Fault-Tolerant Distributed Algorithms

RiSE Winter School 2013 (Part 2)

Ulrich Schmid
Embedded Computing Systems Group E182/2
s@ecs.tuwien.ac.at
Content (Part 2)

- Advanced Topics in Distributed Algorithms
  - Randomization
  - Self-Stabilization

- The Role of Synchrony Conditions
  - Failure Detectors
  - Real-Time Clocks

- Partially Synchronous Models
  - Models supporting lock-step round simulations
  - Weaker partially synchronous models
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)

• The adversary is typically constrained in
  – what information it can observe in the execution
  – how much computational power it has

• An execution $E = \text{exec}(A, C_0, R)$ of some algorithm involves
  – a particular adversary function $A$, which takes the current execution prefix and provides the next event
  – an initial configuration $C_0$
  – a sequence of random numbers $R$ obtained by the processes in $E$

• Typically, one focuses on worst case adversary functions that maximize some complexity measure of interest.
Randomization (III)

• Given assertion $P$ (like “the algorithm terminates”) on executions, a fixed adversary function $A$, and some $C_0$

$$\text{Prob}[P]=\text{Prob}[R : \text{exec}(A, C_0, R) \text{ satisfies } P]$$

• For an assertion $P = P_k$ (like “the algorithm terminates in $k$ phases”) defining a random variable, the expected value is

$$E[k] = \sum k \ P_k$$

• Taking worst case adversary functions $A$ provides worst-case probabilities and expectations
Randomized Consensus

• Adding randomization per se does not circumvent FLP impossibility

• We also need to relax consensus properties, e.g.:
  – Probabilistic termination: Non-faulty processors must decide with some nonzero probability
  – Keep the same agreement and validity conditions

• Provide two examples of binary consensus algorithms (similar to Phase King algorithm):
  (1) Decide when there is "overwhelming majority" for a value
  (2) Otherwise, use coin flipping for "symmetry breaking" (i.e., resolving a bivalent configuration) \(\rightarrow\) eventually leads to (1)
Simple Randomized Consensus (I)

Code for process $p_i$:
Initially $r = 1$ and $\text{prefer} = p_i$'s input $x_i$
1. while true do /* loop over phases $r = 1, 2, \ldots */
2. send $<\text{prefer}, r>$ to all
3. wait for $n - f$ round $r$ messages
4. if $n - f$ rcvd. messages have value $v$ then
5. prefer := $v$; decide $v$ /* but continue to help others */
6. elseif $n - 2f$ 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 Consensus (II)

- Requires $n \geq 9f + 1$ processes, up to $f$ may be Byzantine
- 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 by any phase occurs with probability at least $2^{-n}$:
  - 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 for their decision, then
    - overwhelming majority $\rightarrow$ all of those set their preference to **same** value $v$
    - 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: \( V := \text{get-core}(v) \)
  – Disseminates local value \( v \) as consistently as possible
  – Returns set (array) \( V \) holding the values disseminated by every process (or \( \bot \))

• Get-core uses 3 asynchronous rounds \( (n \geq 2f + 1, \text{up to } f \text{ crashes}) \):
  – 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
\( \Rightarrow \) return values \( V_j \) may differ!

But one can prove:
Every returned \( V \) contains \( v_i \) of every \( p_i \in \mathcal{C} \), where \( |\mathcal{C}| > n/2 \)
\( \Rightarrow \bigcap V_j > n/2 \)
Simple Implementation of Common Coin (II)

- Consider system of \( n \geq 2f + 1 \) processes, up to \( f \) may crash.
- Weak adversary: Cannot read message contents.
- Implementation of common-coin(), with bias \( \rho = \frac{1}{4} \):

\[
\begin{align*}
    c &:= \begin{cases} 
        0 & \text{with probability } 1/n \\
        1 & \text{with probability } 1 - 1/n
    \end{cases} \\
    \text{coins} &:= \text{get-core}(c) \\
    \text{if there exists } j \text{ s.t. } \text{coins}[j] = 0 \\
    \text{then return 0} \\
    \text{else return 1}
\end{align*}
\]

\[
\begin{align*}
    \text{Calculate } P[1] = \text{Prob[return } = 1]: \\
    &\quad (1-1/n)^n \to 1/e \text{ monotonically incr.} \\
    &\quad P[1] \geq (1-1/n)^n \geq \frac{1}{4} \text{ (for } n=2) \\
\end{align*}
\]

\[
\begin{align*}
    \text{Calculate } P[0] = \text{Prob[return } = 0]: \\
    &\quad \text{If some } p_i \in \mathcal{C} (\text{where } |\mathcal{C}| > n/2) \text{ chose } c_i=0 \Rightarrow \text{every proc returns 0} \\
    &\quad P[0] \geq 1 - (1-1/n)^{|\mathcal{C}|} \\
    &\quad > 1 - (1-1/(2|\mathcal{C}|))^{\mathcal{C}} \\
    &\quad \geq 1 - e^{-1/2} > \frac{1}{4}
\end{align*}
\]
Improved Randomized Consensus (I)

