Introduction; Mutual Exclusion in Shared-Memory Systems

Bring:
30 copies of:
H1, H2, H3
Lamport paper
Friedberg, Peterson paper

1 Introduction

Work from H01.

6.895: Advanced Distributed Algorithms
Nancy Lynch
room, etc.

Prerequisites

1.1 Course overview

Explore recent theory literature on distributed algs.
Emphasis: algorithms for dynamic settings (Internet, mobile)
Processes, channels fail, recover.
Timing variations
Join, leave
Algorithms must adapt.

Traditional distributed algorithms, e.g., my book: mostly static.
E.g., known participants.

Want to extend the theory to new dynamic settings.
New algorithms, new lower bounds.
New models, measures.

Papers I've chosen are the best I could find.
More remains to be done.
We'll consider their ideas, and how they can contribute to the new theory, how to extend.

Emphasize:
Communication (multicast, gossiping, group comm.)

Data management (consistency guarantees, replicated data managment, quorums,...)
Also: failure detection, consensus, resource allocation (mutex is special case), performing work, clock synch, namespace mgmt.

I see the problems as *services* to be implemented in a dynamic distributed setting.
That's how such systems are being built in practice—layered services.
Modularity.
Only, the services aren't so well specified in practice.
I'm interested in what kind of theory should go along with this.
Regard the services as problems, consider algorithms, impossibility results.

Theory course:
Focus on algorithms with precise claims (correctness + performance + f-t).
Look for impossibility results.
Some less theoretical papers, especially is the second half of the course.
We can discuss what theoretical work might be done.

1.2 Course conduct:

Long list of readings (H02).
Shared memory,...mobile.
Known and unknown participants.

Topic each day.
Required reading, other background reading.
Everyone should read the "required" papers.

Someone leads discussion.
Me sometimes, students sometimes.
Maybe some other experts.

Idea is not to cover every detail of the papers.
Pick out what is interesting.
Try to answer questions:

1. What is the technical contribution of the paper? What are the key algorithmic ideas? Why does the algorithm work? What is its performance? How does the impossibility argument go?
2. What is the significance of this work? What models/assumptions does the algorithm or impossibility result apply to? Is this significant in practice? What does this have to do with dynamic distributed settings?
3. What new work (relevant to dynamic distributed computing settings) does this suggest? What would make the results more practical?

Everyone should participate actively in discussions.

1.3 Administrative details

Grading:
No exams, no problem sets

Grades based on:
Reading the papers, participating in discussions, having useful things to say.
Collaborating, helping other people to understand the material.
Leading the discussion.
Producing notes.

Grades mainly based on effort, participation, helping others to learn

Collaboration strongly encouraged
Best way to learn.

1.4 Course Content

From H02, go over list of topics.
From H03, look at approximate amounts of time to spend on each.
Flexible. We can expand some and drop some, depending on what looks most interesting.

Scary list.
Roughly divided based on IPC method, from tight to loosest coupling.
The strongest results should hold in the tighter-coupled cases.

Section 1: Shared memory.

The most tightly-coupled.
Here, I started with the new papers by Merritt and Taubenfeld as my goal.
Those provide results for infinite set of participants, unknown actual participants.
Clearly any theory for dynamic systems would have to cope with this kind of undertainty.

But reading the papers was hard, because the results build on results in other papers.
So I collected some of the most important prerequisite papers.
I'll present these first.

Section 1.5 is a bit different—deals with shared objects, not shared memory.
Here the objects themselves may be faulty.

Section 2: Client-server seems worth distinguishing as a separate model.
Client processes, lighter weight, more uncertainty.
Servers, responsible for more, heavier weight, maybe assumed to be more reliable.
I found three kinds of interesting work here: on Byzantine quorum systems, on consensus (Disk Paxos), and on data replication.

Worth spending time on these.
Especially: the new Chockler, Malkhi paper looks very interesting because it deals explicitly with

infinitely many potential participants.

Section 3, networks with known participants.
Somewhat tightly-coupled network models, maybe good for LANs.
E.g., the algorithms do things like:
Keep track of all the participants.
Detect who has failed.
Send messages to all and wait for responses.
Etc.

I listed lots of papers on logical time and clock synch, but don't want to spend too much time on this—pretty standard background.

Failure detectors seem interesting.
Some basic papers on FDs, some papers that say what you can do with them.
Then some newer papers that attempt to bring in more practical concerns, like performance issues.

Fault-tolerant broadcast also pretty basic stuff by now.

