Fault-Tolerant Wait-Free Shared Objects

JAYANTI, CHANDRA AND TOUEG

(1998)

A presentation by

Behdad Esfahbod

behdad@cs.toronto.edu

CSC2415, April 14 2004

Problem Statement

- Concurrent asynchronous processes
- Typed linearizable shared objects

Problem Statement

- Concurrent asynchronous processes
- Typed linearizable shared objects
- Wait-free implementations
- Failure of processes

Problem Statement

- Concurrent asynchronous processes
- Typed linearizable shared objects
- Wait-free implementations
- Failure of processes
- Failure of base objects *(new)*

Object Failures

Object Failures

Responsive

Object Failures

Responsive

- Crash

Object Failures

Responsive

- Crash

- Arbitrary

Object Failures

Responsive

- Crash
- Omission
- Arbitrary

Object Failures

Responsive

- Crash
- Omission
- Arbitrary

Non-responsive

Object Failures

Responsive

- Crash
- Omission
- Arbitrary

Non-responsive

- NR-Crash

Object Failures

Responsive

- Crash
- Omission
- Arbitrary

Non-responsive

- NR-Crash
- NR-Arbitrary

Object Failures

Responsive

- Crash
- Omission
- Arbitrary

Non-responsive

- NR-Crash
- NR-Omission
- NR-Arbitrary

Some Terminology

A t-tolerant implementation $\mathcal{I}$ for failure mode $\mathcal{F}$

- Wait-free
- Correct
- At most t base objects fail by $\mathcal{F}$

More Terminology

- Resource complexity

More Terminology

- Resource complexity
- Self-implementation

More Terminology

- Resource complexity
- Self-implementation
- **Gracefully degrading**

Organization

- Model and Definitions *(12 pages)*
- Tolerating responsive failures *(11 pages)*
- Tolerating non-responsive failures *(5 pages)*
- Feasibility of graceful degradation *(15 pages)*

I/O Automata

- Non-deterministic automaton
- Finite/infinite set of states, including starting *state*s
- Sets of input/output/internal events
- Transition relation (s, e, s'), the *step*s

I/O Automata (continued)

Enabled event e at state s:

$\exists s' : (s, e, s') \in$ transition relation.

Execution of automaton A: $s_0, e_1, s_1, e_2, s_2, \ldots$.

History of an execution: $e_1, e_2, \ldots$.

I/O Automata (continued)

A new automaton A can be constructed by *composing* a set of "compatible" automata $A_1, A_2, \ldots, A_k$.

The *history of a component* A_i is denoted by $H|A_i$.

Object Type

Type T is a tuple (OP, RES, G, τ)

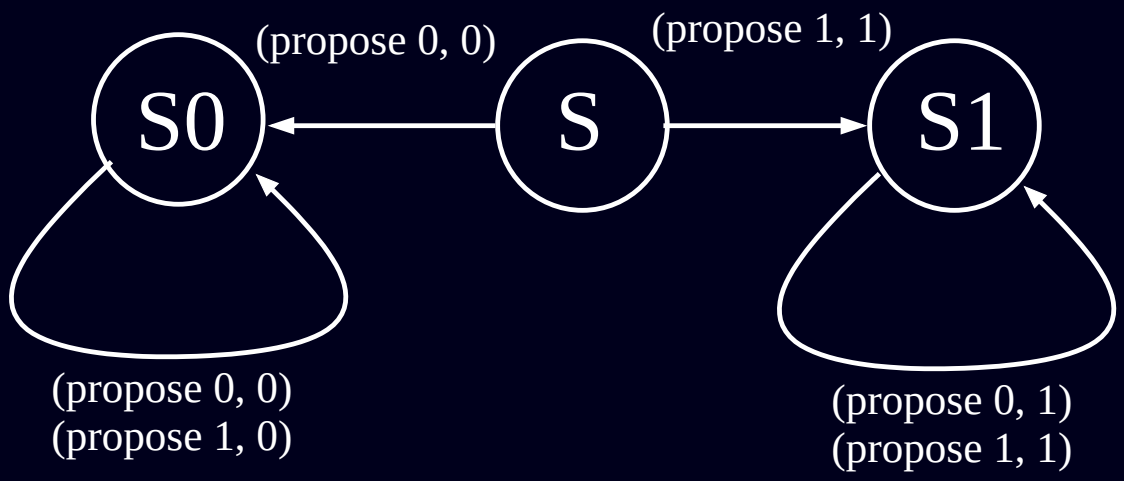
- OP is the set of *operations*
- RES is the set of *responses*
- G is the *sequential specification* of T
- τ, the *transformation functions* is explained later!

