182.703: Problems in Distributed Computing
(Part 2)
WS 2021

Ulrich Schmid
Institute of Computer Engineering, TU Vienna
Embedded Computing Systems Group E191-02
s@ecs.tuwien.ac.at

TU Informatics
Content (Part 2)

- Advanced Topics in Distributed Algorithms
  - Randomization
  - Self-Stabilization
  - Self-Stabilization in VLSI Circuits
Randomization
Randomization (I)

- Allow algorithms to toss a (biased) coin/dice, and perform different state transitions based on its outcome.

- Need to separate variabilities in executions:
  - Variabilities caused by adversary: Message delivery, processor scheduling, failures (subject to admissibility conditions)
  - Variabilities caused by probabilistic choices of the processes
Randomization (II)

- An execution $E = \text{exec}(A, C_0, R)$ of a specific algorithm is determined by
  - the adversary $A$
  - an initial configuration $C_0$
  - a sequence of random numbers $R$ obtained by the processes in $E$

- The adversary maps every execution prefix $E(t)$ to sets of extensions $E'(t')$ of $E(t)$, typically constrained by
  - what information it can actually observe in $E(t)$
  - degree of clairvoyance regarding future random choices
  - how much computational power it has
Randomization (III)

• Common adversaries:
  – **Oblivious** ($C_0 \rightarrow E(\infty)$): Fixes execution beforehand, unaware of random choices by the algorithm
  – **Adaptive on-line**: Knows complete state in prefix $E(t)$, including random choices taken so far (but not future ones)
  – **Adaptive off-line**: Knows complete state in prefix $E(t)$, including random choices taken so far and (next) future ones

• Given assertion $P$ (like “the algorithm terminates”) on executions, a fixed adversary $A$, and initial config. $C_0$
  \[ \text{Prob}[P] = \text{Prob}[R : \text{exec}(A, C_0, R) \text{ satisfies } P] \]
Randomized Consensus

- Adding randomization per se does not circumvent FLP impossibility

- We also need to relax consensus properties. Two possibilities:
  - "Las-Vegas"-Type Randomized Consensus Algorithms
    - Probabilistic termination: Non-faulty processors must decide with some non-zero probability
    - Keep the standard agreement and validity conditions
  - "Monte-Carlo"-Type Randomized Consensus Algorithms
    - Deterministic termination
    - Allow (small) probability for disagreement of terminated processors
Randomized Consensus Algorithm Examples

- Las-Vegas-type binary consensus algorithms (similar to Phase King algorithm):
  
  (1) Decide when there is "overwhelming majority" for a value (but do not terminate, i.e., continue to send own value)
  
  (2) Otherwise, use coin flipping for "symmetry breaking" (i.e., resolving a bivalent configuration) \(\rightarrow\) eventually leads to (1)

- Two variants: Slow/fast expected termination time

- Can be turned into Monte-Carlo-type algorithms, just by letting every process decide (and terminate) at the end of some a priori given round \(K\).
Simple Randomized Binary Consensus (I)

Code for process $p_i$:

Initially $r = 1$ and $prefer = p_i$'s input

1. while true do /* loop over phases $r = 1, 2, \ldots$ */
2. send $<prefer, r>$ to all
3. wait for $n - f$ round $r$ messages
4. if $n - 2f$ rcvd. messages have value $v$ then
5. \hspace{1em} prefer := $v$; decide $v$ /* but continue */
6. elseif $n - 4f$ rcvd. messages have value $v$ then prefer := $v$
7. else prefer := \begin{cases} 0 & \text{with probability } \frac{1}{2} \\ 1 & \text{with probability } \frac{1}{2} \end{cases}
8. $r := r + 1$

“Overwhelming majority” (alg. requires $n \geq 9f + 1$!)

“Symmetry breaking”
Simple Randomized Binary Consensus (II)

• Requires \( n \geq 9f + 1 \) processes, up to \( f \) may be Byzantine

• Validity follows from **Unanimity Lemma**: If all correct procs that reach phase \( r \) prefer \( v \), then all correct procs decide \( v \) by phase \( r \)

• Agreement follows from **Decision Lemma**: If \( p_i \) decides \( v \) in phase \( r \), then all correct procs decide \( v \) by phase \( r + 1 \)