Consensus has a lot of ongoing work, including Paxos in networks, early-stopping consensus results, leader-based consensus.
Worth a new look.
Consensus can be a major building block for distributed algorithms, including dynamic algorithms (we used it in two recent papers, on dynamic atomic bcast, and on dynamic impls of atomic objects).
So it's certainly a good idea to review the major new developments.

Data replication is pretty standard.

Performing work—Alex Shvartsman will talk.
Given a set of tasks that everyone knows, farm them out so that they all get done, as quickly as possible and with as little overhead as possible.
Have been many papers, covering many different models.
But not much yet on highly dynamic models.
He'll try to overview the existing work and suggest new problems.

Self-stabilization: Assume that the system starts in an absolutely arbitrary state, and you want it to recover within a reasonable amount of time and resume doing whatever it's supposed to do.
We'll start with Dijkstra's paper and then try to summarize the (very large) field. Probably spend a week.

Section 4: Now we get to network models that are more suited for WANs. Well, actually the division isn't clean—some of the earlier work, e.g., some on self-stabilization, applies to WANs.

Some of this is basic: A couple of neat algorithms for routing and maintaining good structures for communication (sparse partitions, MSTs).

Then I can overview the work on group communication. This defines separable services that isolate handling of changes—joins, leaves, and fails—in a dynamic network, allowing the building of applications on top that, for the most part, ignore the changes.
An interesting and useful approach, but it has its costs.
Actually, these services are implemented best in LANs, but I put it here because the application processes are unaware of who the participants are. ???

The work on bcast communication is a new DISC paper that shows how to implement a consistent atomic bcast efficiently in a very dynamic setting.

Location services are for peer-to-peer networks.
This area has exploded recently, with its own workshops.
Worth a look to see what the interesting theory is.

Under data management I have a queue-management protocol and our new RAMBO atomic data-management protocol.

The last part is on mobile computing.
Not much good theory here yet.
Most papers seem to deal with basic communication issues.
It would be nice to get a summary here.
There's just a start at theoretical work for interesting higher-level problems to be solved in mobile networks.
I picked out a few of these papers: on mutual exclusion, leader-election, etc.
And a bit of new work on robots.

1.5 Mechanics

Need volunteer:
Webmaster, help fix up web site, maintain a bit, set up email list.
Students send email to...to get on the mailing list.

First assignment:
Students should choose topics to talk about, by next meeting.
I plan to cover section 1 myself, will take about 3 classes.
Then I'd like students to cover some of the other topics, in particular:

Section 2:
Byz. quorum systems
Data replication
Consensus: Can work with me

Section 3:
Logical time and clock synch: Work with me
Failure detectors: Work with me
Fault-tolerant bcasts: Work with me
Consensus: Work with me
Data replication and consistency

Performing work
Self-stabilization

Section 4:
Routing
Communication structures
Location services

Section 5: All topics

In any case, I'll try to work closely with students covering topics.
Want to really understand the ideas and get them across, will take a lot of reading/thinking/preparation.

Look at the schedule to get an idea of what one day might cover, and pick roughly based on a 1-day unit. Two students might cover a two-day unit, working together, e.g., self-stabilization.

Also write more generally what you're interested in (topics, specific papers), because the schedule is flexible.

If there's lots of interest in a topic or paper, we might devote more time to it and drop something else.

Give some alternatives.

Due next time.

Need a volunteer scribe.

2 Today's actual material

Readings:
Lamport. Fast Mutual Exclusion
Friedberg, Peterson. An Efficient Solution to the Mutual Exclusion
Problem Using Weak Semaphores.

These seem to me to be small and standard enough that I can just talk about them today, without your having to read them first.

Next time:
Read Moir-Anderson, Sections 1-4.
Attiya, Fouren, "Adaptive and Efficient Algorithms for Lattice Agreement and Renaming", from SIAM JC 2001.

Optional: I will probably rely on two other papers, one by A-F on long-lived (2k-1) renaming, and one by AST on Long-lived and adaptive collect.
But these looks sort of complicated, so I won't go over all the details.

Assuming you just have a reasonable amount of time, read the first two papers.

All of these papers are intended to be helpful in understanding the newest papers.

3 Lamport's paper on fast mutual exclusion

3.1 Overview

This short paper presents an algorithm for deadlock-free mutual exclusion in a shared-memory setting, with a fixed number n of participants.

