Constructing Reliable Objects From Unreliable Components

Our Goal

- Given n components (disks, registers, etc) prone to Byzantine failures, construct a reliable object (disk, register, etc), which can tolerate up to t ($3t < n$) failures among its components

Some Definitions

Safe SWMR Register

- Every complete read operation that does not overlap any write operation returns the register's value (i.e., the value of the last write)
- Otherwise the read operation returns an arbitrary value

Some Definitions

Regular SWMR Register

- Same as safe SWMR register, with one difference:
- When write operations overlap the read operation, the latter returns one of the values being written, or the last value of the register (before the overlapping writes)

Some Definitions

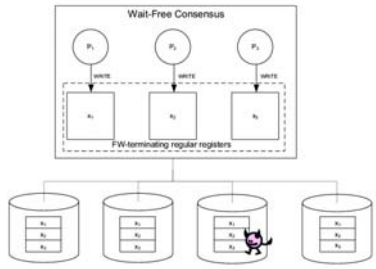
Wait Free Register

- All operations complete

Finite Writes Terminating Register

- All write operations complete
- Read operations complete, provided that there are finite number of write operations

Sufficiency of FW-Termination



FW-Terminating Regular Register Emulation

- We are given n SWMR FW-Terminating regular registers, up to t of which may fail arbitrarily ($3t < n$)
- We want to emulate a reliable (t-resilient) SWMR FW-Terminating regular register

<TimeStamp, Value> Pairs

- We have a single writer, hence we can use time stamps
- From now on, all records in the registers will be <TimeStamp, Value> pairs (TSVals for short)
- Two records in each register: *pw* and *w* (because we will have a two-phase write op)

Intuition

- Two phases: pre-write and write, each writing to $n - t$ registers
- $t + 1$ *pw* records for a value testify that a write for that value has actually began
- $2t + 1$ lower-timestamped *w* records different from a given value means that the value's write phase did not complete

Intuition

- Hence the read operation looks for the value with the highest timestamp, among those written to at least $t + 1$ registers without being "testified against" by more than $2t$ registers

Note: "testimony" above means having a **lower**-timestamped record.

Going Over The Code…

In the CHECK subroutine of the WRITE operation we use the following notation for the low-level operations:

- *enabled[i]* means the register has not been contacted on this round yet
- *pending[i]* means the register has been contacted, but it hasn't responded yet

Going Over The Code…

CHECK:
for $1 \leq i \leq n$
if $(enabled[i] \wedge \neg pending[i])$ **then**
$\langle enabled[i], pending[i] \rangle \leftarrow \langle$false,true$\rangle$;
INVOKE write$(x_i, \langle pw, w \rangle)$;
if (x_i RESPONDED) **then**
$pending[i] \leftarrow$ false;

Going Over The Code…

```
/* Write phase */
w ← ⟨ts, v⟩;
for 1 ≤ i ≤ n, enabled[i] ← true;
repeat
    CHECK;
until |{i : ¬enabled[i] ∧ ¬pending[i]}| ≥ n − t;

return ack;
```

Going Over The Code…

In the READ operation we also define:

- *pw[i], w[i], tmpPW[i]* and *tmpW[i]* contain the values read from the corresponding register
- *old[i]* specifies whether the pending read was invoked in a previous execution of the READ operation

Going Over The Code…

```
CHECK:
    for 1 ≤ i ≤ n
        if (enabled[i] ∧ ¬pending[i]) then
            ⟨enabled[i], pending[i]⟩ ← ⟨false,true⟩;
            INVOKE ⟨tmpPW[i], tmpW[i]⟩ ← read(x_i);
        if (x_i RESPONDED) then
            if (¬old[i]) then
                pw[i] ← tmpPW[i];
                w[i] ← tmpW[i];
            pending[i] ← false;
            old[i] ← false;
```

