Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 5
September 19, 2002

Fault-Tolerant Objects

Bring:
Extra notes for scribe.

Today's papers:
Carrying over from last time:
Merritt, Taubenfeld
Gafni, Merritt, Taubenfeld
New:
*** Afek, Greenberg, Merritt, Taubenfeld, Computing with Faulty Shared Objects
*** Jayanti, Chandra, Toueg, Fault-Tolerant Wait-Free Shared Objects

Next time: Disk Paxos papers

1 Introduction

Last time, we were working on the Merritt, Taubenfeld paper on computing with infinitely many processes.
Recall this was in the model with reliable shared variables.
We covered an impossibility result saying that, even if we know how many processes will participate (out of a bigger set of processes), we can't solve a simple leader election problem with bounded memory (read/write shared variables).
Then for contrast, we did an algorithm that solved leader election with only two read/write shared variables, but they were allowed to take on unboundedly many values.
In particular, they could record process ids, in fact, sets of at most c process identifiers, where c is a lower bound on participation and upper bound on concurrency.

Today, I want to describe one more result from this paper, because it connects with the mutual exclusion results we talked about on the first day of class.

This is a very powerful result for mutual exclusion using read/write memory.
It guarantees lockout-freedom.
It tolerates unbounded concurrency.
It's adaptive (in the point contention sense), and symmetric (processes are identical except for ids, and use their ids only in comparisons for equality).
No participation is required (as usual for mutual exclusion).

After than, I'd like to present one result from the Gafni, Merritt, Taubenfeld paper—a snapshot algorithm that works for unbounded concurrency.

Then go on to the two papers on computing with faulty objects.

2 Merritt, Taubenfeld

2.1 Adaptive, symmetric mutual exclusion algorithm with read/write registers

Theorem 3.2, p. 12.

They first present a deadlock-free m.e. algorithm and then sketch how to modify it to make it lockout-free. I'm not completely sure of the adaptivity claim here.

2.1.1 A new kind of splitter

They use a slightly modified splitter, p. 13.
Prop. 3.1 lists the properties they now want; they are slightly stronger than what we have seen so far.
Namely, they divide the processes into "early" and "late", where "late" means that they invoke the splitter after the first time when some process exits it.
"Early" are the others—those who enter it before the first exit.

Now instead of just saying "not all processes go right", they make the stronger statement: "not all the early processes go right".

And instead of just saying "not all processes go down", they strengthen this to say (1) the latecomers all move right, and (2) either at least one latecomer moves right, or else at most n-1 (early) processes move down.

(Actually, I think this should say: (2) either at least one latecomer does its first assignment, or else at most n-1 (early) processes move down. Since the splitter is wait-free, processes are supposed to be allowed to stop anytime. Therefore, someone could do the assignment to y but never actually move right.)

Lamport's algorithm still satisfies this slightly stronger statements.

Now they modify the splitter, weakening the liveness guarantee to lockout-freedom, rather than wait-freedom.
This is weaker because it counts on everyone continuing to take steps, that is, it assumes no failures.

What they gain from this is an additional property: If any process wins then no one moves down.

See code, p. 15.

The detailed algorithms isn't exciting.
They add a couple of new flags, b, and z.
The await statement adds some dependencies, so termination is only ensured if everyone keeps taking steps. LTTR.

The new flags ensure the new safety property: If anyone wins then no one moves down.

(1) We can't have someone winning while someone goes down as a result of the final "else move down", on line 9.
That's because this works like Lamport's splitter, and no 2 processes get to the corresponding point in Lamport's splitter.

(2) We can't have someone winning while another goes down as a result of the first "move down" statement, on line 6.
That's because a process can only win if it sees that z is set to 1 before b is first set to 1.
But a process can only move down on line 6 if b is set to 1 before z is first set to 1.
Incompatible.

The other properties needed for the Lamport splitter carry over.
LTTR.

2.1.2 The mutual exclusion algorithm, using the splitters

A linear array of new-splitters (going downward).

See code, p. 16, figure 6.

It's just what you'd expect, except that moving right is transformed into another await statement, which waits for a shared variable "next" to get set to a new splitter, larger than the one where you're currently contending.
At that point, the process goes back to the beginning of its code, but starting with the new splitter indicated in "next".
Thus, as time goes on, the splitters that are used for competition progress down the array.