Features:

1. Uses only read/write variables, which are considered to be a weak primitive.
   The r/w variables are assumed to be able to hold a process id, e.g., an integer in the range 1,...n.
2. Has a cute structure, which has reappeared in many other papers.
3. Performs in constant time when no other processes are around.
   This is optimizing for the "normal case", which is a good algorithm design principle.
   Different from the algorithms in my book, e.g., Dijkstra's, Burns,...where even in the normal case, you have to look at variables belonging to all the other processes. Time O(n).

Drawbacks:

1. It does not satisfy lockout-freedom. Lamport maintains this isn't important in practice, because contention is rare.
2. It still requires $\Theta(n)$ shared variables, at least if we want this to work in the asynchronous setting.

He includes a timing-based variant of the algorithm that uses only a constant number of shared variables, independent of n.
But that only works under certain assumptions about lengths of delays vs. time for processes to take steps.

Latency defaults to time $\Theta(n)$ when there's any contention.
Ideally, one might like a "graceful degradation" property for algorithms, in which the latency depends on the number of contenders.
That issue is addressed in the several papers we'll cover, on "adaptive" algorithms.
But here, it's just optimal vs. everything else.

Though this is about shared memory, since it's a weak form of memory (read/write), we can think about its possible use in networks.

3.2 Technical results

3.2.1 Fischer's algorithm

Lamport begins by talking about a trivial timing-dependent algorithm due to Mike Fischer.
It uses only one shared variable.
However, even in the non-contending case, this incurs a delay which is assumed to be large (compared to step times).
Lamport would like to avoid such a delay.

(p. 2 code)

How this works:
Repeatedly wait for $x = 0$, set x to your own index i, wait a while and check that it's still your own index.
If not, start over.
If so, go to cs.

Assumes that the delay is larger than time between successive steps of any other process.

Why this gives m.e. (intuition):
Suppose i does the set, then check, then goes to CS.
The delay rules out a situation where j reads x before the assignment $x := i$, seeing it as 0, then after i does its recheck (seeing $x = i$), j does its assignment $x := j$.
That would let both go to CS.

Lamport makes a case that the delay has to be fairly large to be sure it's bigger than everyone's step times.
So he rules this out and tries to improve Fischer's algorithm.

3.2.2 Another timing-dependent algorithm

(Code p. 5)

The first attempt is still timing-dependent, but it moves the delay so that it isn't on the no-contention path.

Again, the algorithm uses the strategy of setting x to your own index i, then checking that x is unchanged.
If it is, go right on to the CS, with no delay on this path.

Now it also uses a second variable y.
y will be 0 when no one is around at all.
i sets $y := i$ during the "fast path", but never checks this during the fast path, just goes to the CS.

On the other hand, if x is discovered to have changed, i now waits for a delay.
At the end of that delay, i checks y.
Now if this is still equal to i, x goes on to the CS (so, this is a second way it can enter the CS.

Otherwise, if the y check also comes out wrong, it just starts over.

Now the delay has to be long enough to ensure that any process j that might have read $y = 0$ before the assignment $y := i$ has already either restarted or entered and left the cs.
That means that the delay has to be very long—not just big enough for process steps, but bigger than CS time.

Probably not such a reasonable condition.
So I won't say any more about this algorithm.

3.2.3 A non-timing-dependent algorithm

However, I bothered to introduce the timing-dependent algorithm because it leads to another, non-timing-dependent algorithm, which is represented by the generic code on p. 7.

(code p. 7)

Notice this has the same structure as the timing-dependent one, but:
Adds labels. Has assertions.
Has a mysterious "achieve P", instead of the delay. This has to be pinned down to make this an actual algorithm.

What it actually is, is (see p. 8):
"If y = i then no j is in (or just before or after) the CS, nor at γ, nor at δ."

Of course, that's not a real program statement for i, because it involves checking other processes' states.

One way to ensure this is via the delay given on p. 5.
But we're not pursuing this any further...
Another way is represented by the more detailed code on p. 6.

(code p. 6)

Now each process i has a personal flag variable $b[i]$, which it sets to indicate that it is beginning and unsets when it either leaves the fast path or finishes the CS.

Checking that all these variables are false is enough for process i to guarantee that no one is either in the CS, nor at γ or δ.

Now, this isn't completely obvious to me.
It seems to me that it is possible for some process to be in the CS while its flag is false.
This is because when it sees $x \neq i$ in line δ, it first sets $b[i]$ to false, then waits, and might still get into the CS.

