Constructing Reliable Registers From Unreliable Byzantine Components

Seth Gilbert

Review

Space of registers:

– Dimension 1: binary vs. multivalued

– Dimension 2: safe vs. regular vs. atomic

– Dimension 3: SRSW vs. MRSW vs. MRMW

Review

Transformations:

– binary SRSW safe --> binary MRSW safe

– binary MRSW safe --> binary MRSW regular

– binary MRSW regular --> multival MRSW regular

– multival SRSW regular --> multival SRSW atomic

– multival SRSW atomic --> multival MRSW atomic

– multival MRSW atomic --> multival MRSW atomic

Review

Space of registers:

– Dimension 1: *number of values*
  binary vs. multivalued

– Dimension 2: *consistency*
  safe vs. regular vs. atomic

– Dimension 3: *# readers, # writers*
  SRSW vs. MRSW vs. MRMW

– Dimension 4: *modes of failure*
  none vs. responsive vs. non-responsive

Review

- Algorithm 1: implement SWMR register out of t+1 SWMR responsive failure-prone registers.
- Algorithm 2: implement SWSR register out of 2t+1 SWSR non-responsive fault-prone registers.

Today

- New mode of failure: **NR-Arbitrary**
  - NR = non-responsive
    - A failed register may or may not respond to a read or write request.
  - Arbitrary = Byzantine
    - A failed register can return *any* value: a real value, a fake value.
- We think of the register as controlled by a malicious adversary.

Fault-Prone Registers

- Example: Storage Area Network (SAN)
  - Networked storage available for storing large amount of data reliably.
  - SAN consists of a large array of hard-drives.
  - In order to store and retrieve data, servers send requests to the SAN.
  - When a hard-drive fails, it may crash, or it may return invalid (corrupted) data.
- See IBM TotalStorage SAN256B

Fault-Prone Registers

- Basic Model:
  - Registers x1, ..., xn
  - When a process wants to read a register xj, it does:
    - INVOKE *read/write* xj
  - If the register is correct, it does:
    - (for a write:) RESPOND xj
    - (for a read:) RESPOND xj v
  - If the register is faulty, it may or may not respond, and the response may be bad.

Our Goal

- Given:
  - n components (registers) prone to NR-Arbitrary failures
  - at most $t < n/3$ failures
- Construct:
  - a reliable, fault-free object (register)

Recall

- Safe Register:
  - Every complete read operation that does not overlap any write operation returns the value of the last write operation. Otherwise, the read operation returns an arbitrary value.
- Regular Register:
  - Every complete read operation returns the value of the last preceding write operation or a current write operation.

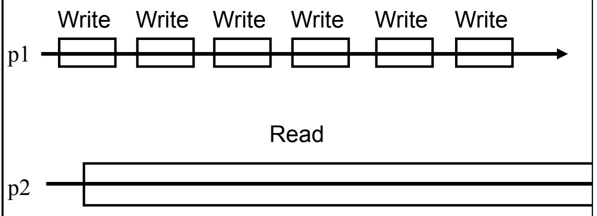
Termination

- Wait-freedom:
  - Every operation eventually terminates.

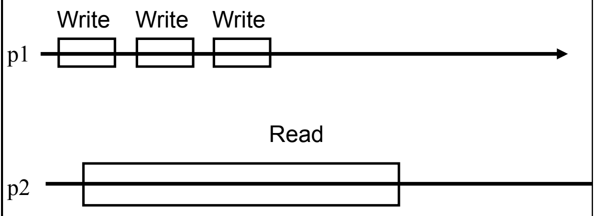
Termination

- Wait-freedom:
  - Every operation eventually terminates.
- Finite-Writes (FW)-Termination
  - All write operations complete.
  - In every execution with only a finite number of write operations, every read operation terminates.

FW-Termination



FW-Termination



Basic Algorithm

- Assume 5t+1 registers.
- At most t are NR-Arbitrary faulty.
- Goal: SWSR safe register

Basic Algorithm

- Writer's state:
  - ts : timestamp
  - val : value
- Write(v) :
  1. ts <-- ts+1
  2. invoke *write* (ts, v) on all registers
  3. on receiving (4t+1) responses, return *ack*.