At the first splitter of a competition, all the latecomers move right, so only finitely many processes move down.
Each splitter either has a winner (who is the process who gets the critical section), or else eliminates at least one process who moves right.
So, within finitely many levels, a process wins.

After the winner of a competition leaves the CS, it sets the shared var next to the next unused level, to indicate where the next competition should start.

Thus, successive competitions begin at later and later splitters.
The properties of the new-splitter ensure that when a process wins at a level, no process is already active in any splitter at a greater level.

Satisfies mutual exclusion, no deadlock.

Time complexity: Measure from when no one is in CS and someone is waiting until someone enters CS.
At worst, someone has to progress through $O(n)$ splitters before entering CS, where n is the number of early processes at the "starting splitter" for the new competition.
Time $O(n)$.
Since $n \leq$ point contention, that gives an adaptive time bound.

Q: Does this work with ordinary Lamport splitters, if we use a 2-dimensional array and start each competition at the top of a new column?

A: Not directly. Nothing would stop a second process from winning in a column, below the first splitter in that column where someone wins.

This algorithm doesn't guarantee lockout-freedom as is.
To add lockout-freedom, they hand-wave about adding some flags and having a process that leaves the CS examine the flags to help select the next process.
Are these per-process flags?
If so, then checking all of these sems to require some kind of global diagonalization over these flags, which seems to incur time delays that depend on the process ids. That wouldn't be adaptive. ???
Perhaps they are assuming use of renaming first...
Q: What about storing names in splitters?
A: Maybe could work, but the solution would still need some work.
Q: What about unbounded registers?
A: Remember that the algorithm needs to not depend on total number competing, so this might not work.
Anyway...Even deadlock-free m.e. is enough to solve leader election or consensus.
Using m.e., the first process selected can be the leader, can use its value for consensus.
This does seem to preserve the adaptivity and symmetry properties.

This result doesn't quite subsume Theorem 2.4, because that used only two registers, whereas this uses infinitely many.
(M.e. is a harder problem than leader election.)

2.2 Significance

They also revisit the Friedberg-Peterson result, which (recall) gives a lockout-free mutex algorithm using 2 weak semaphores and 2 r/w/shared vars.
MT just observe that, according to their new defs, FP allows unbounded concurrency, and has no required participation.
It is adaptive and symmetric.
The adaptive, symmetric read/write register results for leader election (Theorem 2.4) and m.e. (Theorem 3.2) seem quite powerful.

MT are trying not only to get individual algorithms but also general classification results.
They have set up a nice set of definitions to use for classification: concurrency and participation bounds, adaptivity, symmetry.
They would like results that say exactly which problems can be solved under which of these assumptions, with which properties.

2.3 Further work

Needs polishing. Lots of things don't quite work or aren't well explained.

Continue working on the classification theory:

- Exactly which problems can be solved under which sets of concurrency/participation assumptions, with which properties? Which results imply which other results?
- Extend this to fault-tolerant algorithms, for problems for which we normally consider this, such as renaming, lattice agreement, snapshot, collect, consensus.
- Get separation theorems for different sets of requirements. E.g., are there interesting problems that can be solved for any finite, known number of processes but can't be solved for a finite, unknown number of processes?
- General transformations to convert algorithms for finite numbers of processes to algorithms for infinitely many processes (perhaps under certain assumptions, like symmetry)?

3 Gafni, Merritt, Taubenfeld

3.1 Overview

Continuing on the same theme.
Unlike the other paper, they are now allowing process stopping faults, in fact, they are looking for wait-free algorithms.
Now the problems are things like snapshot, renaming, etc., for which wait-free algorithms are usually considered.
They consider read/write registers only.

The paper has two different foci:

1. They are trying to get a general classification theory for problems, based on the kind of concurrency assumptions that are required for solving the problems. The most interesting case is "unbounded concurrency", as defined in the MT paper. Any number of processes might participate at once.
2. They try to obtain algorithms that work with unbounded concurrency.

This paper does not deal with participation lower bounds.

Section 2 contains hierarchy results.
They show that "there exist problems" that can be solved with bounded concurrency but not unbounded concurrency, and also problems that can be solved with k-bounded concurrency but not $k+1$-bounded.
These are bounded versions of snapshot problems.
Somewhat contrived.