T is deterministic...

T is total... We only deal with total types.

T is finite...

Object Type (continued)



Sequential specification of `consensus`

Objects and Processes

Both are modeled as automata

Objects and Processes

Both are modeled as automata

Obect O:

- type T
- initial state s of T

Objects and Processes

Both are modeled as automata

Obect O:

- type T
- initial state s of T

Processes

- can be made to crash
- are deterministic, unless mentioned otherwise

Concurrent System

Is the automaton composed from

- Processes $P_1, P_2, \ldots, P_n$, and
- Objects $O_1, O_2, \ldots, O_m$.

shown by $(P_1, P_2, \ldots, P_n; O_1, \ldots, O_m)$.

Concurrent System

Is the automaton composed from

- Processes $P_1, P_2, \ldots, P_n$, and
- Objects $O_1, O_2, \ldots, O_m$.

shown by $(P_1, P_2, \ldots, P_n; O_1, \ldots, O_m)$.

For each P_i and O_j and object operation op:

- $invoke(P_i, op, Oj)$
- $respond(P_i, op, Oj)$

Matching response r and invocation i.

An operation is an invocation and its matching response.

Incomplete operation: no matching response.

Operation p precedes q...

Concurrent operations.

Sequential history H: no concurrent operation.

Well-formed History

- No prefix of $H|P_i$ has more than one incomplete operation

- $(H|P_i)|O_j$ begins with an invocation and has alternating invocation and responses

Fairness

Execution E is *fair*:

- If E is finite, no internal or output event is enabled in the final state of E

Fairness

Execution E is *fair*:

- If E is finite, no internal or output event is enabled in the final state of E

- If E is infinite, for each internal or output event e, E contains either infinitely many occurrences of e or infinitely many states in which e is not enabled

Linearizability

A *linearization* of H with respect to (T, s) is a sequential history S which:

- is legal from state s of T
- includes every complete operation in H
- either does not include incomplete invokes, or includes them with an arbitrary response.
- includes no other operation
- if $oper <_H oper'$ then $oper <_S oper'$

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness.

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness. But not all objects are linearizable:

`safe register`

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness. But not all objects are linearizable:

`safe register`, `consensus with safe-reset`

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness. But not all objects are linearizable:

`safe register`, `consensus with safe-reset`, `1-reader 1-writer register`

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness. But not all objects are linearizable:

`safe register`, `consensus with safe-reset`, `1-reader 1-writer register`, `1-reader 1-writer safe register`.

Well-Behavedness

Linearizablity looks like a good measure for well-behavedness. But not all objects are linearizable: `safe register`, `consensus with safe-reset`, `1-reader 1-writer register`, `1-reader 1-writer safe register`.

For object O of type $T = (OP, RES, G, \tau)$ initialized to state s, and history H in execution E, we say that O is *well behaved* in E if $\tau(H)$ is linearizable with respect to (T, s).

Implementation

Let T be a type and s be a state of T.

$\mathcal{L} = (\mathcal{T}_\infty, \mathcal{T}_\in, \ldots)$ is a list of base types.

$\Sigma = (s_1, s_2, \ldots)$ is a list of initial states for T_i's.

$\mathcal{I}(\mathcal{O}_\infty, \mathcal{O}_\in, \ldots)$ is an *implementation of* (T, s) *from* $(\mathcal{L}, \Sigma)$ *for processes* $P_1, P_2, \ldots, P_N$.