However, on p. 11, it indicates that "$b[j]$ is true when process j is in $\gamma - \eta$"—but this region of code includes the CS.
???

The property as stated on p. 8 still seems to me to be true—the proof sketch in the paper just doesn't seem to explain why.
More work is needed for a convincing argument.

Q: Can you find a convincing proof? What's the best way to prove this? Maybe some kind of mapping from the detailed algorithm to the abstract version?
Try to use a model-checker (for bounded n) or other automated methods?

Anyway, the main interesting work done in the paper is in the proof of the generic algorithm.
This proof looks more persuasive, because it's assertional and more complete.

The idea is to state several invariants.
We're used to stating global invariants about the global state of a distributed algorithm.
He's doing something different: using Owicki-Gries-style conventions for writing invariants.

Namely, he annotates the program text itself with assertions that involve the global state.
Such assertions should be interpreted as:
If control of this process is here, then this is true of the global state...

The full set of invariants for the algorithm is then a bunch of conditionals of that form.
To show that these are really invariants, we have to show that all the conditionals are true in the start state and that they are preserved by all steps (as usual).

The start case, for this algorithm, is trivial, because there are no assertions labeling the initial contol points.

For the steps, the fact that this is done with annotations means that the proof of a conditional property breaks down naturally into two kinds of cases: when the process whose annotation we are considering is the one taking the step, and when it is someone else.
The proofs of the cases are the usual kind of algebraic deduction, however.

In this case, the generic algorithm, there are three interesting invariants.
The most interesting is the one associated with ϵ:
"If $y = i$ then no j is in (or just before or after) the CS, nor at γ, nor at δ."

(i has managed to clean out all the dangerous places, by passing the tests.)

The other two properties:
δ: "If we (still) have $x = i$, then $y \neq 0$."

η (for the CS): $y \neq 0$, and
for all $j \neq i$, j is not in CS, and if j is at the "dangerous places" γ and δ, then $x \neq j$.

The third of these implies m.e.

Not terribly enlightening.
All I can say is that he proves that the conjunction is preserved.

The use of concurrency in this algorithm, as for many in this sub-discipline, is very low-level, fine-grained.
Have to reason at a detailed level.
Lampson: "hard concurrency".

A good algorithm design aim is to create "building blocks" that encapsulate the such low-level reasoning.
Do it once, use the solutions repeatedly.
That's "easy concurrency".

The flavor of algorithms like this mean that they are good candidates for formal proofs, using invariants and deduction.
Small cases can also be model-checked.

3.2.4 Not lockout-free

Because a process that sees a problem goes all the way back to the beginning.
Next time, it could see a problem again, etc.

3.3 Significance

Fast path idea.

Leads naturally to the idea of algorithms for which good cases have correspondingly good performance.
This idea reappears in many other papers, of which we'll see a few.

The structure of the algorithm reappears in many other papers.

3.4 Work this suggests

1. Look for similar mutual exclusion algorithms that exhibit more graceful degradation of performance when contention increases.
   Latency, even number of variables, might depend on the amount of contention, not blow up to n as soon as there is any contention at all.

2. Contention-sensitive algorithms for other problems.

3. (For those who are interested in proofs) A nice mapping proof for the more detailed algorithm?

4. Add stronger properties, like lockout-freedom.

5. Consider lower bounds.

4 Friedberg, Peterson

4.1 Overview

One more (little) paper today.

Again, solves mutual exclusion, with a fast path.
This time no lockout.

Now, instead of just read/write variables, he uses:
two read-write binary variables and two "weak binary semaphores".

Works for an arbitrary number of processes.
In contrast to Lamport.

What's a weak binary semaphore?
A shared object that can be owned by exactly one process at a time.
Accessed by P and V operations, where P is an "acquire" and V is a "release".
When someone invokes P, they block, waiting for a (positive) response.
When someone invokes V, they (get an immediate response and) go on.
Generally, it is assumed that the V is instantaneous.

The semaphore keeps track of the pending invocations.
It guarantees some fairness, defined by the "weak" property:
If process i does a V (gets a V response?) and at that instant, some other process is waiting at the semaphore, and if i subsequently invokes another P, it does not get the grant until at least one other process has gotten a grant.

That is, i doesn't get two consecutive grants if at the moment it returned its first grant someone else was waiting.

This doesn't say that all the waiting processes get ahead of i—just at least one.

A weak semaphore yields mutual exclusion and a weak kind of fairness.
The main problem is to use such a semaphore to get mutual exclusion and a stronger kind of fairness, lockout-freedom.