Code for process $p_i$:

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

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

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

Symmetry breaking

U. Schmid

RiSE Winter School 2013
Improved Randomized 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 by any phase occurs with probability at least $\rho$
  - Case 1: All correct procs set prefer using common-coin $\Rightarrow$ chose same $v$ with probability $2\rho$ (either $v=0$ or $v=1$)
  - Case 2: Some correct procs set prefer to unanimous outcome value $w$ $\Rightarrow$
    - all of those saw *same* unanimous outcome value $w$ and set prefer to it
    - all remaining procs choose same $w$ with probability $\rho$
- The latter implies Termination within an expected number of phases $1/\rho = 4$ – which is indeed quite good!
Further Reading


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

- Recovers even from totally corrupted state:
Classification of States

- SS algorithm does not start from specific initial states:
  - States not just correct/incorrect
  - Define legal executions, which satisfy problem specification

- Partitioning of system states:
  - Legal states (LS): Any execution starting from a LS is legal (→ involves LS only)
  - Safe states: Any outgoing transition leads to a LS
  - Pseudo-legal states (PLS): Any execution starting from PLS has a legal suffix (but may stay within PLS and never reach a LS!)
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)

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

![Diagram of processes and registers]

- 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 executions LE for the token circulation problem:
  – Every $E \in \text{LE}$ is admissible (“central daemon”)
  – 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
  – $p_i = p_0$ and $R_0 = R_{n-1}$
  – $p_i \neq p_0$ and $R_i \neq R_{i-1}$

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

Code for \( p_0 \):

\[
\text{while true do}
\begin{align*}
\text{if } R_0 &= R_{n-1} \text{ then} \\
R_0 &:= (R_0 + 1) \mod K
\end{align*}
\text{endif}
\text{endwhile}
\]

Only \( p_0 \) can increment values!

Code for \( p_i, i \neq 0 \):

\[
\text{while true do}
\begin{align*}
\text{if } R_i &\neq R_{i-1} \text{ then} \\
R_i &:= R_{i-1}
\end{align*}
\text{endif}
\text{endwhile}
\]

executes atomically
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.
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 $p_{n-1}$. 
Further Reading

The Role of Synchrony Conditions
Recall Distributed Agreement (Consensus)
Recall Consensus Impossibility (FLP)

Fischer, Lynch und Paterson [FLP85]:

“There is no deterministic algorithm for solving consensus in an asynchronous distributed system in the presence of a single crash failure.”

Key problem:
Distinguish slow from dead!
Consensus Solvability in ParSync [DDS87] (I)

Dolev, Dwork and Stockmeyer investigated consensus solvability in Partially Synchronous Systems (ParSync), varying 5 "synchrony handles":

- Processors synchronous / asynchronous
- Communication synchronous / asynchronous
- Message order synchronous (system-wide consistent) / asynchronous (out-of-order)
- Send steps broadcast / unicast
- Computing steps atomic rec+send / separate rec, send
Consensus Solvability in ParSync [DDS87] (II)

Consensus possible for $f = 1$

Wait-free consensus possible

Consensus impossible
The Role of Synchrony Conditions

Enable failure detection  ⇔  Enforce event ordering

• Distinguish "old" from "new"
• Ruling out existence of stale (in-transit) information
• Creating non-overlapping "phases of operation" (rounds)

• Distinguish slow from dead
Failure Detectors
Failure Detectors [CT96] (I)

- Chandra & Toueg augmented purely asynchronous systems with (unreliable) failure detectors (FDs):

  - Every processor owns a local FD module (an "oracle" – we do not care about how it is implemented!)