Implementation (continued)

An implementation is *wait-free* if every derived object $\mathcal{O}$ has this property: if E is an execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$ in which all base objects of $\mathcal{O}$ are wait-free, then $\mathcal{O}$ is wait-free in E.

An implementation is *k-bounded wait-free* if it is wait-free and every derived object $\mathcal{O}$ has this property: if E is an execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$ and for all P_i, between an invocation on $\mathcal{O}$ by P_i and its matching response, P_i has *no more than k invocations on all base objects of $\mathcal{O}$* put together.

Failure Modes

A failed object may return special response $\perp$

A process does not invoke operations on $\mathcal{O}$ anymore after it receives $\perp$ from $\mathcal{O}$

Failure Modes
continued

Responsive Failure Modes

- Crash
- Omission
- Arbitrary

Failure Modes
continued

Crash

Object $\mathcal{O}$ fails in execution E by crash if it is not well-behaved, but:

- $\mathcal{O}$ is wait-free in E.
- Responses from $\mathcal{O}$ belong to $RES \cup \perp$. An operation that returns $\perp$ is an *aborted* operation.
- For operations op and op' in the history $\mathcal{H}$, if op precedes op' and op is an aborted operation, then op' is also an aborted operation.
- Let $\mathcal{H}'$ be the history obtained by removing all aborted operations in $\mathcal{H}$. Then, $\tau(\mathcal{H}')$ is linearizable with respect to (T, s).

Omission

Object $\mathcal{O}$ fails in execution E by omission if it is not well-behaved, but:

- $\mathcal{O}$ is wait-free in E.
- Responses from $\mathcal{O}$ belong to $RES \cup \perp$.
- Let $\mathcal{H}'$ be the history obtained by removing the response events associated with the aborted operations in $\mathcal{H}$. Then, $\tau(\mathcal{H}')$ is linearizable with respect to (T, s).

Failure Modes
continued

Arbitrary

Object $\mathcal{O}$ fails in execution E by the arbitrary failure mode if it is not well-behaved, but is *wait-free* in E.

Failure Modes
continued

Non-responsive Failure Modes

- NR-Crash
- NR-Omission
- NR-Arbitrary

Failure Modes
continued

NR-Crash

Object $\mathcal{O}$ fails in execution E by NR-crash if it is not wait-free, but:

- $\mathcal{O}$ is well behaved in E.
- The number of responses from $\mathcal{O}$ is finite.

Failure Modes
continued

NR-Omission

Object $\mathcal{O}$ fails in execution E by the NR-omission if it is not wait-free, but is *well behaved* in E.

Failure Modes
continued

NR-Arbitrary

Object $\mathcal{O}$ fails in execution E by the NR-arbitrary if it *fails* in E.

Fault-Tolerance

Implementation $\mathcal{I}$ is called *t-tolerant* if the derived object $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots)$, for every execution E of the concurrent system $(P_1, P_2, \ldots, P_N; \mathcal{O})$, if at most t objects among $O_1, O_2, \ldots$ fail, and they fail by $\mathcal{F}$, then $\mathcal{O}$ is correct.

Graceful Degradation

Implementation $\mathcal{I}$ is called *gracefully degrading for failure mode* $\mathcal{F}$ if the derived object $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots)$, for every execution E of the concurrent system $(P_1, P_2, \ldots, P_N; \mathcal{O})$, if all faulty objects among $O_1, O_2, \ldots$ fail by $\mathcal{F}$, then either $\mathcal{O}$ is correct or $\mathcal{O}$ fails by $\mathcal{F}$.

Fault-Tolerance & Graceful Degradation

Compositional Lemma

Suppose that T has a t-tolerant implementation from $\mathcal{L}$ for failure mode $\mathcal{F}$, where $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ is a list of types. Furthermore, suppose that each T_i has a t_i-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$. Then we have:

Fault-Tolerance & Graceful Degradation

Compositional Lemma

Suppose that T has a t-tolerant implementation from $\mathcal{L}$ for failure mode $\mathcal{F}$, where $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ is a list of types. Furthermore, suppose that each T_i has a t_i-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$. Then we have:

1. T has a *t'-tolerant* implementation from $\mathcal{L}'$ for failure mode $\mathcal{F}$, where $\mathcal{L}' = \mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \ldots \cdot \mathcal{L}_n$ and $t' = \mathit{MinSum}(t+1, \langle t_1+1, t_2+1, \ldots, t_n+1 \rangle) - 1$.