Section 3 gives algorithms that work with unbounded concurrency.
In particular, Section 3.1 focuses on a cute snapshot algorithm that handles unbounded concurrency.
Its time complexity, in the initial presentation, depends not only on the actual contention but on the process indices, so it's not "adaptive".
Then they describe how to make it adaptive by doing renaming.

The kinds of restrictions on concurrency they consider are:

- k-bounded: A finite bound k on the number of processes that are active at any point in any execution (point contention).
- bounded: In each execution, the concurrency is bounded by some k.
- unbounded: No bound on the concurrency.

3.2 The hierarchy results

They show the existence of a problem that can be solved with bounded concurrency but not unbounded.
And for each k, a problem that can be solved with k concurrency but not $k+1$.

Bounded snapshot problem: They fomulate the atomic snapshot problem in a somewhat simplified, unusual way. In fact, it looks to me just like lattice agreement.

Consider one-shot only—everyone invokes at most once.

Only one kind of operation, rather than separate update and scan operations as in the Collect service.
The operation is supposed to return a set of processes, all of which have invoked.
Each process' set should include itself, and the sets returned should be totally ordered by containment.

New twist: For each execution, there is some bound r on the difference between the sizes of two consecutive sets in the nesting order.

(I'm not sure exactly what this means. The setting is wait-free, so processes can fail. Are we allowed to assign sets to processes that don't finish their operations? Probably so.)

Theorem 2.1: The bounded snapshot problem isn't solvable is the unbounded concurrency model.

(It's trivially solvable with bounded concurrency—just use an ordinary snapshot algorithm.
If concurrency is bounded in an execution by, say, c, then in that execution, the difference between the sizes of two consecutive snapshots will be at most c.

Proof: Enumerate the sets returned in nondecreasing order, counting repeats.
The smallest set in this sequence must have size between 1 and c, inclusive.
That's because someone terminates before the $c+1$st process starts.
The second smallest must have size between 2 and $c+1$, because two must terminate before the $c+2$nd process starts.
(And the second must contain the first.)
Etc.
The kth must have size between k and $c+k-1$, inclusive.

Then the difference between two consecutive entries in this list is at most $(c+k-1)-(k-1)=c$.)

To see that it's not solvable with unbounded concurrency:
Divide the processes into batches, $G_1, G_2, \ldots$, each with more processes than the preceding one.

They claim that if a batch comes in all together, it must be possible for them all to return the entire batch.
Since the batches get larger and larger, this isn't bounded.

The claim about them all returning the entire batch is "proved" in a separate lemma.
The proof seems rather odd to me, and I'm not convinced (though I'd bet the result is correct).
The proof they give is based on the lower bound of $2k-1$ for renaming, and uses a sort of reduction.
I think this needs some work. LTTR.
QA: A clarification about the issue is that a reduction argument should produce an algorithm and the authors don't do that here.
A similar construction is used to "separate" k-concurrency from $k+1$-concurrency, based on a k-bounded version of the snapshot problem.
This is similar to "bounded snapshot", except the difference is always at most k.

3.3 One-shot snapshot algorithm whose latency depends on contention and process indices

Theorem 3.1 gives an (ordinary, not bounded) snapshot algorithm that handles unbounded concurrency. Interesting.

The time complexity of the basic algorithm depends on both the concurrency that happens to occur, and the maximum index of a participant.
But then they show, using various tricks, how to remove the dependency on the max index.

One-shot version, but any number of processes.
(Could allow repeated visits simply by letting each process use multiple names.)

Code, p. 5.

The main data structures used by this code are an infinite array of start bits and an infinite array of flag bits.
There's also an infinite array of "snap" sets, but let's ignore these for the moment.
(Basically, they provide short-cuts—answers that someone has already calculated, that another process can borrow rather than completing the work of doing its own snapshot.)

Trace through the code without looking at the snap values.

Process i arrives, sets its start[i] flag, then proceeds to collect the names of other processes who have set their start flags.

How does it know where to look? It can't just read all the registers.
It traverses registers from the smallest as long as the flag bits are set.
It accumulates, in its set "collect", the indices of processes whose start bits are set.

It does this again.
The second time (and subsequent times) it saves the result of its previous collect in its set "doublecollect".
Keeping two sets will allow it to compare the results it obtains on two successive attempts.