• Decision in any phase occurs with probability at least \( \rho = 2^{-n} \):
  - Case 1: All correct procs set preference using coin flipping \( \Rightarrow \) chose same \( v \) with probability at least \( 2 \cdot 2^{-n} \) (either \( v = 0 \) or \( v = 1 \))
  - Case 2: Some correct procs do not use coin flipping \( \Rightarrow \)
    • all of those set their preference to **same** value \( v \), as pigeon hole argument reveals that \( 2(n-4f-f) > n - f \) correct processes would be required for choosing both 0 and 1 in line 6
    • all remaining procs choose same \( v \) with probability at least \( 2^{-n} \)

• The latter implies Termination within an expected number of phases = \( 2^n \) – this is quite bad …
Implementing a Common Coin

- Randomization also allows to implement a common coin (also called shared coin):
  - All processes toss (suitably biased) local coins
  - Exchange tossed values among all processes
  - Produce the same outcome at all processes with large (ideally constant) probability \( \rho \)

- Very effective for fast (probabilistic) symmetry breaking
- Effort needed for implementation depends on power of adversary
Simple Implementation of Common Coin (I)

• Building block for exchanging values: $V := \text{f-cast}(v)$
  – Disseminates local value $v_i$, of all processes $p_i$, as consistently as possible
  – Returns set (array) $V$ holding the value disseminated by every process (or $\bot$)

• f-cast uses 3 asynchronous rounds ($n \geq 2f + 1$, up to $f \geq 1$ crashes):
  – First round:
    • Send $v$ to all
    • Wait for $n - f$ first round messages
  – Second round:
    • Send values received in first round to all
    • Wait for $n - f$ second round messages
    • Merge data from second round messages
  – Third round:
    • Send values received in second round to all
    • Wait for $n - f$ third round messages
    • Merge data from third round messages and return resulting set $V$

Note: Different processes may see $n - f$ different processes here $\Rightarrow$ returned arrays $V_j$ may differ!

But one can prove:
Every returned $V_j$ contains $v_i$ of all $p_i \in \mathcal{C}$, where $|\mathcal{C}| \geq n - f > n/2$
$\Rightarrow \cap V_j > n/2$
Simple Implementation of Common Coin (II)

• Consider system of \( n \geq 2f + 1 \) processes, up to \( f \) may crash

• Weak adversary: Cannot determine procs most recent coin-flip

• Implementation of common-coin(), with bias \( \rho = \frac{1}{4} \):

\[
c := \begin{cases} 
0 \text{ with probability } \frac{1}{n} \\ 
1 \text{ with probability } 1 - \frac{1}{n}
\end{cases}
\]

\[
\text{coins} := \text{f-\text{cast}}(c) \quad // \quad \text{Note: } \perp = 1
\]

if there exists \( j \) s.t. \( \text{coins}[j] = 0 \)

then return 0

else return 1

---

**Calculate** \( P[1] = \text{Prob}[\text{return} = 1] \):

• \((1 - 1/n)^n \to 1/e\) monotonically incr.

• \( P[1] \geq (1 - 1/n)^n \geq \frac{1}{4} \) (for \( n=2 \))

**Calculate** \( P[0] = \text{Prob}[\text{return} = 0] \):

• If some \( p_i \in \mathcal{C} \) (where \( |\mathcal{C}| > n/2 \)) chose \( c_i = 0 \) \( \Rightarrow \) every proc returns 0

• \( P[0] \geq 1 - (1 - 1/n)^{|\mathcal{C}|} \)

\[
> 1 - (1 - 1/(2|\mathcal{C}|))^{|\mathcal{C}|}
\]

\[
\geq 1 - e^{-1/2} > \frac{1}{4}
\]
Improved Randomized Binary Consensus (I)

Code for process \( p_i \):

Initially \( r = 1 \) and \( \text{prefer} = p_i \)'s input

1. while true do /* loop over phases \( r = 1,2, \ldots */ *
2. \( \text{votes} := \text{f-cast}(<\text{VOTE}, \text{prefer}, r>) \)
3. let \( v \) be majority of phase \( r \) votes
4. if all phase \( r \) votes are \( v \) then decide \( v /* \text{but continue } */ */
5. \( \text{outcomes} := \text{f-cast}<\text{OUTCOME}, v, r>) \)
6. if all phase \( r \) outcome values are \( w \)
7. then \( \text{prefer} := w \)
8. else \( \text{prefer} := \text{common-coin}() \)
9. \( r := r + 1 \)

Ensures a high level of consistency w.r.t. what different procs get

Symmetry breaking
Improved Randomized Binary Consensus (II)