Fault-Tolerance & Graceful Degradation

Compositional Lemma

Suppose that T has a t-tolerant implementation from $\mathcal{L}$ for failure mode $\mathcal{F}$, where $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ is a list of types. Furthermore, suppose that each T_i has a t_i-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$. Then we have:

1. T has a *t'-tolerant* implementation from $\mathcal{L}'$ for failure mode $\mathcal{F}$, where $\mathcal{L}' = \mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \ldots \cdot \mathcal{L}_n$ and $t' = \mathit{MinSum}(t+1, \langle t_1+1, t_2+1, \ldots, t_n+1 \rangle) - 1$.

2. If the t-tolerant implementation of T from $\mathcal{L}$ is gracefully degrading for $\mathcal{F}$, then T has a t'-tolerant *gracefully degrading* implementation from $\mathcal{L}'$ for failure mode $\mathcal{F}$.

Fault-Tolerance & Graceful Degradation

Corollary — Introducing Fault-Tolerance

Suppose that T has a *(0-tolerant)* implementation from $(T_1, T_2, \ldots, T_n)$. Furthermore, suppose that each T_i has a t-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$, where $\mathcal{L}_i$ is some list of types. Then we have:

1. T has a *t-tolerant* implementation from $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \ldots \cdot \mathcal{L}_n$ for failure mode $\mathcal{F}$.

2. If the (0-tolerant) implementation of T from $(T_1, T_2, \ldots, T_n)$ is gracefully degrading for $\mathcal{F}$, then T has a t-tolerant *gracefully degrading* implementation from $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \ldots \cdot \mathcal{L}_n$ for failure mode $\mathcal{F}$.

Fault-Tolerance & Graceful Degradation

Corollary — Self Implementation

If T has a t-tolerant gracefully degrading *self-implementation* $\mathcal{I}$ of resource complexity n for failure mode $\mathcal{F}$, then T has a $(t^2 + 2t)$-*tolerant* gracefully degrading self-implementation $\mathcal{I}'$ of resource complexity n^2 for $\mathcal{F}$.

Fault-Tolerance & Graceful Degradation

Corollary — Self Implementation

If T has a t-tolerant gracefully degrading *self-implementation* $\mathcal{I}$ of resource complexity n for failure mode $\mathcal{F}$, then T has a $(t^2 + 2t)$-*tolerant* gracefully degrading self-implementation $\mathcal{I}'$ of resource complexity n^2 for $\mathcal{F}$.

Corollary — Booster Lemma

If T has a 1-*tolerant* gracefully degrading self-implementation of resource complexity k for failure mode $\mathcal{F}$, then T has a t-tolerant gracefully degrading self-implementation of resource complexity $O(t^{\log_2 k})$ for $\mathcal{F}$.

Fault-Tolerance & Graceful Degradation

Lemma — Graceful Degradation for Arbitrary Failures

If T has a t-tolerant *k-bounded* implementation from $\mathcal{L}$ for arbitrary failures, then T has a t-tolerant *gracefully degrading* k-bounded implementation from $\mathcal{L}$ for (responsive) arbitrary failures.

Tolerating Responsive Failures

Tolerating Responsive Failures

- `consensus`
- `register`
- Universal implementation

Consensus

- *Integrity:* all responses are either 0 or 1
- *Weak integrity:* all responses are either 0, 1, or $\perp$
- *Validity:* if there is a response $v \in \{0, 1\}$, then there has been an invocation of $propose\ v$
- *Agreement:* for any two responses $v_1, v_2 \in \{0, 1\}$, $v_1 = v_2$

Proposition — Correctness

Object $\mathcal{O}$ of type `consensus` is *correct* in E iff it:

- is wait-free,
- satisfies integrity,
- satisfies validity,
- satisfies agreement.