  - In every step [of a purely asynchronous algorithm], the FD can be queried for a hint about failures of other procs
Failure Detectors [CT96] (II)

• make mistakes – the (time-free!) FD specification restricts the allowed mistakes of a FD

• **FD hierarchy**: A stronger FD specification implies
  – less allowed mistakes
  – more difficult problems to be solved using this FD
  – But: FD implementation more demanding/difficult

• Every problem \( Pr \) has a **weakest FD** \( W \):  
  – There is a purely asynchronous algorithm for solving \( Pr \) that uses \( W \)
  – Every FD that also allows to solve \( Pr \) can be transformed (via a purely asynchronous algorithm) to simulate \( W \)
Example Failure Detectors (I)

- **Perfect failure detector $P$:** Outputs suspect list
  - *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process
  - *Strong accuracy:* No process is ever suspected before it crashes

- **Eventually perfect failure detector $◊P$:**
  - *Strong completeness*
  - *Eventual strong accuracy:* There is a time after which correct processes are never suspected by correct processes
Example Failure Detectors (II)

- **Eventually strong failure detector $\Diamond S$:**
  - *Strong completeness*
  - *Eventual weak accuracy*: There is a time after which some correct process is never suspected by correct processes

- **Leader oracle $\Omega$: Outputs a single process ID**
  - There is a time after which every not yet crashed process outputs the same correct process $p$ (the "leader")

- **Both are weakest failure detectors for consensus (with majority of correct processes)**
Consensus with $\diamond S$: Rotating Coordinator

Task $T1$:

1. $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
2. while true do
3. \hspace{1em} $c \leftarrow (r_i \mod n) + 1$; $r_i \leftarrow r_i + 1$; $1 \leq r_i < +\infty$

------------- Phase 1 of round $r$: from $p_c$ to all -------------

4. if ($i = c$) then broadcast PHASE1($r_i$, $est_i$) endif;
5. wait until (PHASE1($r_i$, $v$) has been received from $p_c \lor c \in suspected_i$);
6. if (PHASE1($r_i$, $v$) received from $p_c$) then $aux_i \leftarrow v$ else $aux_i \leftarrow \perp$ endif;

------------- Phase 2 of round $r$: from all to all -------------

7. broadcast PHASE2($r_i$, $aux_i$);
8. wait until (PHASE2 ($r_i$, $aux$) msgs have been received from a majority of proc.);
9. let $rec_i$ be the set of values received by $p_i$ at line 8;
% We have $rec_i = \{v\}$, or $rec_i = \{v, \perp\}$, or $rec_i = \{\perp\}$ where $v = est_c$
10. case $rec_i = \{v\}$ then $est_i \leftarrow v$; broadcast DECISION($est_i$); stop $T1$
11. $rec_i = \{v, \perp\}$ then $est_i \leftarrow v$
12. $rec_i = \{\perp\}$ then skip
13. endcase
14. endwhile
Why Agreement? Intersecting Quorums

Intersecting Quorums:

$n=7$
$f=3$

$p$ decides $v$ every $q$ changes its estimate to $v$
Implementability of FDs

• If we can implement a FD like $\Omega$ or $\diamond S$, we can also implement consensus (for $n > 2f$)

• In a purely asynchronous system
  – it is impossible to solve consensus (FLP result)
  – it is hence also impossible to implement $\Omega$ or $\diamond S$

• Back at key question: What needs to be added to an asynchronous system to make $\Omega$ or $\diamond S$ implementable?
  – Real-time constraints [ADFT04, …]
  – Order constraints [MMR03, …]
  – ???
Real-Time Clocks
Distributed Systems with RT Clocks

• Equip every processor $p$ with a local RT clock $C_p(t)$

• Small clock drift $\rho \Rightarrow$ local clocks progress approximately as real-time, with clock rate $\in [1-\rho, 1+\rho]$

• End-to-end delay bounds $[\tau^-, \tau^+]$, a priori known
The Role of Real-Time

- Real-time clocks enable both:
  - Failure detection
  - Event ordering

- [Show later: Real-time clocks are not the only way ...]
Failure Detection: Timeout using RT Clock

\[ \text{status} = \text{do\_roundtrip}(q) \]
\[
\{
\text{send ping to } q \\
\text{TO} := C_p(t) + 5 \text{ seconds} \\
\text{wait until } C_p(t) = \text{TO} \\
\text{if pong did not arrive then} \\
\quad \text{return DEAD} \\
\text{else} \\
\quad \text{return ALIVE}
\}
\]

\( p \) can reliably detect whether \( q \) has been alive recently, if
- the end-to-end delays are at most \( \tau^* = 2.5 \text{ seconds} \)
- \( \tau^* \) is known a priori [at coding time]
Event Ordering: Via Clock Synchronization