- Requires \( n \geq 2f + 1 \) processes, up to \( f \) may crash (provably optimal)
- Validity follows from **Unanimity Lemma**: If all procs that reach phase \( r \) prefer \( v \), then all nonfaulty procs decide \( v \) by phase \( r \)
- Agreement follows from **Decision Lemma**: If \( p_i \) decides \( v \) in phase \( r \), then all nonfaulty procs. decide \( v \) by phase \( r + 1 \)
- Decision in any phase occurs with probability at least \( \rho = \frac{1}{4} \):
  - Case 1: All correct procs set preference using common-coin \( \rightarrow \) chose same \( v \) with probability \( 2\rho \) (either \( v=0 \) or \( v=1 \))
  - Case 2: Some correct procs do not use common-coin \( \rightarrow \)
    - all of those saw **same** unanimous value \( v \) and set preference to it
    - all remaining procs choose same \( v \) with probability \( \rho \)
- The latter implies Termination within an **expected number of phases** \( 1/\rho = 4 \) – quite good!
Further Reading


1. Prove the Decision Lemma (Slide 10) for the simple randomized binary Byzantine consensus algorithm.

2. Consider a simplified implementation of $f$-cast, which returns the array $V$ already at the end of round 2 (i.e., round 3 is dropped).

   Prove that every returned $V$ contains $v_i$ of all $p_i \in C$, where $|C| > n-2f$, provided that $n \geq 2f+1$ and $f \geq 1$ processes may crash.

3. For the original 3-round version of $f$-cast, prove that every returned $V$ contains $v_i$ of all $p_i \in C$, where $|C| \geq n-f$, provided that $n \geq 2f+1$ and $f \geq 1$. 
Self-Stabilization
Motivation (I)

- Admissible execution for $f=1$ Byzantine process failures:

  - What if faults are transient (i.e., „go away“)?
    - Above execution obviously still admissible
    - Not the case if another fault occurs, despite the fact that there is only one faulty process at every time
Motivation (II)

- Self-stabilizing distributed algorithms:
  - Convergence: Reaches legal state within stabilization time
  - Closure: Remains within set of legal states afterwards
  - No (further) transient failure during stabilization allowed

Recovers even from totally corrupted state:
Classification of States/Executions

- **SS:** No initial state → **Suffixes** replace correct/incorrect execs
  - Legal (satisfy specification),
  - Pseudo-legal (appear legal for finite time)
  - Incorrect (violate specification)

- **Partitioning of system states:**
  - **Legal states:** Any execution starting from it is legal
  - **Safe states:** Any outgoing transition leads to a legal state
  - **Pseudo-legal states:** Any execution starting from it has a legal suffix (may finitely often reach erroneous state, though!)

All possible states
Variants of Self-Stabilization

• Classic SS:
  – Any failure may lead to arb. state
  – No further failures allowed during stabilization

• Local SS:
  – Moderate failures lead to states „close to“ safe ones
    → fast stabilization

• Fault-tolerant SS:
  – Only excessive failures may lead to arbitrary state
  – Restricted number of failures allowed both in legal executions and during stabilization
Dijkstra‘s Classic SS Algorithm (I)

- $n$ Processes are arranged in a unidirectional ring
- Dedicated master process $p_0$
- Goal: Token circulation

Model of computation:
- $p_i$ communicates data to $p_{i+1}$ via dedicated virtual R/W register $R_i$
- In one atomic step, $p_i$ can
  - read $R_{i-1}$
  - compute locally
  - write $R_i$
- Process scheduler: Fair one-by-one (“central daemon”)

- $p_i$‘s local state consists solely of an integer (ranging from 0 to $K - 1$), stored in $R_i$
- We choose: $K = n + 1$
Dijkstra’s Classic SS Algorithm (II)

• Legal execution suffix LE for token circulation problem:
  – Every $E \in \text{LE}$ must be admissible
  – In every configuration in $E$, only one processor holds the token
  – Every processor holds the token infinitely often in $E$

• Processor $p_i$ holds the token if
  – $p_i = p_0$: $R_0 = R_{n-1}$
  – $p_i \neq p_0$: $R_i \neq R_{i-1}$

• Applications:
  – Token passing rings
  – Mutual exclusion
Dijkstra‘s Classic SS Algorithm (III)

Code for $p_0$:
while true do
  if $R_0 = R_{n-1}$ then
    $R_0 := (R_0 + 1) \mod K$
  endif
endwhile

Only $p_0$ can increment values!

executes atomically

Code for $p_i$, $i \neq 0$:
while true do
  if $R_i \neq R_{i-1}$ then
    $R_i := R_{i-1}$
  endif
endwhile

endwhile
Dijkstra‘s Classic SS Algorithm (IV)

• **Some obvious facts:**
  – If all registers are equal in a configuration, then the configuration is safe
  – In every configuration, there is at least one integer in \( \{0, \ldots, n\} \) that does not appear in any register since we only have \( n \) processes

• **Lemma:** In every admissible execution (starting from any configuration), \( p_0 \) holds the token (and thus changes \( R_0 \)) at least once during every \( n \) complete ring cycles.