Proposition — Omission

Object $\mathcal{O}$ of type `consensus` *fails by omission* in E iff it fails in E and it:

- is wait-free,
- satisfies *weak* integrity,
- satisfies validity,
- satisfies agreement.

Crash and Omission

$O_1, O_2, \ldots, O_{t+1}$:
consensus objects, initialized to the uncommitted state

Procedure `Propose`(p, v_p, $\mathcal{O}$) /* $v_p \in \{0, 1\}$ */
  $estimate_p, w, k$: `integer` local to p
begin
  $estimate_p := v_p$
  for $k := 1$ to $t + 1$
    $w :=$ `propose`(p, $estimate_p$, O_k)
    if $w \neq \perp$ **then** $estimate_p := w$
  return($estimate_p$)
end

t-tolerant self-implementation of `consensus` for omission

Self-implementation.

$t + 1$ base objects, resource optimal.

Not trivial, all work done is in synchronous message passing systems.

We are in asynchronous shared-memory systems.

Is not gracefully degrading.

There is a gracefully degrading consensus for OMISSION, by $2t + 1$ which is optimal.

There is NO gracefully degrading consensus for CRASH.

Fault-Tolerant Consensus

Arbitrary Failures

Using divide-and-conquer:

- O_1, a $\lceil (t-1)/2 \rceil$-tolerant consensus object
- O_2, a $\lfloor (t-1)/2 \rfloor$-tolerant consensus object
- $10t + 3$ (0-tolerant) consensus objects:
  $A_0[1 \ldots 3t+1], A_1[1 \ldots 3t+1], B[1 \ldots 4t+1]$

Fault-Tolerant Consensus
continued

Arbitrary Failures

Efficient t-tolerant self-implementation of `consensus` for arbitrary farilures.

$A_0[1 \ldots 3t+1]$, $A_1[1 \ldots 3t+1]$, $B[1 \ldots 4t+1]$: (0-tolerant) consensus objects, initialized to the uncommitted state

O_1 : $\lceil \frac{t-1}{2} \rceil$-tolerant consensus objects, initialized to the uncommitted state

O_2 : $\lfloor \frac{t-1}{2} \rfloor$-tolerant consensus objects, initialized to the uncommitted state

Procedure `Propose`$(p, v_p, \mathcal{O})$

$count_p[0..1]$, $\text{WitnessCount}_p[0..1]$, $belief_p, ans1_p, ans2_p, v'_p, i, w$: `integer` local to p

1 $count_p$[0..1], WitnessCount_p[0..1] := (0,0)

2 **Phase** 1: **for** $i := 1$ to $3t+1$
3 $w := \texttt{f-propose}(p, v_p, A_{v_p}[i])$
4 **if** $w = v_p$ **then** $count_p[v_p] := count_p[v_p]+1$

5 **Phase** 2: $ans1_p := \texttt{f-propose}(p, v_p, O_1)$

6 **Phase** 3: **for** $i := 1$ to $4t+1$
7 $w := \texttt{f-propose}(p, ans1_p, B[i])$
8 $\text{WitnessCount}_p[w] := \text{WitnessCount}_p[w]+1$

9 **Phase** 4: **for** $i := 1$ to $3t+1$
10 $w := \texttt{f-propose}(p, v_p, A_{\overline{v_p}}[i])$
11 **if** $w = \overline{v_p}$ **then** $count_p[\overline{v_p}] := count_p[\overline{v_p}]+1$

12 **Phase** 5: Choose $belief_p$ s. t. $\text{WitnessCount}_p[belief_p] > \text{WitnessCount}_p[\overline{belief_p}]$
13 **if** $\text{WitnessCount}_p[belief_p] \geq 3t+1$ and $count_p[belief_p] \geq 2t+1$ **then**
14 return($belief_p$)
15 **if** $\text{WitnessCount}_p[belief_p] \geq 2t+1$ and $count_p[belief_p] \geq t+1$ **then**
16 $v'_p := belief_p$
else
17 $v'_p := v_p$
18 $ans2_p := \texttt{propose}(p, v'_p, O_2)$
19 return($ans2_p$)

Fault-Tolerant Consensus

Theorem

The algorithm on the previous page presents a t-tolerant gracefully degrading self-implementation of `consensus` for arbitrary failures of resource complexity $O(t \log t)$.

