182.703: Problems in Distributed Computing
(Part 1)
WS 2023
Ulrich Schmid
Institute of Computer Engineering, TU Vienna
Embedded Computing Systems Group E191-2
s@ecs.tuwien.ac.at

http://ti.tuwien.ac.at/ecs/teaching/courses/prdc
Target: Fault-tolerant Distributed RT Systems

Spatially distributed reactive computations

Real-time requirements

Partial failures

Worst-case response time $RT \leq T_{\text{max}}$
Interdisciplinary Research

REAL-TIME SYSTEMS

FAULT-TOLERANT DISTRIBUTED ALGORITHMS

DIGITAL INTEGRATED CIRCUITS

U. Schmid

182.703 PRDC
Motivation:
Distributed Fault-Tolerant Clock Generation in Systems-on-Chip
Clocking in Systems-on-Chip (I)

Classic synchronous paradigm

- **Concept:** Common notion of time for entire chip
- **Method:** Single crystal oscillator
  Global, phase-accurate clock tree

Disadvantages

- Cumbersome clock tree design (physical limits!)
- High power consumption
- Clock is **single point of failure!**
Clocking in Systems-on-Chip (II)

Alternative: DARTS clocks

- **Concept:** Multiple synchronized tick generators
- **Method:** Distributed FT tick generation algorithm
  Implemented in (asynchronous) HW

http://ti.tuwien.ac.at/ecs/research/projects/darts

Advantages

- Reasonable synchrony
- Uncritical clock distribution
- Clock is no single point of failure!
The DARTS Distributed Algorithm

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

If got $\text{tick}(l)$ from $f+1$ nodes and $l > C$
→ send $\text{tick}(C+1), \ldots, \text{tick}(l)$ to all;

$C := l$;

If got $\text{tick}(C)$ from $2f+1$ nodes
→ send $\text{tick}(C+1)$ to all;

$C := C+1$;

For $n \geq 3f + 1$ and up to $f$ node failures, with (small) e-t-e delays $\in [d, d+\varepsilon]$:

- Suppose node $p$ sends $\text{tick}(C+1)$ at time $t$
- Then, node $q$ also sends $\text{tick}(C+1)$ by time $t+d+2\varepsilon$

$\Rightarrow$ Clock ticks occur approximately at the same time
$n \geq 3f+1$: Why do Failures hurt so much?

Toy example:

With this algorithm, B and C never get closer together

Will prove: Majority $n = 2f + 1$ not enough for $f$ Byz. failures!
Pipe Compare Signal Generators (PCSGs): There exists a dedicated detection circuit for each pair of pipes which generates the status signals $GEQ_{p,q}^{o/e}(t)$ and $GR_{p,q}^{o/e}(t)$. In particular, $GEQ_{p,q}^{o}(t')$ becomes active (i.e.,
(i) $r_{p,q}^{se}(t) = 1$ or $r_{p,q}^{se}(t)$
(ii) $r_{p,q}^{se}(t) = 0$

**Definition 4.1.** (Direct Causality). Let $I(t')$ and $O(t)$ be two events of some specific signal input and output, respectively, of a correct component $C$. Then $I(t')$ and $O(t)$ are directly causally related, denoted by $I(t') \rightarrow O(t)$, if

(i) $t' < t$
(ii) $t' \neq t$

**Theorem 4.13.** (Precision). The precision $\pi \geq |b_q(t) - b_p(t)|$ of our algorithm is bounded by $\pi \leq \left\lceil \frac{T_{fir}}{T_{fis}} \right\rceil + 1$.

**Proof.** First of all, it is established for a given $i$, $k, i.e., r_{p,q}^{se}(t) \geq \left\lceil \frac{T_{fis}}{T_{fis}} \right\rceil + 1$, $t_k \leq \max_{p}(t')$.