A stopping condition is that collect = doublecollect (so nothing new has been found on the last pass), and also i itself is in the set.
Until this condition is reached, the process just keeps making attempts.

In one situation, it takes some positive action before beginning the next attempt:
(line 10) If collect = doublecollect but i itself isn't in the set.
That means that the traversal didn't get far enough—didn't reach the i entry.
That must be because not enough flags were set.
So in this case, before beginning the next attempt, process i sets its own flag[i] and all smaller flag variables.
Then tries again.

The next time it sees collect = doublecollect, it will also have i itself in the set, because it is guaranteed to reach i the next time (enough flags are set).

From what has been said so far, it is still possible that process i might continue making attempts forever, and on each pass, learn about more new entrants.
That could cause nontermination.
To get around this, we use the same trick as in the earlier algorithm of Afek, Attiya,..., and let a process that determines a good snapshot record it someplace for other processes to use.

In this case, a "good" snapshot, worth recording, is one that sees collect = doublecollect, and where the process itself isn't in the set.
(Q: Why make that last restriction?)
So in this case, just before process i sets the flags, it also records the "good" snapshot.

Now look how the snapshots are used in the code: in line 5, 9, 15, as a stopping condition—when i sees someone else's snap that already contains i, i can just take that snapshot and stop.

The safety arguments aren't given in this paper but are apparently (seems believable) analogous to those for standard snapshot.

Q: Work this out? The argument from the standard case doesn't seem to carry over immediately. First, the snapshot that someone writes isn't necessarily a "true" snapshot at any time in its interval. And, it's not obvious that the interval of the operation that produces a snap is included in the interval of an operation that uses it. ???

Termination:
Process i can't loop forever reading a fixed number of shared variables.
(Or would eventually see collect = doublecollect, set flags, see again, terminate.)
So the only problem is if a process, over time, reads all the flag variables, 1,2,3,...
Suppose it does.

After some point, we have $flag[k] = 1$ for all $k \leq i$.
At that point, let $max \geq i$ be the maximum id of any process that has ever been active up to this point.
Now, eventually $flag[max + 1]$ has to get set to 1, and this has to be done by some process with

index $p \geq max + 1$.
Before it does this, it also sets $snap[p]$ to a snapshot set.
That set must contain i, because the snapshot that produces this set starts after the flags are already set through i.
So eventually i sees this.

Key idea: The flag bits are set only after a new snap has been written.
Thus, a process helps other processes out (by supplying a snapshot for them) before extending the flag prefix (which could cause other processes to go on forever).

Can extend this easily to a long-lived algorithm, with multiple snapshots per process, by allocating multiple indices per process.

Not adaptive, latency depends on process ids.

3.4 Making the snapshot adaptive

They claim that they can modify this to get the algorithm to be adaptive—latency depending on actual total number of participants.
Still handles unbounded concurrency, still one-shot.

Do this by using a one-shot $f(k)$-renaming algorithm that is adaptive (latency depends on k), and handles potentially unbounded number of participants, arriving in any concurrency pattern.

First apply the renaming algorithm to reduce the set of names so that they depend on the actual number of arrivals.
Then use the resulting names in the snapshot algorithm just sketched.

Getting such a renaming algorithm:
Take a standard one-shot adaptive renaming algorithm like Attiya-Fouren's, which uses an upper bound n on the size of the namespace, and modify it so it doesn't assume such an upper bound.

Algorithms like AF (they say) can be expressed in a form where they compete for names in order, 1,2,3,...; when they fail to get one, they try to get the next one. (Check this—that isn't exactly how AF's algorithm is described.)

"Interleaving trick": Let the processes in AF compete for only the odd names.
If process i fails to acquire any odd number through $2i - 1$, it just chooses $2i$.
Thus, it defaults to twice its own index, a name that is reserved for it.
Although this depends on i, it's OK here because the fact that i reaches this point means that at least (approx) i processes are participating.

Alternatively, they give a second trick, which works for all adaptive one-shot renaming algorithms, but causes the namespace to be squared.
LTTR.

Go over if time, from p. 6.

Other results in the paper extend these to long-lived algorithms for renaming, collect, and snapshot, all for unbounded concurrency.

3.5 Significance

The algorithms seem much more interesting than the hierarchy results.
Unbounded processes, with unbounded concurrency, can solve all kinds of problems in a wait-free manner.
Using just (infinitely many) read/write registers.
Costs can be made adaptive, based on number of actual participants.