Basic Algorithm

- Read()
  1. invoke *read* on all registers
  2. On receiving (4t+1) responses:
     a) If any (ts,val) pair is returned by at least (2t+1) of registers, then return the val with the largest timestamp (that is returned by at least 2t+1 registers).
     b) Otherwise, return default value v0

Basic Algorithm

- Termination: at most t faulty, hence always get enough responses.

Basic Algorithm

- Safety:
  – Consider a read operation that does not overlap with any write operations.
  – Let (ts, val) be the last thing written prior to the Read() operation.
  – Write(val) received responses from 4t+1 registers.
  – Thus, at least 3t+1 correct registers have (ts,val).

Basic Algorithm

- Safety (cont.):
  – The Read() receives responses from 4t+1 registers.
  – The (4t+1) read-set intersects the (3t+1) write-set in at least 2t+1 registers.
  – Thus, the Read() receives (ts,val).
  – At most t correct processes are not in the write-set.
  – At most t processes are NR-Arbitrary.
  – Hence, at most 2t returns a value *not* (ts, val).
  – Thus, the Read() returns (ts, val) as desired.

Robustness

- We assumed n > 5t??
- Why?
  - Needed a lot of intersection to ensure correct processes win.
- But: we only need n > 3t!

Today

- Two Algorithms:
  - Algorithm 1: Construct a FW-terminating MRSW *regular register* from n SWMR FW-Terminating regular registers of which up to t < n/3 may have NR-Arbitrary failures.
  - Algorithm 2: Construct a wait-free MRSW *safe register* out of n MRSW wait-free safe registers, up to t<n/3 may have NR-Arbitrary failures.

Lower Bounds

- Theorem:
  - It is impossible to implement a safe, wait-free register if $t \geq n/3$.

Lower Bounds

- Theorem:
  - To implement a t-tolerant FW-terminating SWSR binary safe register, a WRITE operation requires at least two consecutive *write* invocations on the same correct base object.
- A Write() requires two rounds!

Lower Bounds

- Theorem:
  - To implement a t-tolerant SWSR safe register when the READ() does not invoke *write* operations, a READ() operation requires at least t+1 rounds of *read* invocations.
- A Read() requires at least t+1 rounds!

Algorithm 1

- Given:
  - n MRSW FW-terminating regular registers x1, x2, …, xn
  - t NR-Arbitrary failures.
- Construct:
  - MRSW FW-terminating regular register

Writer’s Algorithm

- Writer maintains timestamp *ts*.
- With every write, the timestamp is incremented. There is a unique timestamp associated with each value.
- TSVal = [timestamp, value]

Writer’s Algorithm

- Writer’s state:
  - pw: TSVal (pre-write value)
  - w: TSVal (write value)
- Two phase algorithm:
  1. Write new TSVal to pw.
  2. Write new TSVal to w.
- Each phase contacts at least (n-t) registers, at least t+1 of which are correct.

Writer’s Algorithm

```
Registers: x1, x2, …, xn
Perform_Write(pw, w)
    for 1 ≤ i ≤ n do
        if (enabled[i] and not pending[i]) then
            enabled[i] <-- false
            pending[i] <-- true
            INVOKE write(xi, <pw, w>)
        if (xi RESPONDED) then
            pending[i] <-- false
```

Writer’s Algorithm

```
Registers: x1, x2, …, xn
Write(v)
    ts <-- ts+1
    pw <-- [ts, v]
    for 1 ≤ i ≤ n do enabled[i] <-- true
    while |{i : not enabled[i]}| ≥ n-t
        Perform_Write(pw, w)
    w <-- [ts, v]
    for 1 ≤ i ≤ n do enabled[i] <-- true
    while |{i : not enabled[i]}| ≥ n-t
        Perform_Write(pw, w)
```

Reader’s Algorithm

- Repeatedly read (n-t) registers.
- A value is ***safe*** if it is read from at least t+1 registers.
  - At least one register must be correct.
  - Thus, the value was written by *some* write operation.
- Return the *safe* value with the highest timestamp.

