182.703: Problems in Distributed Computing

(Part 1)

WS 2017

Ulrich Schmid

Institute of Computer Engineering, TU Vienna
Embedded Computing Systems Group E182/2
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 CIRCIRTS

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:**

- A: 08:00
- B: 10:00
- C: 08:00

$\Rightarrow$ 08:00

**A** ("Byzantine" faulty)

08:00 10:00

08:00 10:00 08:00

C (correct)  B  $\Rightarrow$ 10:00

- 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.,

\[ GEQ_{p,q}^{o}(t') \]

\[ GR_{p,q}^{o}(t') \]

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) they are

(ii) there is

Similarly

(i) \( r_{p,q}^{self}(t) \in \mathbb{N}_{odd} \) and

Theorem 4.13. (Precision). The precision \( \pi \geq |b_{q}(t) - b_{p}(t)| \) of our algorithm is bounded by \( \pi \leq \left\lfloor \frac{T_{last}}{T_{first}} \right\rfloor + 1 \).

Proof. First of all, established for all \( k + 1 \), i.e., \( t_{k} \leq b_{max}(t') \). Assume that

\[ b_{max}(t') \]

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 \( \max \left\{ 0, \left\lfloor \frac{\Delta - T_{sync} - T_{+}}{D} \right\rfloor \right\} \leq |b_{p}(t_{2}) - b_{p}(t_{1})| \leq \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + \min \left\{ \pi + 1, \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{first}} \right\rfloor \right\} \).

Proof. The upper bound for accuracy will be shown first: It is known that \( \forall t : b_{p}(t) \geq b_{max}(t) - \pi + (1 - I_{sync}(t)) \) and \( \forall t : b_{p}(t) \leq b_{max}(t) \) from Lemma 4.11 and Lemma 4.13. Thus \( b_{p}(t_{2}) - b_{p}(t_{1}) \leq b_{max}(t_{2}) - b_{max}(t_{1}) + \pi - (1 - I_{sync}(t_{1})) \). By applying Lemma 4.11, \( b_{p}(t_{2}) - b_{p}(t_{1}) \leq \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + 2I_{sync}(t_{1}) - 1 + \pi \leq \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + \pi + 1 \leq \left\lfloor \frac{\Delta}{D} \right\rfloor + \pi + 1 \). Moreover, from Lemma 4.7 it follows that \( b_{p}(t_{2}) - b_{p}(t_{1}) \leq \left\lfloor \frac{\Delta}{D} \right\rfloor \). Hence, \( b_{p}(t_{2}) - b_{p}(t_{1}) \leq \min \left\{ \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + \pi + 1, \left\lfloor \frac{\Delta}{D} \right\rfloor \right\} \leq \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + \min \left\{ \pi + 1, \left\lfloor \frac{\Delta}{D} \right\rfloor - \frac{\Delta}{T_{first}} \right\} \leq \left\lfloor \frac{\Delta}{T_{first}} \right\rfloor + \min \left\{ \pi + 1, \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{first}} \right\rfloor \right\} \) since \( \lfloor x + y \rfloor \leq \lfloor x \rfloor + \lfloor y \rfloor \).

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 \( t_{b_{1}+1}^{p} > t_{2} \),

U. Schmid

182.703 PRDC
DARTS Implementation

Node p

Pipeline 1
Diff-Gate

Remote Pipe

Pipeline 3f+1

Local Pipe

Diff-
Gate

Pipe Compare Signal Gen.

Remote
Pipe

Clocks

Bus/Signal

Time

120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns

Clock 1

0 1 0 1 0 1 0 1 3

Clock 2

0 1 0 1 0 1 0 1 0

Clock 3

0 1 0 1 0 1 0 1 0

Clock 4

0 1 0 1 0 1 0 1 0

Clock 5

0 1 0 1 0 1 0 1 0

Remote clock

3f+1

1

= 2f+1 = 2f+1

= f+1 = ...

Compare Signal Gen.

Remote
Pipe

Pipeline 3f+1

Local
Pipe

Diff-
Gate

Pipe Compare Signal Gen.

Remote
Pipe

Diff-
Gate

Local
Pipe

GEO°

Threshold Logic

GEO°

1

= 2

= f

= g

U. Schmid

182.703 PRDC

11
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

- 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 $\Rightarrow$ 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

Impossibility proof for $n \leq 2f$:

$p \in S_0, |S_0| = n/2, q \in S_1, |S_1| = n/2$

 Merge $\alpha_0 & \alpha_1$: Indistinguishable for $S_0, S_1$!

- 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 some step!
  - \(p\) and \(q\) can **agree** on order of having done some step 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

\[
\begin{align*}
p & \quad \text{Write}_p R[p]:=1 \\
R[p]=0, O[p]=⊥ & \quad \text{Write}_q R[q]:=1 \\
R[q]=0, O[q]=⊥ & \quad \text{Write}_q O[q]:=y \\
\end{align*}
\]

\[
\begin{align*}
x:=\text{Read}_p R[q] & \quad \text{Write}_p O[p]:=x \\
y:=\text{Read}_q R[p] & \quad \text{Write}_q O[q]:=y \\
\end{align*}
\]
Correctness Proofs

- **Global state transitions**
  - Configuration $C =$ 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 (→ Async. systems: Adv. subject to admissibility (fairness) conditions)
  - Algorithm determines what actually happens in the corresponding step

- **Executions and traces**
  - Execution $E =$ sequence of configurations alternating with events $C_0, \varphi_1, C_1, \varphi_2, C_2, \varphi_3, C_3, \ldots$
  - Trace $T =$ (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

**bcast**($p_s, m_s$) at $p_s$

send ($init, p_s, m_s$) to all processors

**accept**($p_s, m_s$) at every $p_i$

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 $f + 1$
   → send ($echo, p_s, m_s$) to all [once]
if got ($echo, p_s, m_s$) from $2f + 1$
   → call **accept**($p_s, m_s$)

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 proc
- 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 $\text{bcast}(p_s, m)$ by $t$, then every correct processor eventually calls $\text{accept}(p_s, m)$ by $t+2(d+\varepsilon)$

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

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

Relay:

- $p_i$ at $t$
- any $p_j'$ at $t+\varepsilon$
- any $p_j$ at $t+d+2\varepsilon$
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 novel approach:** Apply finite-state parameterized **model checking**, based on advanced abstractions

Some nice results:

- Rigorous foundation: Interval and counter abstractions for threshold bounds: [JKSVW13b]
- Fast model-checking of Byzantine fault-tolerant atomic broadcast algorithms: [JKSVW13]
Consensus
A Classic Problem: Distributed Agreement (Consensus)

Yes

Yes

No

None meet

All meet

No

No

29

U. Schmid
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
Asynchronous Consensus Impossibility

Fischer, Lynch and 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!
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:
  - $x_1 \neq x_2 \rightarrow y_1 = y_2$
Arrange 6 correct processors in a ring:

Resulting execution will not solve consensus, but ...
Local view of $p_1, p_2$:

By Validity: Decision must be $y_1 = y_2 = 0$ ...
„Easy Impossibility Proofs“ [FLM86] (III)

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 $y_2 = y_3 \rightarrow Contradicion$
Food for Thoughts
Exercise: Consensus via Lossy Links

- Consider a synchronous system of $n=2$ processors that never fail, which are connected via a pair of directed links that may lose messages:

- Prove that deterministic consensus is impossible even when at most one link may be lossy in a round.
- Prove that consensus can be solved if it is known that the link $p \rightarrow q$ is not lossy in some round.
- Prove that consensus can be solved if exactly one of the links is lossy in every round.
The End
(Part 1)
References