Going Over The Code…

We also define the following predicates:

$$
\begin{aligned}
\text{readFrom}(c,i) &= c \in TSVals \wedge (pw[i] = c \vee w[i] = c)\\
\text{safe}(c) &= |\{\, i \;:\; \text{readFrom}(c,i)\,\}| \geq t+1\}\\
\text{invalid}(c) &= |\{\, i \;:\; \exists c' \;:\; \text{readFrom}(c',i) \wedge c'.ts < c.ts \vee\\
&\quad \vee (c'.ts = c.ts \;\wedge\; c'.v \neq c.v)\}| \geq 2t+1\\
\\
\text{highestValue}(c) &= \forall c' \forall i \;:\; (\text{readFrom}(c',i) \wedge\\
&\quad \wedge\, c'.ts \geq c.ts \;\wedge\; c' \neq c) \rightarrow \text{invalid}(c')
\end{aligned}
$$

Going Over The Code…

```
READ():
1:  for 1 ≤ i ≤ n, if(pending[i]) then old[i] ← true;
2:  for 1 ≤ i ≤ n, pw[i], w[i] ← ⊥;
3:  repeat
4:      for 1 ≤ i ≤ n, enabled[i] ← true;
5:      repeat
6:          CHECK;
7:      until |{i : ¬enabled[i] ∧ ¬pending[i]}| ≥ n − t;
8:      C ← {c : safe(c) ∧ highestValid(c)};
9:  until (C ≠ ∅);
10: return c.val : c ∈ C;
```

Proving Regularity

- The returned value *c* is **safe**, hence there was indeed a WRITE invocation for it
- Assume our construction is not regular: then *c* must have been written **before** the last completed WRITE operation (let's denote it by WRITE(*v*))

Proving Regularity

- After this last WRITE(v) no more than t registers remained with a timestamp less than that of v
- Hence, v cannot be marked as **invalid**, because no more than $2t$ registers would report a smaller timestamp

Proving Regularity

- We assumed c has a smaller timestamp than v, hence c would not pass the **highestValid** test because of v
- Hence, c could not get to the set C and thus cannot be returned by the READ operation => contradiction

Proving FW-Termination

- WRITE operations always complete (as $n - t$ registers always respond)
- Similarly READ operations never get stuck in the inner (lines 5-7) loop, waiting for responses.

Proving FW-Termination

- If we have finite number of writes, then there exists a time T when all writes have completed (and all correct registers have returned, as they satisfy FW-Termination)
- Let's consider an execution of the outer loop of a READ operation at time T

Proving FW-Termination

- Consider the last WRITE operation, which has completed its pre-write phase (let's denote it WRITE(v))
- We have two cases. In case WRITE(v) did not complete its write (second) phase, then the Writer must have crashed

Proving FW-Termination

- In that case the v holds at least $t+1$ pw records of correct registers, hence it is **safe**
- Also, none of the $n-t$ (which is at least $2t+1$) correct registers holds a timestamp greater than that of v, hence v will not be judged **invalid**, while all other records with higher timestamps will (as they could exist only in faulty registers)

Proving FW-Termination

- In the second case WRITE(v) has completed its write (second) phase and since no later write has completed its pre-write phase, v holds the w records of all n-t correct registers, which are at least $2t$+1
- This makes all higher time-stamped records **invalid** and keeps v from invalidation too

Proving FW-Termination

- In both cases the READ operation, executed at time T returns v, hence our algorithm satisfies FW-Termination.