Adding Reset Capability

Procedure `Reset`$(p, \mathcal{O})$
  i : `integer` local to p
begin
  `reset`(p, O_1)
  `reset`(p, O_2)
  for $i := 1$ to $3t + 1$
    `reset`$(p, A_0[i])$
    `reset`$(p, A_1[i])$
  for $i := 1$ to $4t + 1$
    `reset`$(p, B[i])$
  return(ack)
end

Reset procedure of the t-tolerant self-implementation of `consensus` with `safe-reset` for arbitrary failures.

Register

- *read* and *write* operations
- `unbounded register` $=$ ∞`-valued register`
- `boolean register` $=$ `2-valued register`

$R_1, R_2, \cdots, R_{2t+1}$: 1-reader 1-writer safe registers, initialized to the initial value of the derived register

```
Apply(P_r, read, R)
  val, i : integers, local to P_r
  S : multi-set of integers, local to P_r
begin
  S := ∅
  for i := 1 to 2t + 1
    val := apply(P_r, read, R_i)
    S := S ∪ {val}
  return mode(S)
end
```

```
Apply(P_w, write v, R)
  i : integer, local to P_w
begin
  for i := 1 to 2t + 1
    apply(P_w, write v, R_i)
  return ack
end
```

t-tolerant self-implementation of `1-reader 1-writer safe register` for arbitrary failures.

Fault-Tolerant Register

Theorem

`register` has a t-tolerant gracefully degrading self-implementation for arbitrary failures.

Universality Results

Theorem — Herlihy (1991)

For all types T, there is a k such that T has a (0-tolerant) k-bounded implementation from $\{$`consensus with safe-reset`, `unbounded register`$\}$.

Theorem — Plotkin (1989)

For all *finite* types T, there is a k such that T has a (0-tolerant) k-bounded implementation from $\{$`consensus with safe-reset`, `boolean register`$\}$.

Fault-Tolerant Impl. of Generic Types

for x **in** {`boolean`, `unbounded`}

Corollary

Let T be any (**if** $x =$ `boolean` $\rightarrow$ finite) type.

- T has a t-tolerant gracefully degrading implementation from {`consensus with safe-reset`, x `register`} for arbitrary failures.

- If each of `consensus with safe-reset` and x `register` has a *0-tolerant* gracefully degrading implementation from T for arbitrary failures, then T has a *t-tolerant* gracefully degrading self-implementation for arbitrary failures.

Fault-Tolerant Impl. of Common Types

Corollary

`compare&swap`, `move`, and `m-m swap` have t-tolerant self-implementations for arbitrary failures.

Corollary

`queue`, `stack`, `test&set`, and `fetch&add` have t-tolerant self-implementations for arbitrary failures. These implementations are for two processes.

Tolerating Non-responsive Failures

- *parallel* access to objects
- `consensus`, *impossible!*
- `register`
- Universal *randomized* implementation

Consensus Revisited

Theorem

There is *no* 1-tolerant implementation of `consensus`, even for two processes, for NR-crash(or for *unfairness to a known process*).

Consensus Revisited

Theorem

There is *no* 1-tolerant implementation of `consensus`, even for two processes, for NR-crash(or for *unfairness to a known process*).

Theorem — Loui, Abu-Amara, Dolev, Dwork, and Stockmeyer

The consensus problem for n processes has *no* solution if processes may communicate only via registers and at most one process may crash.

Consensus Revisited
continued

Corollary

If a type T implements `consensus` for two processes, then T has *no* 1-tolerant implementation, for two processes, for NR-crash or for unfairness to a known process.

Consensus Revisited
continued

Corollary

If a type T implements `consensus` for two processes, then T has *no* 1-tolerant implementation, for two processes, for NR-crash or for unfairness to a known process.