3.6 Future work

Lots of details and clean-up needed.

Some more digesting, to turn this into a general theory with the best results presented in the best way, the best modularity, etc.

What does all this have to do with real distributed computing?

4 Afek, Greenberg, Merritt, and Taubenfeld

Cover: Sections 1-4, 5.1-2, 6.1, 6.4

4.1 Overview

This and the Jayanti paper have a lot of overlap. It seems to me that the overlapping part is the most interesting. I'll emphasize that.

AGMT consider faulty objects instead of reliable shared variables.
Faults: Objects can return completely wrong results. However, they aren't allowed to not respond.
So, this is a kind of "synchronous Byzantine" model.
Processes are also allowed to fail by stopping—they consider wait-free algorithms.

Interesting results:

1. Theorem 2. A "fault-tolerance-boosting" result, showing how implementations that tolerate small numbers of faulty objects can be combined to tolerate any larger number of faulty objects (but more total objects are needed).
2. Corollary 8. A reliable atomic read/write register from atomic read/write registers, some of which may be faulty.
3. Theorem 10. A reliable test-and-set register from test-and-set registers, some of which may be faulty, plus reliable atomic read/write registers.
4. Theorem 12. A reliable consensus object from RMW registers, some of which may be faulty.

5. Theorem 18. A "universality" result, which say that "all objects" can be implemented reliably in terms of certain kinds of basic objects (consensus objects or RMW objects), where some of the basic object can be faulty.

They consider situations where the number of faults that can occur in some objects is bounded (their k parameter).
But I don't see any interesting results about this.
So I'll just restrict attention to the bound f, on the number of faulty objects.

They use the notation $CONST(X, (f, \infty), Y)$ to denote the minimum number of n-process objects of type X that are needed to implement a reliable n-process object of type Y, if at most f of the objects may be faulty.
QA: The ∞ refers to the number of arbitrary state changes a faulty object can make.
Most of their results are stated in terms of this CONST measure.

The results in this paper seem basically easy.

4.2 Composition results

p. 8, Theorem 2 describes a way of "boosting" fault-tolerance.

It bounds the number of objects of type X needed to implement a reliable object of type X, assuming f of the lower-level objects may be faulty.

This bound is expressed in terms of the number of objects that are needed to implement X from X tolerating t faulty objects, where $t < f$.
Namely, if $(t+1)^L$ objects are sufficient to implement X, tolerating t faults, then $((t+1)f)^L$ are enough to implement it, tolerating f faults.

The construction is recursive.

E.g., consider an arbitrary f that we want to tolerate, and assume for simplicity that f is a multiple of $t+1$.
Use $C = (t+1)^L$ "intermediate objects" of type X, each of which tolerates up to $f/(t+1)$ faulty objects in its own implementation.
(These will be constructed recursively.)
Use the assumed t-fault-tolerant construction at the top level.
This will work if at most t of the C intermediate objects are faulty.

Suppose that at least $t+1$ of the intermediate objects are faulty.
Since each of these tolerates $f/(t+1)$ faults among its implementing objects, that means there must be at least $(t+1)(f/(t+1)+1)$ total faulty objects among all the low-level objects.
This is greater than the f failures that the entire implementation needs to be able to tolerate.

This argument provides the inductive step for fault-tolerance of the entire algorithm.
It all bottoms out when we reach $f = 1$.

How many objects are needed?
If $f = 2^k$, this expands exponentially to give C^{k+1}.

4.3 Atomic register constructions

The main result here is Corollary 8 (err, 4.4), p. 12. Says that an atomic read/write register can be implemented from $20f + 8$ atomic read/write registers, tolerating f faulty registers.

The direct construction for this isn't actually in this paper, but they refer to the conference version.

Here, they outline various approaches to implementing atomic-from-faulty-atomic, though I don't think they get exactly the same number in this way. But here's the general idea:

Rely on standard results that show how to implement reliable atomic registers in terms of weaker forms of registers—not atomic, but "safe" (defined by Lamport).
All the lower-level registers are assumed to be reliable in these constructions.
These now-standard constructions are well known.