**Internal CS:**
- Precision $|C_p(t) - C_q(t)| \leq \pi$
- Progress like RT (small drift $\rho$)
- CS-Alg must periodically resynchronize

**External CS:**
- Accuracy $|C_p(t) - t| \leq \alpha$
- CS-Alg needs access to RT
- External CS $\rightarrow$ internal CS $\pi = 2\alpha$

\[ T \]
\[ C_p(t) \]
\[ C_q(t) \]
\[ \leq \pi \]
FT Midpoint Internal CS-Alg [LWL88]

- A priori bounded $[\tau^-, \tau^+]$ allows to estimate all remote clocks
- Discard $f$ largest and $f$ smallest clock readings (could be faulty)
- Set local clock to midpoint of remaining interval

Before resync ...

After resync ...

$\pi' \leq \pi/2$

$C_p$

$C_q$

$C_q$

$C_p$
Global Positioning System (GPS)

- 4 satellites required to determine $\chi = (x, y, z)$ and $\Delta$
- 1 satellite sufficient for $\Delta$ if $\chi$ is already known

**GPS satellites**

- Satellite clocks synchronized to USNO atomic master clock
- GPS-Receiver solves system of equations
  $$t_i + |\chi - s_i|/c + \Delta = T_i$$
  $$\text{Rec. time: } t_1, t_2 \text{ (unknown)}$$
  $$\text{Local rec. time: } T_1, T_2 \text{ (known)}$$
  $$\text{3D-pos: } s_1 \text{ (known)}$$
  $$\text{3D-pos: } s_2 \text{ (known)}$$
  $$\text{3D-pos: } \chi \text{ (unknown)}$$
Why are Synchronized Clocks Useful?

- Synchronized clocks allow to simulate communication-closed **lock-step rounds** via clock time [NT93]:

![Diagram showing synchronization with clocks](image)

- Only requirement: \( R \geq \tau^+ + \pi \) holds!
- Lock-step rounds \( \rightarrow \) perfect failure detection at end of rounds
Perfect FD \xrightarrow{\text{Lock-Step Round Simulation}}

- Attempt round simulation at $p$: Waiting for either
  - arrival of round message from $q$, or
  - $p$’s instance of $P$ suspects $q$

- Problem faced by $q$:
  - $msg_k$ not received in round $k$, although $p$ alive after round $k$
  - $q$ even receives $msg_{k+1}$ in round $k+1$ in this example
Using RT Clocks: Deficiencies

- Algorithms like `do_roundtrip(.)` have system-dependent time values (unit „seconds“) in their code / variables → not easily portable to e.g. faster hardware
- Fail-operational systems might tolerate occasional loss of timeliness properties – but never of safety properties
- Unfortunately:
  - Safety properties like agreement typically rely on the reliable operation of `do_roundtrip(.)` and similar primitives
  - End-to-end delay bounds $\tau^+$ that always hold are difficult to determine in real systems

➢ Try to relax timing assumptions in ParSync models …
Partially Synchronous Models
Recall: Synchronous Model

• „The“ classic model
  – Transmission delay bound $\tau^+$
  – Computing step time bound $\mu^+$
  – Bounded-drift local clocks available

• Allows (Byzantine-tolerant) implementation of
  – Internal clock synchronization
  – Lock-step rounds
  – etc.
The Timed Asynchronous Model

- Cristian & Fetzer [CF99]:
  - Alternating bad and good periods:
    - Transmission delay bound $\tau^+$
    - Computing step time bound $\mu^+$
  - Bounded-drift local RT clocks available
  - Local clocks allow to detect good/bad periods $\implies$ TA algorithms are always safe and live in good periods

- TA algorithms allow to implement (non-Byzantine) fail-aware services, including eventual lock-step rounds
Classic Partially Synchronous Models (I)

• „The“ classic ParSync models
  Dolev, Dwork & Stockmeyer [DDS87]
  Dwork, Lynch & Stockmeyer [DLS88]
  Attiya, Dwork, Lynch & Stockmeyer [ADLS94]

• Semi-synchronous model by Ponzio & Strong [PS92]

• Common system parameters:
  – Bounded processor speed ratio \( \Phi = \frac{\mu^+}{\mu^-} \)
  – Transmission delay bound \( \Delta \)

• Archimedean model by Vitanyi [Vit84]
  – Bounded speed ratio \( S = \frac{\tau^+}{\mu^-} \)
Classic Partially Synchronous Models (II)

