)$.

Sources: Sections 2.1-2.5, 3.1. This is also the reading assignment for this week.

Recitation material: A precise description of the $O(n \log n)$ messages leader election algorithm for rings.

1.1 What are distributed systems and a simple algorithm

This covers Sections 2.2 and 2.3.

Loosely coupled: A collection of computers (processors), typically with unrestricted computation power.

Communicate by passing messages (or sharing memory, but this will be described later in the course).

Processors are located at the nodes of a communication network with a given topology; processors on adjacent nodes can send a message directly to each other.

Want to broadcast and collect information, do some joint calculations, and coordinate actions.

To broadcast a message M from a node, assuming there is a tree rooted at the node (Algorithm 2.1):

- The root sends to its children and 'terminates'.
- A node receiving a message, sends a message to its children (if it has any), and 'terminates'.

(Figure 2.1 shows an example execution.)

How many messages are sent by this algorithm? $O(n)$.

How long it will take to for all nodes (processors) to get the message? $O(d)$ time.

Lecture notes on

Distributed Algorithms

Lecturer: Hagit Attiya
Teaching assistant: Arie (Leonid) Fouren

Winter 1997-8, The Technion

Supplement to
Distributed Computing:
Fundamentals, Simulations and Advanced Topics
by Attiya and Welch

Introduction

This semester-long course meets once a week for a two-hours lecture; in addition, there is a weekly one-hour recitation. The notes contain 13 lectures; the last lecture included a presentation of the projects written by the students (see below). The class had approximately 30 students, approximately half of them seniors, and the rest graduate students.

The course followed a fairly aggressive schedule, and assumes that students read details in the book (after class). The description of the material in class is fairly informal, at most times; however, their written assignments were supposed to be quite formal.

In some cases, the recitations covered material not in the book. In these cases, we point to the paper(s) on which the recitation was based.

Evaluation of the students was (approximately) equally based on four homework assignments handed out during the semester, and on a written project.

Additional material can be found through the book homepage:

http://www.cs.technion.ac.il/~hagit/DC/

Also, look at the course's homepage, for project ideas and guidelines:

http://www.cs.technion.ac.il/~hagit/DA98/