• **Proof:**
  – Suppose in contradiction there is a segment of \( n \) cycles in which \( p_0 \) does not change \( R_0 \)
  – Once \( p_1 \) takes a step in the first cycle, \( R_1 = R_0 \), and this equality remains true
  – …
  – Once \( p_{n-1} \) takes a step in the \((n-1)\)-st cycle, \( R_{n-1} = R_{n-2} = \ldots = R_0 \)
  – So when \( p_0 \) takes a step in the \( n \)-th cycle, it will change \( R_0 \), contradiction.

\[
\begin{align*}
\text{if } R_0 &= R_{n-1} \text{ then } \\
R_0 &= (R_0 + 1) \mod K \\
\text{if } R_i &\neq R_{i-1} \text{ then } \\
R_i &= R_{i-1}
\end{align*}
\]
Dijkstra‘s Classic SS Algorithm (V)

• **Theorem:** In any admissible execution starting at any configuration $C$, a safe configuration is reached within $O(n^2)$ complete cycles.

• **Proof:**
  – Let $j$ be a value not in any register in configuration $C$
  – By our lemma, $p_0$ changes $R_0$ (by incrementing it) at least once every $n$ cycles
  – Thus eventually $R_0$ holds $j$, in configuration $D$, after at most $O(n^2)$ cycles
  – Since other processes only copy values, no register holds $j$ between $C$ and $D$
  – After at most $n$ more cycles, the value $j$ propagates around the ring from $p_0$ to $p_{n-1}$. 
Food for Thoughts

(1) Show that Dijkstra’s algorithm works also if $K = n$, i.e., equal to the number of processors $n \geq 3$.

(2) Show (by means of a counterexample) that this is no longer true for $K = n - 2$, i.e., two less than the number of processors $n$. 
Further Reading

Self-Stabilization in VLSI Circuits
Radiation-induced Transient Failures (I)

- Sources of cosmic radiation generate high energy (GeV-TeV) charged particles (electrons, protons, nuclei):
  - Rotating neutron stars (pulsars)
  - Supernovae
  - Double star systems
  - Galactic centers, black holes
  - Extragalactic sources (quasars)

- Generation/acceleration mechanisms:
  - Acceleration of charged particles in time-varying magnetic fields („cyclotron mechanisms“)
  - Shock wave acceleration, by particle reflection at fast shock waves (e.g. Supernovae explosions)
Radiation-induced Transient Failures (II)

Soft error rates dominate in VLSI!

Powell, 1959

Soft error rates dominate in VLSI!

[VPSS13]

U. Schmid
Transient Failures: Single-Event Upsets (SEU)

Example: Muller C-Gate

Normal operation:

Fault flips $c_{\text{old}}$ in C-Gate:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$c_{\text{old}}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$c_{\text{old}}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Informal semantics:
- AND for signal transitions
- wait-for-all

Failure goes away
Effects of SETs on Asynchronous Circuits

Example: SEU-tolerant ring oscillator

For simplicity:
* Delay-free C-Gates
* Delay-free wires

Normal operation:

Fault-tolerant operation: SEU in C_a
Effects of SETs on Asynchronous Circuits

Example: SEU-tolerant ring oscillator

For simplicity:
* Delay-free C-Gates
* Delay-free wires

Normal operation:

Faulty operation: Pulse injected

Never goes away!
Simple Running Example: Clock Distribution

Multiple synchronized clock sources (FATAL$^+$)

Clock distribution network (HEX)

Link propagation delays $\epsilon [d^-, d^+]$

Max. skew $\sigma_0$

Max. neighbor skew $\sigma$

Nodes 1-3 are stable, Node 4 is faulty

U. Schmid

192.703 PRDC
Self-Stabilization

Recovery from arbitrary transient faults
⇔
“Correct” state reached from arbitrary initial state
Byzantine Fault-Tolerance

Masking of up to \( f \) arbitrary faults

Node 1
Node 2
Node 3
Node 4

Nodes 1-3 are stable, Node 4 is faulty

requires \( f < n/3 \)
Self-Stabilization + Byzantine Tolerance

Recovery from arbitrary transient faults despite $f < n/3$ permanent faults

Transparent masking of $f < n/3$ transient faults
FATAL\(^+\): SS Byz-FT Clock Generation [DFPS12]