Processes can **locally time-out** messages:

- **The classic ParSync models** [DDS87, DLS88] and [ADLS94] assume
  
  • $\Delta$ given in multiples of (unknown) minimal computing step time $\mu$ [hence $\tau^+ = \Delta \cdot \mu$ real-time seconds]
  
  • spin loop counting $f(\Phi, \Delta)$ steps allows to time-out messages
    
    [implements local clock with real-time rate $\in [1/\Phi, 1]$]

- **Archimedean model** [Vit84] also allows to time-out messages via spin-loop for $S$ steps

- **Semi-synchronous model** [PS92] assumes
  
  • $\Delta = \tau^+$ given in real-time seconds
  
  • bounded-drift local RT clocks available for timing-out messages
Classic Partially Synchronous Models (III)

**Variants of ParSync models:** System parameters \((\Delta, \Phi)\)

1. known and hold from the beginning

2. known and hold from unknown global stabilization time (GST) on

3. unknown and hold from the beginning / from GST on: Learn \((\Delta, \Phi)\), by continuously increasing estimate values
Time-Free Message-Timeout in ParSync?

- Implementation of \texttt{do\_roundtrip(p)} in the ParSync models of [DLS88] or [Vit85]:

\begin{verbatim}
{ send ping to p
  for i=1 to x do no-op /* x=f(\Delta, \Phi) \text{ resp. } x=f(s) \text{ is dimensionless! */
    if pong did not arrive then
      return DEAD
    else
      return ALIVE
}
\end{verbatim}

- But: No obvious correlation between processor step times and message delays \(\rightarrow\) not really time-free …
The $\Theta$/ABC-Model

For classic ParSynt models:

- Timing assumptions are primarily used for ordering events.
- Actual duration ($D$) irrelevant.
- Is it possible to define a time-free ParSynt model based on event ordering in the first place?

\[ \Theta = 5 \]

\[ D_p \]

\[ D_r \]

\[ D_q \]

\[ \text{status} = \text{do_roundtrip}(q) \]
\[ \text{send ping to } q \]
\[ \text{for } i = 1 \text{ to } \Theta \text{ do} \]
\[ \begin{align*}
\text{send } \text{delay_ping}(i) \text{ to } r \\
\text{wait for } \text{delay_pong}(i) \text{ from } r
\end{align*} \]
\[ \text{end} \]
\[ \text{if } \text{pong} \text{ did not arrive then} \]
\[ \text{return DEAD} \]
\[ \text{else} \]
\[ \text{return ALIVE} \]
\[ \} \]
The $\Theta$-Model: Bounded E-t-E Delay Ratio

Widder & Schmid [WS09]

- End-to-end delays of all messages in transit at $t$
  - minimum $\tau^-(t)$
  - maximum $\tau^+(t)$
- $\tau^+(t)$ and $\tau^-(t)$ may vary arbitrarily with time, but:
- Ratio $\tau^+(t)/\tau^-(t)$ bounded by [known or even unknown] system parameter $\Theta$
Byzantine FT Clock Sync in the $\Theta$-Model

For $n \geq 3f + 1$ with up to $f$ Byz. failures:

- Suppose $p$ sends $\text{tick}(C+1)$ at time $t$
- Then, $q$ also sends $\text{tick}(C+1)$ by time $t + 2\tau^+ - \tau^-$
- Fastest tick-frequency of any $p$: $1/\tau^-$

$\Rightarrow$ Clock ticks occur approximately synchronously, with precision $\pi(\Theta)$

On init

$\rightarrow$ send $\text{tick}(0)$ to all; $C := 0$;

If got $\text{tick}(l)$ from $f + 1$ nodes and $l > C$

$\rightarrow$ send $\text{tick}(C+1), \ldots, \text{tick}(l)$ to all;

$C := l$;

If got $\text{tick}(C)$ from $2f + 1$ nodes

$\rightarrow$ send $\text{tick}(C+1)$ to all;

$C := C + 1$;
The Asynchronous Bounded Cycle Model

Robinson & Schmid [RS08]

- The ABC Model just bounds the ratio of the number of forward and backward-oriented messages in cycles

- Example: $\Theta = 4.5$
- 2 consecutive „slow“ messages
- Cycle with 9 enclosed „fast“ messages
- No larger cycles allowed

- No implicit or explicit reference to real-time
  - Messages with $\tau^-(t) = 0$ allowed
  - No need to relate independent messages in the system
  - We proved: Any $\Theta$-algorithm works correctly in the ABC model
Partial Order of ParSync Models

- DLS … [DLS88] with known $\Delta$, $\Phi$
- $\Theta$ … ABC/$\Theta$-Model with known $\Theta$
- $\text{DLS}^u$ … [DLS88] with unknown $\Delta$, $\Phi$
- $\Theta^u$ … ABC/$\Theta$-Model with unknown $\Theta$
- FLP … FLP-Model
Even Weaker ParSync Models?