**Theorem 4.14.** (Accuracy). Given $\Delta = t_2 - t_1$, the accuracy $|b_p(t_2) - b_p(t_1)|$ of any correct process $p$ is bounded by

$$b_{max}(t') = \left\lceil \frac{\Delta}{T_{fis}} \right\rceil + \min \left\{ \pi, \left\lceil \frac{\Delta}{D} - \frac{\Delta}{T_{fis}} \right\rceil \right\}.$$

**Proof.** The upper bound for accuracy will be shown first: It is known that $\forall t_i, b_p(t) \geq b_{max}(t) + a + (1 - I_{sync}(t))$ and $\forall t_i, b_p(t) \leq b_{max}(t) + \pi - (1 - I_{sync}(t_1))$. By applying Lemma 4.11, $b_p(t_2) - b_p(t_1) \leq \left\lceil \frac{\Delta}{T_{fis}} \right\rceil + \pi - (1 - I_{sync}(t_1))$. Moreover, from Lemma 4.7 it follows that $b_p(t_2) - b_p(t_1) \leq \left\lceil \frac{\Delta}{D} \right\rceil + \pi + 1$. Hence, $b_p(t_2) - b_p(t_1) \leq \min \left\{ \left\lceil \frac{\Delta}{T_{fis}} \right\rceil + \pi + 1, \left\lceil \frac{\Delta}{D} \right\rceil \right\} + \min \left\{ \pi, \left\lceil \frac{\Delta}{D} - \frac{\Delta}{T_{fis}} \right\rceil \right\} \leq \left\lceil \frac{\Delta}{T_{fis}} \right\rceil + \min \left\{ \pi + 1, \left\lceil \frac{\Delta}{D} - \frac{\Delta}{T_{fis}} \right\rceil \right\} \text{ since } [x + y] \leq [x] + [y].$

To prove the lower bound, first define $b_1 = b_p(t_1)$, $b_2 = b_p(t_2)$ and $t_{b_1}^p \leq t_2$, $t_{b_2}^p \leq t_2$ as the points in time when $p$ sends tick $b_1$ and $b_2$. Clearly $t_{b_2}^p > t_2$, $t_{b_1}^p < t_2$. The sum $t_{b_2}^p - t_{b_1}^p$ is the delay to send $b_2$ after $b_1$ has been sent.
DARTS Implementation

Node p

Remote Pipe

Local Pipe

Pipe Compare Signal Gen.

Remote Pipe

Diff-Gate

Local Pipe

Threshold Logic

GEQ°

GEO°

= 2

= f

Remote clk_in

Pipeline 1

Diff-Gate

Pipeline 3

Pipeline 2

Pipeline 3f+1

Pipe Compare Signal Gen.

Time

Bus/Signal

<table>
<thead>
<tr>
<th>Bus/Signal</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock1</td>
<td>0</td>
</tr>
<tr>
<td>clock2</td>
<td>0</td>
</tr>
<tr>
<td>clock3</td>
<td>0</td>
</tr>
<tr>
<td>clock4</td>
<td>0</td>
</tr>
<tr>
<td>clock5</td>
<td>0</td>
</tr>
</tbody>
</table>

3f+1

1

= 2f + 1

= 2f + 1

= f + 1

Compare Signal Gen.

Remote
Pipe

Pipeline

Local
Pipe

Diff-
Gate

Remote
Pipe

Pipeline

Local
Pipe

Threshold Logic

GEQ°

GEO°

= 2

= f

U. Schmid

182.703 PRDC
DARTS Extension: Self-Stabilization

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

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

force node reset
Introduction to Distributed Algorithms
Content (Part 1)

- Basics:
  - Distributed Computing Model
  - Synchrony and Fault-Tolerance
  - Correctness Proofs

- Some Appetizers:
  - Consistent Broadcasting
  - Consensus

- Food for Thoughts
Classic Modeling and Analysis

• Processors/processes modeled as interacting state machines

• **Zero-time** atomic computing steps, usually time-triggered
  – Message Passing (MP): [receive] + compute + [send]
  – Shared Memory (SHM): [accessSHM] + compute