- **SS Pulse Synchronization**
  - Self-stabilizing, but moderate skew, low frequency

- **Tick Synchronization (like DARTS)**
  - Nominally low skew, high freq., but not self-stabilizing

- Force node resync
The HEX Grid [DFLPS16]

- Synchronized clock sources
- Direction of clock propagation
- Width (wrap around)
HEX Algorithm: Firing rules

left-triggered
HEX Algorithm: Firing rules

centrally triggered
HEX Algorithm: Firing rules

right-triggered
HEX Algorithm

Algorithm 1: Pulse forwarding algorithm for nodes in layer $\ell > 0$.

once received trigger messages from (left and lower left) or (lower left and lower right) or (lower right and right) neighbors do

| broadcast trigger message; // local clock pulse
| sleep for some time within $[T^-, T^+]$;
| forget previously received trigger messages

= clock pulses
Analysis Goals

- All message delays non-deterministically in $[d^-, d^+]$, with $\varepsilon = d^+ - d^-$
- Initial skews at most $\sigma_0$
- What is max. layer $\ell$ neighbor skew $\sigma_\ell$?
Skews (Probabilistic Message Delays)

Simulations:

![Graph showing skews in probabilistic message delays](image-url)
Skews (Worst Case: Fault-Free)

- $\max \sigma_\ell$ depends on $[d^-,d^+]$, $\sigma_0$, layer $\ell$ and max. width $W$
- complex "non-local" worst case scenarios $\rightarrow$ analysis difficult
Fault-Tolerance

- Single Byzantine-faulty neighbor per node
- Many faulty nodes system-wide
Fault-Tolerance

- Pulse wave propagates around faults
Fault-Tolerance

- Pulse wave propagates around faults
Fault-Tolerance

- Pulse wave propagates around faults
Fault-Tolerance

- Pulse wave propagates around faults
HEX Implementation

\[ [d^-,d^+] = [d1^-, d2^- + d3^-], d1^+ + d2^+ + d3^+] \]

U. Schmid

192.703 PRDC
Recall: Self-Stabilization

Starting from **arbitrary system state** (e.g., after massive transient faults), we want this:
Does HEX Self-Stabilize?

If nodes in a layer are awake when pulses arrive:

- they are triggered
- they will go to sleep
- they will clear memory when waking up
- they will be awake when the next pulse arrives

=> Self-stabilization (by induction on layers)
Self-Stabilization Despite Byzantine Faults

Consider node $p$ in the following state (e.g. after a transient fault):

- $p$ memorizes pulse
Self-Stabilization Despite Byzantine Faults

- \( p \) memorizes pulse
- faulty node triggers \( p \)
  \( \rightarrow \) goes to sleep
Self-Stabilization Despite Byzantine Faults

- $p$ memorizes pulse
- faulty node triggers $p$ → goes to sleep
- next pulse arrives on left
Self-Stabilization Despite Byzantine Faults

- $p$ memorizes pulse
- faulty node triggers $p$ \(\rightarrow\) goes to sleep
- next pulse arrives on left
- $p$ wakes up & forgets
Self-Stabilization Despite Byzantine Faults

- $p$ memorizes pulse
- faulty node triggers $p$ → goes to sleep
- next pulse arrives on left
- $p$ wakes up & forgets
- next pulse arrives on right
Self-Stabilization Despite Byzantine Faults

→ Original HEX algorithm never synchronizes $p$!

- $p$ memorizes pulse
- faulty node triggers $p$ → goes to sleep
- next pulse arrives on left
- $p$ wakes up & forgets
- next pulse arrives on right
- repeat
Self-Stabilization Despite Byzantine Faults

Fix for HEX-Algorithm: “Forget” pulses after a while

Algorithm 1: Pulse forwarding algorithm for nodes in layer $\ell > 0$.

\begin{algorithm}
\textbf{upon} receiving trigger message from neighbor \textbf{do}
\begin{itemize}
  \item memorize message for $\tau \in [T_{\text{link}}^-, T_{\text{link}}^+]$ time;
\end{itemize}
\textbf{upon} having memorized trigger messages from (left and lower left) or (lower left and lower right) or (lower right and right) neighbors \textbf{do}
\begin{itemize}
  \item broadcast trigger message; // local clock pulse
  \item sleep for $\tau \in [T_{\text{sleep}}^-, T_{\text{sleep}}^+]$ time;
  \item forget previously received trigger messages;
\end{itemize}
\end{algorithm}

Also improves stabilization time
The End
(Part 2)
References