Reader’s Algorithm

```
Perform_Read()
    for 1 ≤ i ≤ n do
        if (enabled[i] and not pending[i]) then
            enabled[i] <-- false
            pending[i] <-- true
            old[i] <-- false
            INVOKE read (xi)
        if (xi RESPONDED [a,b]) then
            pending[i] <-- false
            if not old[i] then
                pw[i] <-- a
                w[i] <-- b
            old[i] <-- false
```

Reader's Algorithm

```
Read()
    for $1 \le i \le n$ do old[i] <-- true
    for $1 \le i \le n$ do pw[i] <-- NIL
    for $1 \le i \le n$ do w[i] <-- NIL
    Repeat
        for $1 \le i \le n$ do enabled[i] <-- true
        while $|\{i : \text{not enabled}[i]\}| \ge n-t$
            Perform_Read()
        C <-- {c : safe(c) and highestValid(c) }
    until $C \ne \{\}$
    return c.val : c in C
```

Reader's Algorithm

- *safe* (c) :
  - There exists a set of registers P where:
    - $|P| \ge t+1$
    - For every j in P, either:
      - pw[j] = c
      - w[j] = c
  - Implies that at least t+1 registers responded with value c for either of [a, b].

Reader's Algorithm

- *invalid* (c) :
  - There exists some c' where either:
    - c'.ts < c.ts
    - c'.ts = c.ts **and** c.val ≠ c'.val
  - There exists a set of registers P where:
    - $|P| \ge 2t+1$
    - For every j in P, either:
      - pw[j] = c'
      - w[j] = c'
  - Implies that 2t+1 processes vote against c.

Reader's Algorithm

- *highestValue* (c) :
  - For all c' in pw[*] or w[*] where:
    - c'.ts ≥ c.ts
    - c' ≠ c
  - Then invalid(c').
  - Implies that every larger timestamp is invalid, i.e., is voted against by at least 2t+1 processes.

Regularity

- Lemma 1: If c is *safe*, then there is some Write(v) operation where v = c.val.

- Proof: If c is safe, then it was returned by at least t+1 registers, at most t of which can be failed.

Regularity

- Lemma 2: If some Write(v) operation completes and writes c = [ts, v], then c is not *invalid*.

- Proof: After the Write(v) operation completes, there are at most t registers such that xi ≠ c, i.e., at most t where xi has a TSVal with timestamp < ts. And there are at most t faulty registers. Thus, there are never 2t+1 votes against c.

Regularity

- Theorem 3: The emulated register is regular.
- Proof:
  - Consider a Read() operation that has concurrent write operations.
  - Assume that it returns some value v, associated with some TSVal c = [ts, v].
  - We know that c is *safe,* so by Lemma 1, some Write(v) wrote [ts, v].
  - Let c' = [ts', v'] be the TSVal written by the Write(.) operation immediately preceding the Read() operation.

Regularity

- Proof (continued):
  - Goal: show that (ts ≥ ts'). Assume not.
  - Since the Write(v') of c' completed, there are at least n-t registers with timestamp ≥ ts'.
  - The Read() operation accesses at least 2t+1 registers.
  - Thus there is some correct register that has timestamp ≥ ts' and is accessed by the Read() operation.
  - Let xj be the correct process with the smallest TSVal cj = [tsj, vj] with tsj ≥ ts' that responds to the Read().

Regularity

- Proof (continued):
  - Since tsj ≥ ts' > ts, we know that if cj is not *invalid*, then c = [ts, v] cannot be *highestValid*. So assume cj is invalid.
  - Thus, there are 2t+1 registers that "vote against" cj, i.e., that have timestamps < tsj or have different values and respond to the Read().
  - One of these registers xk must have been correct and also one of the n-t contacted by Write(v'), so:
    - tsk ≤ tsj, since it "votes against" cj
    - tsk ≥ ts', since it is contacted during Write(v')
  - Since cj is the smallest ≥ ts', tsk = tsj.

Regularity

- Proof (continued):
  - Since xk "voted against" cj, but tsk=tsj, we can conclude that vk ≠ vj.
  - But both registers xk and xj are correct.
  - But you can't have two different values associated with the same timestamp! Contradiction!

QED

FW-Terminating

- Theorem 4: The algorithm guarantees FW-termination.
- Proof:
  - Easy to see that writes terminate, since at most t faulty registers.
  - Easy to see that reads never get stuck waiting for responses, since at most t faulty registers.
  - Hard part: show that in a FW execution, eventually there is a c that is *safe* and *highestValid*.