![Diagram of process interaction]

• System timing parameters:
  – Operation durations modeled via **inter-step times** $\epsilon[\mu^-,\mu^+]$ (often $\mu^- = 0$)
  – Message delays modeled as **end-to-end delays** $\epsilon[\tau, \tau^+]$ (often $\tau = 0$)
Synchrony Models: 2 Extremes …

Lock-step synchronous systems

- Computing step times:
  \[ \mu^- = \mu^+ = R \]
- Message delays
  \[ 0 \leq \tau \leq \tau^+ \leq R \]
- Perfectly synchronized rounds

Asynchronous systems

- Computing step times:
  - \( \mu^- = 0 \)
  - \( \mu^+ \) finite (but unbounded)
- Message delays
  - \( \tau = 0 \)
  - \( \tau^+ \) finite (but unbounded)
Failure Models

• „Deterministic“ failure models
  – At most $f$ of $n$ processors in the system may fail
  – Correct processes do not a priori know who has failed and when and how

• Failure semantics ranging from
  – Crash failures: Processors stop operating, possibly within a step
  – Byzantine failures [LSP82]: Processors can do what they want

• Real processors etc. fail probabilistically → Coverage analysis

• Restrict our attention to message passing systems here:
  – Typically fully connected, with dedicated links between every pair of processors
  – Receiver cannot be spoofed w.r.t. sender of a message
  – [Communication between correct processes typically considered reliable]
Message Passing vs. Shared Memory (I)

- MP can always be simulated in a SHM system
- The opposite is not generally true:
  - Linerarizable AsyncSHM can be simulated in AsyncMP only when a majority of processes do not crash \( n > 2f \)
- MP is more elementary than SHM
- SHM is more powerful than MP

\[ \text{Impossibility proof for } n \leq 2f : \]
\[ p \in S_0, |S_0| = n/2, \ q \in S_1, \ |S_1| = n/2 \]

\[ \alpha_0 : \begin{cases} \text{Write}_p R := 1 \\ \text{Read}_q R = 0 \end{cases} \]
\[ \alpha_1 : \begin{cases} \text{Read}_q R = 0 \\ \text{Write}_p R := 1 \end{cases} \]
\[ \alpha_2 : \begin{cases} \text{Read}_q R = 0 \end{cases} \]

Merge \( \alpha_0 \& \alpha_1 : \text{Indistinguishable for } S_0, S_1 ! \)

\(-\text{linearizable!}\)
Message Passing vs. Shared Memory (II)

- **Wait-free ($f = n-1$) event ordering** in AsyncSHM:
  - $p$ knows (already by $t_p$) whether $q$ has done the event!
  - $p$ and $q$ can **agree** on order of having done their events if no “in-between” crash occurs!

- **Impossible in AsyncMP!**

- **Uses „write-before-read“:**
  - $p$ sets $O[p]:=1$ if $q$ has set $R[q]:=1$
  - Both $O[p]:=0$ and $O[q]:=0$ impossible
    - Event order $p$ before $q$ if $O[p]=0 \land O[q]=1$ or $O[p]=1 \land O[q]=1$
    - Event order $q$ before $p$ if $O[q]=0 \land O[p]=1$
    - Event order **undecided (forever)** if either $p$ or $q$ crashes in between its two Writes

---

U. Schmid

182.703 PRDC 19
Correctness Proofs

- Global state transitions
  - Configuration $C = \text{vector of processor local states [+ in-transit messages for MP]}$
  - State transition = result of a single processor taking a step

- Algorithm vs. Adversary
  - Adversary determines which and when events $\varphi$ (like processor $p_i$ takes a step) happen ($\rightarrow$ Async. systems: Adv. subject to admissibility (fairness) conditions)
  - Algorithm determines what actually happens in the corresponding step