In this paper, they "bootstrap" the process by showing explicitly how to implement a safe register in terms of $2f + 1$ safe registers where f may be faulty.
Basically, a writer writes all the registers, and a reader reads all the registers.
The reader sees if there's a majority for one value and if so returns it. If not, it returns an arbitrary value.

Now, they use a standard construction implementing an atomic register from reliable safe registers at the top level.
Now implement each reliable safe register as above—using unreliable safe registers.
Each individual implementation of a reliable safe register has to be able to tolerate f failures.

For example, they mention a construction by Tromp, which implements an atomic V-register using 4 safe V-registers and 8 safe binary registers.
As above, each of the safe registers can be implemented using $2f + 1$ unreliable safe registers.
The total number of registers is then $12(2f + 1) = 24f + 12$.

4.4 Test-and-set constructions

Now, to see an actual example or two of their algorithms.
Section 5.1 and 5.2 construct a reliable "single-use test-and-set" object from unreliable SUT&S objects, where at most 1 may be unreliable.
They can use the boosting idea to boost the fault-tolerance to any amount.
Just consider 1 fault.

SUT&S: Very simple primitive, of the sort available in hardware.
Binary values, 0 and 1, initially 0.
One operation, T&S.
Sequential specification says that any T&S operation writes 1 into the register but returns the value the register previously had.

Thus, the first process should get 0 and everyone else should get 1.
Of course, this is an atomic version, so several processes can invoke it concurrently.
The results are supposed to be serialized at some points in their intervals.

2-process algorithm:
Uses 7 low-level registers, tolerates 1 of them being faulty.
Cute, see Figure 3, inner box.

See code top of figure 2:
Process does T&S on the three A registers, and counts how many he "wins" ($0 =$ win here).
If he doesn't win at least 2, he tests the single register B; if he wins that, he returns 1 (loses).

Otherwise, that is, if he either wins at least 2 in the A array or else loses (gets 1) in B, he goes on to compete using the C array. That decides wheter he wins or loses.

Since we are assuming that there is only one faulty register out of the 7, we can argue that this works using a case analysis based on which is the faulty register.
We have to show that one gets a 0 and one gets a 1.
Moreover, for this to be linearizable, we also have to argue that a process can't return a 1 before the other process (the one that returns 0) even starts.
First, a couple of claims:

Claim 1: If no faulty register appears in A and two processes reach A, then one wins A and one loses A. Likewise for C.

Claim 2: If only one process reaches A then it wins. (Regardless of whether the faulty register is in A.)
Likewise for C.
This is because only one of the registers in A can be faulty.
Cases:

1. The faulty register (if any) is in A.
Then it could cause the two processes to take the same branch (either) or different branches.

If both take the "win" branch, then one wins and one loses in C, by Claim 1 applied to C.

If they both take the "lose" branch, then one wins at B and returns 1 for the whole algorithm, and the other goes to the C array. But since he's the only one in the C array, Claim 2 implies that he wins and outputs 0.

If one takes each branch, then the one that goes to B must win at B (he's alone, and B is nonfaulty), and so outputs 1, whereas the one that goes to C must output 0 by Claim 2.

2. The faulty register is B.
Then by Claim 1, A sends one process to C and one to B.
B can send its one participant either way. If it lets it win B, it moves down and outputs 1. But then the other process is alone in C, which is correct, so by Claim 2, it wins C and outputs 0.

If it sends its participant to C, then two compete in a correct C, so Claim 1 implies that one wins and one loses.

3. The faulty register is in C.
Then A and B are correct, which means that one of the processes goes to B, wins there and outputs 1.
The other is alone in C.
Then Claim 2 implies that it wins C and outputs 0.

We still have to argue that a process can't return 1 before the other process arrives.
A process that arrives alone will win at A and C, by Claim 2, and return 0.

This is extended to an n-process algorithm still tolerating only one fault, by adding a doorway that allows at least one and at most two processes though.
Basically, to enter the main algorithm, you have to win some register at each of two preliminary stages.

That means at most two processes can do this, because at least one of the stages has both registers nonfaulty, and each process that wins must grab at least one of these registers.

Claim at least one process will win one at both stages:
Because at most one of the four registers is faulty.
Thus, some process wins one of the registers at stage one and enters stage two.
Then some process wins one of the registers at stage two.

The n-process algorithm can be boosted to an algorithm for T&S from T&S that tolerates any number of failures.