Corollary — Common Types

None of the following types has a 1-tolerant implementation, for two processes, for NR-crash or for unfairness to a known process: `compare&swap`, `fetch&add`, `move`, `queue`, `stack`, `sticky-bit`, `m-m swap`, and `test&set`.

Fault-Tolerant Register Revisited

Present a t-tolerant self-implementation of `1-reader`

`1-writer safe register`

Fault-Tolerant Register Revisited

Present a t-tolerant self-implementation of `1-reader 1-writer safe register`

It uses $5t + 1$ base registers. To apply an operation, it applies the same operation on the base registers asynchronously and waits for $4t + 1$ responses.

The mode of the responses is returned as the response.

Fault-Tolerant Register

NR-ArbitraryFailures

t-tolerant self-implementation of `1-reader 1-wrtier safe register` for NR-arbitraryfailures.

$R_1, R_2, \cdots, R_{5t+1}$: 1-reader 1-writer safe registers, initialized to the initial value of the derived register

Pending$_r$: set, local to the reader process P_r, initialized to $\emptyset$
Pending$_w$: set, local to the writer process P_w, initialized to $\emptyset$

```
Apply(P_r, read, R)
  Invoked_r: set, local to P_r
  Responses_r: multi-set, local to P_r
  val, i : integers, local to P_r
begin
  Invoked_r := ∅
  Responses_r := ∅
  i := 0
  Loop
    i := (i mod 5t + 1) + 1
    if R_i ∈ Pending_r then
      Check if R_i responded
      if (yes) then
        Pending_r := Pending_r − {R_i}
        Let val be the response
        if R_i ∈ Invoked_r then
          Responses_r := Responses_r ∪ {val}
    if (R_i ∉ Pending_r) ∧ (R_i ∉ Invoked_r) then
      Invoke read on R_i
      Invoked_r := Invoked_r ∪ {R_i}
      Pending_r := Pending_r ∪ {R_i}
  Until |Responses_r| = 4t + 1
  return mode(Responses_r)
end
```

```
Apply(P_w, write v, R)
  Invoked_w: set, local to P_w
  Responses_w: multi-set, local to P_w
  val, i : integers, local to P_w
begin
  Invoked_w := ∅
  Responses_w := ∅
  i := 0
  Loop
    i := (i mod 5t + 1) + 1
    if R_i ∈ Pending_w then
      Check if R_i responded
      if (yes) then
        Pending_w := Pending_w − {R_i}
        Let val be the response
        if R_i ∈ Invoked_w then
          Responses_w := Responses_w ∪ {val}
    if (R_i ∉ Pending_w) ∧ (R_i ∉ Invoked_w) then
      Invoke write v on R_i
      Invoked_w := Invoked_w ∪ {R_i}
      Pending_w := Pending_w ∪ {R_i}
  Until |Responses_w| = 4t + 1
  return ack
end
```

Fault-Tolerant Register

Theorem

`register` has a t-tolerant self-implementation for NR-arbitrary failures.

Randomized Fault-Tolerant Implementations of Generic Types

- Processes have access to *fair coins*
- Finite *expected* complexity
- Every type has a *randomized* implementation from `register`! [Herlihy 1991]

Randomized Fault-Tolerant Implementations of Generic Types
continued

Theorem

Every finite (infinite) type has a t-tolerant randomized implementation from `boolean (unbounded) register` for NR-arbitrary failures.

Randomized Fault-Tolerant Implementations of Generic Types
continued

Theorem

Every finite (infinite) type has a t-tolerant randomized implementation from `boolean (unbounded) register` for NR-arbitrary failures.

Corollary — Common Types

Each of `test&set`, `compare&swap`, `move`, `m-m swap`, `fetch&add`, `queue`, and `stack` has a t-tolerant randomized self-implementation even for NR-arbitrary failures.

Tired?

Graceful Degradation for Crash

Theorem

There is *no* 1-tolerant gracefully degrading implementation of any order-sensitive type for crash.

Graceful Degradation for Omission

Theorem

Every type has a t-tolerant gracefully degrading implementation from every universal set of types for omission.

Graceful Degradation for Omission

Proof

1. Every 0-tolerant implementation can be transformed into a 0-tolerant implementation which is gracefully degrading for omission.