- Executions and traces
  - Execution $E = \text{sequence of configurations alternating with events } C_0, \varphi_1, C_1, \varphi_2, C_2, \varphi_3, C_3, \ldots$
  - Trace $T = \text{(sub-)sequence of „interesting“ events (or states)}$

- Correctness proofs: Set of generated traces satisfies
  - Safety properties (“something bad never happens“)
  - Liveness properties (“something good eventually happens“)
Some Appetizers
Consistent Broadcasting
Consistent Broadcasting [ST87]

- Want to build **authenticated reliable broadcasting**: 
  - Any process $p_s$ may have some message $m_s$ to broadcast: $\text{bcast}(p_s,m_s)$
  - Every correct process shall eventually call $\text{accept}(p_s,m_s)$, and shall be sure that the received $m_s$ originates in $p_s$
  - Do not use real authentication (cryptography)!

- Very useful primitive:
  - Clock synchronization
  - Consensus
  - etc.
Properties Consistent Broadcasting

Time-free specification:

- **Correctness:** If a correct processor $p_s$ executes $\text{bcast}(p_s, m_s)$, then every correct processor eventually calls $\text{accept}(p_s, m_s)$
- **Unforgeability:** If a correct processor $p_s$ never executes $\text{bcast}(p_s, m_s)$, then no correct processor ever calls $\text{accept}(p_s, m_s)$
- **Relay:** If some correct processor calls $\text{accept}(p_s, m_s)$, then every other correct processor eventually also calls $\text{accept}(p_s, m_s)$
Implementation

\( \text{bcast}(p_s, m_s) \) at \( p_s \)

send \((\text{init}, p_s, m_s)\) to all processors

\( \text{accept}(p_s, m_s) \) at every \( p_i \)

\[
\begin{align*}
\text{if} & \text{ got } (\text{init}, p_s, m_s) \text{ from } p_s \\
& \quad \rightarrow \text{ send } (\text{echo}, p_s, m_s) \text{ to all } [\text{once}] \\
\text{if} & \text{ got } (\text{echo}, p_s, m_s) \text{ from } f + 1 \\
& \quad \rightarrow \text{ send } (\text{echo}, p_s, m_s) \text{ to all } [\text{once}] \\
\text{if} & \text{ got } (\text{echo}, p_s, m_s) \text{ from } 2f + 1 \\
& \quad \rightarrow \text{ call } \text{accept}(p_s, m_s)
\end{align*}
\]

System model:

- At most \( f \) Byzantine faulty processors
- \( n \geq 3f + 1 \)
- E-t-e delays \( \in [d, d+\varepsilon] \):

- Message sent by correct proc at \( t \) got by correct receiver proc within \( [t+d, t+d+\varepsilon] \)
- Every proc gets at most \( f \) faulty echo messages from different procs
- At most \( f \) echo messages available at \( p_i \) by \( t \) could be missing at \( p_j \) by \( t + \varepsilon \)
Correctness Proof (Time-dependent Version)

- **Correctness**: If a correct proc $p_s$ executes $\text{bcast}(p_s, m_s)$ by $t$, then every correct processor eventually calls $\text{accept}(p_s, m_s)$ by $t+2(d+\varepsilon)$

- **Unforgeability**: If a correct proc $p_s$ does not execute $\text{bcast}(p_s, m_s)$ by $t$, then no correct processor calls $\text{accept}(p_s, m_s)$ by $t+2d$

- **Relay**: If a correct processor calls $\text{accept}(p_s, m_s)$ at $t$, then every other correct processor also calls $\text{accept}(p_s, m_s)$ by $t+d+2\varepsilon$

### Relay:

- $p_i$ at $t$
- Any $p'_j$ at $t+\varepsilon$
- Any $p_j$ at $t+d+2\varepsilon$

\[ 2f + 1 \]

\[ f + 1 \]
A Note on Formal Verification

Problem: Correctness proofs for FT distributed algorithms (at most $f$ out of $n$ processes Byzantine faulty)