Wait-free Safe Register Emulation

- We are given n SWMR wait-free safe registers, up to t of which may fail arbitrarily ($3t < n$)
- We want to emulate a reliable (t-resilient) SWMR wait-free safe register
- Note: we **have** to bound the number of read rounds in order to be wait-free

Intuition

- WRITE operation is the same as in the previous algorithm
- The READ operation for $t = 1$ returns after 2 read rounds
- We are forced to return a particular value only if a WRITE operation does not overlap with our READ operation

Intuition

- Hence it makes sense only to return values that have been written during the last WRITE. Such values certainly occupy at least 2 correct registers' w records.
- Hence if 3 registers respond without a given value in their w records, the value is out of consideration

Intuition

- Also, if the highest time-stamped value v still in consideration has at least two records that contain v or a higher time-stamped value, then v is safe to return (as this either guarantees that v was indeed written in a correct register, or it shows that there's an overlapping WRITE and thus we're free to return an arbitrary value, including v)

Intuition

- Hence for each remaining return value candidate *c*, we have to close the gap and either find 3 registers that do not have a record for *c*, or find 2 registers that do.
- We can do that in just one additional read round by considering two cases:

Intuition

- If *c* was reported by a faulty register, then there is a correct register that hasn't responded in the first round. Whenever its response arrives we'll be able to close the gap and return.

Intuition

- If *c* was reported by a correct register (remember: in its *w* record) this means that the pre-writing of *c* has been completed. Thus on the next round there will be at least two correct registers reporting either *c* or a higher time-stamped value in their *pw* record, which is exactly what we needed.

Going Over The Code…

- For each TSVal *v* we denote the set of registers in whose *w* records it occurs by *ReadW(v)*. Similarly we use *ReadPW(v)* for the *pw* records and *prevReadW(v)* for the *ReadW(v)* set on the previous round.
- This implies some changes in the CHECK subroutine

Going Over The Code…

CHECK:
  for $1 \leq i \leq n$
    if $(enabled[i] \wedge \neg pending[i])$ **then**
      $\langle enabled[i], pending[i] \rangle \leftarrow \langle$false,true$\rangle$;
      INVOKE $\langle pw[i], w[i] \rangle \leftarrow \text{read}(x_i)$;
    if $(x_i$ RESPONDED$)$ **then**
      if $(\neg old[i])$ **then**
        $ReadPW(pw[i]) \leftarrow ReadPW(pw[i]) \cup \{i\}$;
        $ReadW(w[i]) \leftarrow ReadW(w[i]) \cup \{i\}$;
      $pending[i] \leftarrow$ false;
      $old[i] \leftarrow$ false;

Going Over The Code…

- We also add a set *C* containing the return value candidates and we define the following predicates:

$$
\begin{array}{lcl}
\text{Responded} & = & \{i \ : \ \exists \langle w, i \rangle \in ReadW\} \\
\text{highCand}(\langle ts, v \rangle) & = & \langle ts, v \rangle \in C \wedge \\
 & & \quad \wedge \ (ts = max\{ts' \ : \ \langle ts', v' \rangle \in C\}) \\
\text{safe}(c) & = & |ReadW(c) \cup ReadPW(c) \cup \\
 & & \cup_{c', ts > cts}(ReadW(c') \cup ReadPW(c'))| \geq t + 1
\end{array}
$$

Going Over The Code…

```
6:  until |{i : ¬enabled[i] ∧ ¬pending[i]}| ≥ n − t;
7:  C ← {w[i] : |Responded \ ReadW(w[i])| < 2t + 1};
    /* Rounds 2 ... */
8:  while (C ≠ ∅ ∧ (¬∃c ∈ C : highCand(c) ∧ safe(c))) do
9:      prevReadW ← ReadW;
10:     for 1 ≤ i ≤ n, enabled[i] ← true;
11:     repeat
12:         CHECK;
13:     until |{i : ¬enabled[i] ∧ ¬pending[i]}| ≥ n − t ∧
                ∀c ∈ C : (safe(c) ∨ |Responded \ prevReadW(c)| ≥ n − t);
14:     C ← {c ∈ C : |Responded \ ReadW(c)| < 2t + 1};
15: if (C ≠ ∅) then
16:     return c.val : highCand(c) ∧ safe(c);
17: return v0;
```

Proving Safety

- If there are concurrent writes with the READ, all return values are correct.
- Otherwise let's denote the value of the last WRITE with *v*. In this case we have to guarantee that we return no other value than *v*.