FW-Terminating

- Proof (continued):
  - Assume only a finite number of write operations.
  - Assume some Read() operation never terminates.
  - Let T be the point after which no new Write operations are invoked and after all *write* operations invoked in the low-level registers are complete.
  - Let T' > T, be the point after which every correct register has responded to at least one read invocation after time T.

FW-Terminating

- Proof (continued):
  - Let [ts, v] be the TSVal written in the very last complete Write invocation in the execution.
  - Case 1 : no (incomplete) Write completes the pre-write phase after [ts, v].
    - Then (ts,v) appears in at least t+1 registers w[*], and is safe. And by Lemma 2, (ts, v) is not invalid.
    - And the incomplete write *is* invalid, since the 2t+1 correct nodes "vote against" the incomplete write, since they each have w[*] field ≤ ts.
    - Thus, (ts,v) is in C.

FW-Terminating

- Proof (continued):
  - Let [ts, v] be the TSVal written in the very last complete Write invocation in the execution.
  - Case 2 : some (incomplete) Write completes the pre-write phase after [ts, v] with [ts', v'].
    - Choose largest such (ts', v').
    - Then (ts',v') appears in at least t+1 registers pw[*], and is safe. And by Lemma 2, (ts', v') is not invalid.
    - And any other larger write *is* invalid, since the 2t+1 correct nodes "vote against" the larger write, since they each have w[*] field ≤ ts'.
    - Thus, (ts',v') is in C.

QED

Algorithm 2

- Given:
  - n MRSW wait-free safe registers
  - < t failures
  - t < n/3
- Construct:
  - Wait-free safe register
  - (Bounded number of iterations for each op.)

Algorithm 2

- Write(v) :
  - Same as Algorithm 1.
  - Two phase operation:
    1. Increment timestamp ts := ts+1
    2. Prewrite pw = [ts, v] to n-t registers
    3. Write w = [ts, v] to n-t registers

Reader’s Algorithm

- Repeatedly read registers:
  - If 2t+1 registers reject a value (i.e., return some other value), then we continue.
  - Otherwise, we choose the value with the highest timestamp that is *safe*.

Reader’s Algorithm

- Variables:
  - ReadW(v) : set of registers j that returned v for w[j]
  - ReadPW(v) : set of registers j that returned v for pw[j]
  - prevReadW(v) : snapshot of ReadW at the beginning of the current iteration.
  - Responded : set of processes that have responded at some point during the Read() operation.
  - C : set of candidate values

Reader’s Algorithm

```
Perform_Read()
    **for** 1 ≤ i ≤ n **do**
        **if** (enabled[i] **and** not pending[i]) **then**
            enabled[i] <-- false
            pending[i] <-- true
            old[i] <-- false
            INVOKE *read* (xi)
        **if** (xi RESPONDED [a,b]) **then**
            pending[i] <-- false
            **if** not old[i] **then**
                ReadPW(a) <-- ReadPW(a) \/ {i}
                ReadW(b) <-- ReadW(b) \/ {j}
            old[i] <-- false
```

Reader’s Algorithm

```
Read() (Part 1)
    **for** 1 ≤ i ≤ n **do** old[i] <-- true
    **for all** v : ReadW(v), ReadPW(v) <-- NIL
    %% Round 1 %%
    **for** 1 ≤ i ≤ n **do** enabled[i] <-- true
    **Repeat**
            *Perform_Read()*
    **until** |{i : not (enabled[i] and pending[i])}| ≥ n-t
    C <-- {v : |Responded – ReadW(v)| < 2t+1}
```

Reader's Algorithm

```
Read() (Part 2)
    %% Rounds 2, … %%
    while C ≠ 0 and there is no c in C such that:
        highCand(c) and safe(c)
    do
        for 1 ≤ i ≤ n do enabled[i] <-- true
        prevReadW <-- ReadW
        Repeat   Perform_Read()
        until |{i : not (enabled[i] and pending[i])}| ≥ n-t
            and for all c in C : either safe(c) or
            |Responded – prevReadW(c)| ≥ n-t
        C <-- {v in C : |Responded – ReadW(v)| < 2t+1}
```

Reader's Algorithm

```
Read() (Part 3)
    %% Return value %%
    if C not empty then
        return c.val : highCand(c) and safe(c)
    else
        return v0
```

Reader's Algorithm

- Definitions:
  - *highCand*(c) : if c = [ts, v], then every candidate c' in C has a timestamp that is ≤ ts
  - *safe*(c) : at least t+1 registers have returned a candidate value *equal* to c, or with a timestamp > c.ts.