- State-of-the-art: Case-by-case “paper-and-pen” proofs, typically lacking rigorousness

- **Exciting alternative approach:** Apply finite-state parameterized model checking, based on advanced abstractions

First results:

- Rigorous foundation: Interval and counter abstractions for threshold bounds: [JKSVW13b]

- Fast model-checking of Byzantine fault-tolerant atomic broadcast algorithms: [JKSVW13]
A Classic Problem: Distributed Agreement (Consensus)

Yes

Yes

Yes

Yes

Yes

None meet

All meet

No

No

No

No
Consensus Properties

• Every process $p_i$
  – has initial value $x_i$ chosen from some finite set $V$
  – shall irrevocably decide on output value $y_i$

• **Termination:** Every correct processor eventually decides

• **Agreement:** Every two correct processors $p_i, p_j$ decide on the same value $y_i = y_j$

• **Validity:** If all correct processors have the same input value $x$, then $x$ is the only possible decision value
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!
Distributed Agreement (Consensus) - FLP
Synchronous Consensus

Lamport, Shostak and Pease [LSP82]:

“There is a deterministic algorithm for solving consensus in a synchronous distributed system of $n \geq 3f+1$ processors in the presence of at most $f$ Byzantine failures.”

But:

It is impossible to solve consensus if $n = 3f$!
Impossibility of Consensus for $f = 1, n = 3$

- Suppose correct algorithm $\mathcal{A} = (A,B,C)$ for $(p_0,p_1,p_2)$ existed
- Assume $p_0$ faulty
- By Validity:
  - $x_1 = x_2 = 0 \rightarrow y_1 = y_2 = 0$
  - $x_1 = x_2 = 1 \rightarrow y_1 = y_2 = 1$
- By Agreement:
  - $x_1 \neq x_2 \rightarrow y_1 = y_2$
„Easy Impossibility Proofs“ [FLM86] (I)

Arrange 6 correct processors in a ring:

Resulting execution will not solve consensus, but …
„Easy Impossibility Proofs“ [FLM86] (II)

Local view of $p_1, p_2$:

By Validity: Decision must be $y_1 = y_2 = 0 \ldots$
Local view of $p_3, p_4$:

By Validity: Decision must be $y_3 = y_4 = 1$ ...
Local view of $p_2, p_3$:

By Agreement: Decision should be $y_2 = y_3$ → Contradiccion
Food for Thoughts
Communcation Failures

• Correct processes with link failures:
  1. Per communication round: Omission and and even arbitrary link failures
     • Full message exchange: Both send and receive link failure restrictions apply
     • Single broadcast: Only send link failure restriction applies
  2. Different failures in different rounds

• Known results:
  – \( n > f_l^{ra} + f_l^{sa} \) necessary & sufficient for solving consensus with link omission failures only
  – \( n > 2f_l^{ra} + 2f_l^{sa} \) necessary & sufficient for solving consensus with arbitrary link failures
Exercises

1. Find the smallest values for $S,R,S’,R’,S”,R”$ in the consistent broadcasting implem. below for $f^{ra}_l, f^{sa}_l$ arbitrary link failures and up to $f$ byzantine-faulty processes:

   if got $(init, p_s, m_s)$ from $p_s$
   → send $(echo, p_s, m_s)$ to all [once]
   if got $(echo, p_s, m_s)$ from $Sf^{sa}_l + Rf^{ra}_l + f + 1$
   → send $(echo, p_s, m_s)$ to all [once]
   if got $(echo, p_s, m_s)$ from $S’f^{sa}_l + R’f^{ra}_l + 2f + 1$
   → call accept($p_s, m_s$)

2. Find an „easy impossibility proof“ that shows that $n=4$ processors are not enough for solving consensus with $f^{ra}_l = f^{sa}_l = 1$ (and $f = 0$)
The End
(Part 1)
References