Proving Safety

- Since *v*'s write has completed, it is written in the *w* record of at least *n-t* registers, at least *n*-2*t* of which are correct. Hence, *v* becomes a candidate right after the first read round and never ceases to be.
- Also, no other candidate can get returned, as no correct register would return a value with a greater timestamp than *v*

Proving Wait-freedom

- Since at least *n-t* registers are correct, our algorithm always exists the loop between lines 4 and 6 and the first condition on line 13 is always eventually satisfied.
- All we have to do is prove that the second condition on line 13 eventually gets satisfied. In order to do that we consider two cases.

Proving Wait-freedom

- Case 1: At least one register in *PrevW(c)* is correct. This means the pre-write phase of WRITE(*c*) has been completed, which means that at least *t*+1 correct registers have had *c* in their *pw* record => they'll eventually respond with *c* or a higher time-stamped value, which is guaranteed to make *c* **safe**

Proving Wait-freedom

- Case 2: All registers in *PrevW(c)* are faulty, hence at least *n-t* other registers will respond, which will meet the second *OR* clause of the condition we want to satisfy.
- Hence the loop on lines 11-13 always exits. Now let's consider the loop on lines 8-14.

Proving Wait-freedom

- For every candidate *c* after each round (iteration) of the 8-14 loop there are three cases:
- Case 1: *c* becomes **safe** hence it doesn't obstruct the condition on line 8 anymore.
- Case 2: None of the *n-t* register mentions *c*, hence *c* gets excluded from *C*, as $n\text{-}t > 2t$

Proving Wait-freedom

- Case 3: Some of the *n-t* registers mentions *c* and hence *ReadW(c)* grows.

  This case can be exercised at most $t+1$ times, as when the size of *Read(c)* grows to $t+1$ it becomes **safe**. Hence there are at most $t+1$ iterations (read rounds) of the 8-14 loop

Early Stopping

- In fact, the algorithm is early stopping in the sense that it takes $\min(t+1, f+2)$ read rounds
- Intuition: Only case 3 (previous slide) leads to additional rounds and it needs a new faulty register to be added to *ReadW(c)* at each of those rounds.

Lower Bound on Write Rounds

- We consider constructing a SWSR FW-Terminating binary safe register (weakest possible) out of $n < 4t+1$ base registers
- Obviously we need at least one round of writes to our base registers, as otherwise the next READ would not be able to distinguish whether the WRITE occurred or not.

Lower Bound on Write Rounds

- Now we'll prove the need for at least two write rounds on some base register.
- Assume the contrary: i.e., that one round of writes is enough. Then divide the base registers into four groups: S_1, S_2, S_3 and S_4, each of which consisting of no more than *t* base registers. Assume w.l.o.g. that the writer writes to them in order S_1, S_2, S_3, S_4

Lower Bound on Write Rounds

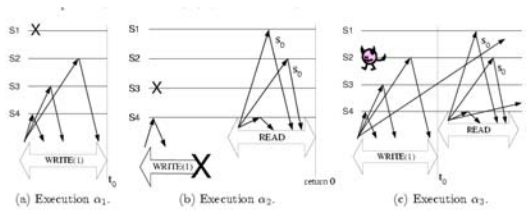


Figure 8: Illustrating the lower bound on WRITE emulations.

Lower Bound on Read Rounds

- There is also a lower bound on the number of read rounds required for $n = 3t+1$, which is $\min(t+1, f+2)$, equal to what our algorithm takes.
- Intuition: We need one more round to resolve the ambiguity arising from each faulty register.

Lower Bound on Read Rounds

- Both the algorithm and the lower bound generalize to $\min([t/k]+1, [f/k]+2)$, where $n = 3t + k$
- Proof left as a mental exercise for (optional) homework ☺

Thank you!