• All the ParSync Models seen so far allow to build
  – lock-step rounds, or at least
  – eventual lock-step rounds

• Solving consensus is easy here.

• We know that lock-step rounds are stronger than failure detectors that are sufficient for solving consensus:
  – Perfect failure detector \( P \)
  – Leader oracle \( \Omega \)

• Are there weaker ParSync models where only such FDs can be implemented?
Weaker Partially Synchronous Models
Finite Average Roundtrip-Time Model (I)

Fetzer, Schmid and Süskraut [FSS04]

– Asynchronous system with crash failures
– Unknown lower bound \( \mu^- \) for computing step time
– Unknown average round-trip time bounds

\[
\lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} RTT(k) < \infty
\]

\( RTT(k) \) and hence \( \tau^+ \) unbounded, yet

– Average after \( n \) „Epochs“ is

\[
\frac{n(n+1)}{n(n+1)-(n-1)n/2} = 2 \cdot \frac{n^2 + n}{n^2 + 3n} < \infty
\]
Finite Average Roundtrip-Time Model (II)

- The FAR model assumptions
  - do not allow to implement lock-step rounds
  - do allow to implement the eventually perfect FD P
  - can solve consensus if $n > 2f$

- Key ideas for P implementation:
  - Implement weak local clock [via spin-loop] for timing-out messages
  - Time-out roundtrips using adaptive timeout value TV
    - If fast RT occurs [before TO]: Increase TV, to prepare for future slow RTs
    - If slow RT occurs [after TO]: (Could) decrease TV, since fast RTs must eventually follow due to finite average RTT
Weak Timely Link Models (I)

Aguilera, Delporte, Fauconnier, Toueg [ADFT04], Hutle, Malkhi, Schmid, Zhou [HMSZ09]:

• Partially synchronous processors ($\Phi$) with crash failures
• Almost all communication asynchronous, except:
• At least one process $p$ must be an $\diamond f$-source:
  – After some (unknown) time, $p$ has timely links to at least $f$ neighbors
    $[No\ message\ sent\ at\ time\ t\ is\ processed\ after\ t+\tau^+\ (unknown)]$
  – Note: A link to a crashed process is timely per definition!
• Allows to implement $\Omega$, and hence solving consensus for $n > 2f$
• An $\diamond f$-1-source is provably not sufficient
• Currently weakest WTL model [HMSZ09]: A moving $\diamond f$-source, where the $f$ timely links can change with time
Weak Timely Link Models (II)

\(\Omega\) implementation: Every process

- periodically broadcasts heartbeat message (HB)
- times-out HBs of all neighbors
  - using weak local clock [implemented via step counting in spin-loop]
  - timeout value increased on every TO [= no HB received before expiration]
- broadcasts \textit{accusation message} \texttt{acmsg}(q) on every TO for q’s HB
- if \(\ n-f\ \) \texttt{acmsg}(q) are received, then increment \texttt{acc_count}[q]
- \(\Omega\)-output: q with min. \texttt{acc_count}[q]

\[\begin{align*}
\end{align*}\]
Even Weaker Models (I)

• Investigate models for weaker problems than consensus

• Candidate of choice: \textit{k-set agreement} [Cha93]:
  – Input values from finite domain \( V \) with \(|V| > k\)
  – Processes must decide on at most \( k \) different output values system-wide

• Well-known properties:
  – Weakening of consensus (= 1-set agreement)
  – Requires \(|f/k| + 1\) rounds in synchronous systems with up to \( f \) crashes
  – Impossible in asynchronous systems if \( f \geq k \) crashes
Even Weaker Models (II)

- $k$-set agreement allows to further explore the synchronous/asynchronous solvability border

- There are models where
  - $k-1$-set agreement (hence consensus) is impossible
  - $k$-set agreement is possible

- Two major directions of research:
  - Failure detectors
  - ParSync models
The End
(Part 2)
References

References

References