Note: timestamps larger than c.ts are okay, since we only care that the register is safe, and a larger timestamp may indicate a concurernt write.

Safety

- Theorem: The register is safe.
- Proof:
  - Let R be a read invocation, and assume that there are no concurrent Write(.) ops.
  - Let [ts, v] be the TSVal written by the immediately preceding write operation.
  - Throughout R :
    1. At least t+1 correct registers have [ts, v].
    2. At most 2t registers respond without [ts, v]: t that are uninformed and t that are failed.
  - Thus, Responded – ReadW < 2t+1, so [ts, v] in C, and never excluded later.

Safety

- Theorem: The register is safe.
- Proof:
  - Need to show that no c'=[ts', v'] can be *highCand* and *safe*.
  - Assume c' is *highCand*, i.e., ts' > ts. There are at most t registers that can returns c', or any timestamp >ts, so c' is not *safe*.
  - So, we conclude that the value returned is v.

Wait-freedom

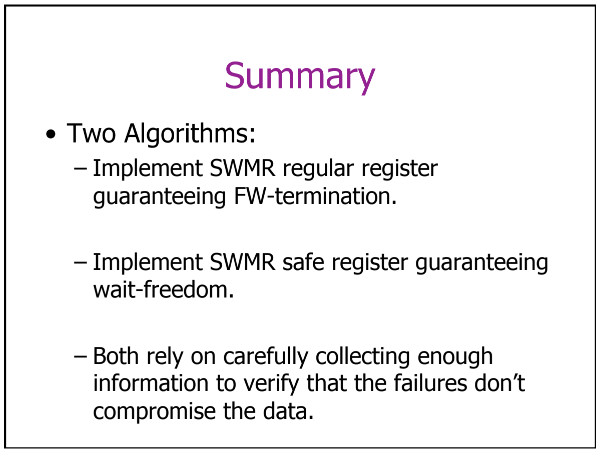
- Theorem: Algorithm 2 is wait-free.
- Proof:
  - Clearly, every write operation completes.
  - Consider a read operation R.
  - We show:
    - The set C is updated at most t+1 times.
    - Each time C is updated, each candidate c gains at least one *supporter*.
    - Thus, at the end, either C is empty or each candidate has t +1 supporters and the algorithm terminates.

Wait-freedom

- Proof (continued):
  - First, it is clear that R is not blocked by waiting for n-t responses.
  - It is also not blocked by waiting for:
    - for all c in C : safe(c) or |Responded – prevReadW(c)| ≥ n-t
    - Fix a c.
    - We know that prevReadW(c) is not empty.
    - If at least one j in prevReadW(c) is correct, then we know that c was pre-written to t+1 correct registers. These registers hold timestamps that are strictly increasing, so eventually c is *safe*.
    - If all the registers in prevReadW(c) are faulty, then n-t correct processes did are not in prevReadW(c). Eventually, every correct process responds, so there are n-t processes in Respnded and not in prevReadW(c).

Wait-freedom

- Proof (continued):
  - Now consider each iteration of the **while** loop:
    while C ≠ 0 and there is no c in C such that *highCand*(c) and *safe*(c)
  - If C is empty, then done.
  - Fix some c in C.
  - At the end of the Perform_Read loop, either *safe*(c) or at least n-t new Respondents not in prevReadW are found.
  - If none of the n-t new respondents have candidate c, then c is removed from C (since n-t voted against it).
  - Thus, at least 1 new respondents returns c, and thus ReadW is bigger than prevReadW.
  - After t+1 iterations, either c is removed from C, or c is *safe*.
  - Thus, *highCand*(c) is *safe*.



Summary

- Two Algorithms:
  - Implement SWMR regular register guaranteeing FW-termination.
  - Implement SWMR safe register guaranteeing wait-freedom.
  - Both rely on carefully collecting enough information to verify that the failures don't compromise the data.