Graceful Degradation for Omission

Proof

1. Every 0-tolerant implementation can be transformed into a 0-tolerant implementation which is gracefully degrading for omission.
2. `register` has a t-tolerant gracefully degrading self-implementation for omission.

Graceful Degradation for Omission

Proof

1. Every 0-tolerant implementation can be transformed into a 0-tolerant implementation which is gracefully degrading for omission.
2. `register` has a t-tolerant gracefully degrading self-implementation for omission.
3. `consensus with safe-reset` has a t-tolerant gracefully degrading implementation from {`consensus with safe-reset`, `register`} for omission.

Graceful Degradation for Register

Again, we present a 1-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register`.

Combining this with other results like before, we obtain a 1-tolerant gracefully degrading self-implementation of `register`.

R_1, R_2, R_3, R_4: `1-reader 1-writer safe register`, initialized to the same value as the initial value of the derived register
$FAILED_w$: set, local to the writer process P_w, initialized to $\emptyset$
$FAILED_r$: set, local to the reader process P_r, initialized to $\emptyset$
ValuesRead: multi-set, local to P_r

`Apply`$(P_r, read, \mathcal{R})$

ValuesRead := $\emptyset$
for i := 1 to 4
  if $R_i \notin FAILED_r$ **then**
    $resp$:= `read`(P_r, R_i)
    if $resp = \perp$ **then**
      $FAILED_r := FAILED_r \cup \{R_i\}$
    else *ValuesRead* := *ValuesRead* $\cup \{resp\}$
if $|FAILED_r| \geq 2$ **then**
  return $\perp$
else return $mode$(*ValuesRead*)

`Apply`$(P_w, write\ v, \mathcal{R})$

for i := 1 to 4
  if $R_i \notin FAILED_w$ **then**
    $resp$:= `write`(P_w, v, R_i)
    if $resp = \perp$ **then**
      $FAILED_w := FAILED_w \cup \{R_i\}$
if $|FAILED_w| \geq 2$ **then**
  return $\perp$
else return ack

1-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register` for omission

Graceful Degradation for Consensus

We present a *t-tolerant* gracefully degrading implementation of `consensus with safe reset` from {`boolean register`, `0-tolerant consensus with safe reset`}.

$\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_{2t+1}$: t-tolerant gracefully degrading boolean registers, initialized to 0

$O_1, O_2, \ldots, O_{2t+1}$: (0-tolerant) consensus-with-safe-reset objects

```
Procedure Propose(P_i, v_i, O)
  V_i[1...2t+1], estimate_i, resp, k,
  set-of-failed: local to P_i
begin
  estimate_i := v_i
  set-of-failed := ∅
  for k := 1 to 2t+1
    resp := Read(P_i, R_k)
    if resp = ⊥ then
      return ⊥
    else if resp = 1 then
      set-of-failed := set-of-failed ∪ {O_k}
  for k := 1 to 2t+1
    if O_k ∈ set-of-failed then
      V_i[k] := ⊥
    else
      resp := propose(P_i, estimate_i, O_k)
      if resp = ⊥ then
        resp := Write(P_i, 1, R_k)
        if resp = ⊥ then
          return ⊥
        else if resp ≠ estimate_i then
          estimate_i := resp
          V_i[1...(k-1)] := (⊥, ⊥, ..., ⊥)
  if V_i has more than t ⊥'s then
    return ⊥
  else return estimate_i
end
```

```
Procedure Reset(P_i, O)
  set-of-failed, resp, k: local to P_i
begin
  set-of-failed := ∅
  for k := 1 to 2t+1
    resp := Read(P_i, R_k)
    if resp = ⊥ then
      return ⊥
    else if resp = 1 then
      set-of-failed := set-of-failed ∪ {O_k}
  for k := 1 to 2t+1
    if O_k ∉ set-of-failed then
      resp := reset(P_i, O_k)
      if resp = ⊥ then
        resp := Write(P_i, 1, R_k)
        if resp = ⊥ then
          return ⊥
  return ack
end
```

Q?