4.2 Technical results

The main idea to obtain fairness is to use two semaphores S0 and S1.
A process that arrives waits on one of them, say S0.
It checks a shared variable Queue to determine which semaphore to wait on.

When several processes arrive in a row, and find the same value for Queue, they all wait on the same semaphore.
At some point, a cutoff happens, and later processees start waiting on the other semaphore, say S1.

The processes waiting on S0 are admitted one at a time, in an arbitrary order.
But they are all admitted before switching to those waiting on the other semaphore S1.
When the other semaphore S1 becomes active, the roles of the two queues are reversed, and new processes now start waiting on S0 again.

(code p. 344 and 345)

Process i (this is all independent of process names) remembers what semaphore to use, in MyQueue.
Then it waits for that semaphore.
When it gets it, it does a test.
This is to see if it is the first one of its batch to get that semaphore.
The first one will play a special role in managing the others waiting on the semaphore.
Called the "doorkeeper".

If it's the doorkeeper, it first grabs the second semaphore, thus taking charge of the entire system.
We don't want two doorkeepers around at once.
It then flips the Queue bit, so new processes will wait on the other semaphore.

Then it goes through a loop letting all the other processes in its batch go to the CS.
It does this by setting a flag NoOthers and giving up the current semaphore.
What's supposed to happen now is that another process waiting, if there is any, will pick up the semaphore (the "weak" property will guarantee someone else will get it, if there is any) and go to CS, then unset the NoOthers flag.

The doorkeeper notices that the flag has been unset, knows someone else has just been admitted, and repeats this.
Eventually, the doorkeeper will notice that the flag has not been unset, but it got the semaphore twice in a row.
That means no one else is left waiting in its batch, and it can now go to the CS, give up both semaphores and leave.

On the other hand, a process that isn't the doorkeeper just waits passively on its semaphore.
When it gets it, it goes directly to the CS, unsets the flag, gives up the semaphore and leaves.

This all sounds simple and believable.
However, there is a technical wrinkle here, described on p. 344:

Suppose a process i comes in, reads Queue to determine which semaphore it's going to wait on, but doesn't actually start waiting on the semaphore yet.
Meanwhile, the doorkeeper switches the Queue.
And the doorkeeper does its entire loop, finishing up and exiting, all without hearing about process i, because i hasn't yet started waiting.

The "late tenant" will, according to this code, eventually start waiting on its semaphore, and eventually get in.

Claims that this does not violate mutual exclusion or liveness.
The proofs here are only sketches, so it's hard to be certain.
E.g., Lemma 3.2 is a temporal, plausibility argument, but not a proof that tries to cover all the cases.

Needs a real proof.

They also claim a 2-bounded-waiting property: that no process is bypassed by another more than 2 times.

4.3 Significance

Mutual exclusion, liveness, with small finite number of very small variables. Fast path, low latency when running alone.

Also, what if there are only a few running concurrently? Say, a batch of size k and then no new entrants stop. It seems that the time is linear in k. So, this exhibits a nice contention-sensitivity property.

However, some of the variables aren't just simple registers—they have some built-in fairness, which makes it easier to obtain the higher-level fairness property required.
It would be nicer to just have reads and writes.
In particular, the semaphores don't seem like the sort of thing one would build in a distributed setting—arise from standard operating systems.

Does the 2-bounded-waiting property yield a contention-sensitivity result?
Get a corollary that bounds the latency in terms of some kind of concurrency bound?

Low-level hard concurrency again. Proofs not convincing. State of the art today permits better proofs; it would be nice if this were the norm. Anyway, it would be nice to have more structure.

4.4 Future work

1. Prove good contention-sensitivity bounds.
2. Better proof?
3. Instead of weak semaphores, what exactly is possible with just reads and writes?
4. Lower bound results?

5 Mutual exclusion with read/write shared variables

Q: Is it possible to solve mutual exclusion with unknown participants (from potentially infinite index set), with finitely many read/write variables?

A: No. n processes require n r/w variables, by Burns-Lynch result.
That's true even if we just want basic mutual exclusion and deadlock-freedom.

This implies that we need infinitely many variables for infinitely many processes.
But that may be OK.
We might not need to use all the variables if not too many participants arrive. Also, it doesn't rule out algorithms that are fast (low latency) when contention is low.

6 For next time:

Read Anderson-Moir,
Attiya-Fouren "Adaptive and Efficient" journal paper.
Can skim the other renaming and snapshot/collect papers.