Target: Fault-tolerant Distributed RT Systems

Spatially distributed reactive computations

Real-time requirements

Partial failures

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

REAL-TIME SYSTEMS

FAULT-TOLERANT DISTRIBUTED ALGORITHMS

DIGITAL INTEGRATED CIRCUITS

deadlines
Pressure
response times
jobs
resources
rate-monotonic
jobs
resources

leader election
reliable broadcast
leader election
reliable broadcast

consenus
clock synchronization
consenus
clock synchronization

uniprocessor
scheduling
uniprocessor
scheduling

ParSync model
message passing
ParSync model
message passing

EDP
metastability
EDP
metastability

Boolean circuits
multiprocessor
Boolean circuits
multiprocessor

simulation
logic synthesis
simulation
logic synthesis

SEU
asynchronous circuits
SEU
asynchronous circuits

systems-on-chip
clocking
dependable architectures
systems-on-chip
clocking
dependable architectures

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

[Link to DARTS project](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

For $n \geq 3f + 1$ and up to $f$ node failures, with (small) $\varepsilon$-t-$\varepsilon$ 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

---

**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;

$C := C + 1$;

---

U. Schmid

182.703 PRDC
$n \geq 3f + 1$: Why do Failures hurt so much?

**Toy example:**

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

$\rightarrow$ 08:00

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

$\rightarrow$ 08:00

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

$\rightarrow$ 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, $GEQ_{p,q}(t')$ becomes active (i.e.,

$GEQ_{p,q}(t')$ previous

(i) $r_{p,q}^{s,e}(t)$

(ii) $[r_{p,q}^{s,e}(t)]_{a}$

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

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

Proof. First of all, it is established for $i.e., \#I' = \pi + 1$, i.e., $t_k \leq b_{\text{max}}(t') + \left\lfloor \frac{\Delta}{T_{\text{first}}} \right\rfloor + \min \left\{ \pi + 1, \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{\text{first}}} \right\rfloor \right\}$.

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 $\max \left\{ 0, \frac{\Delta}{T_{\text{first}}} - \frac{\Delta}{T_{\text{first}}} \right\}$ and $\min \left\{ \pi + 1, \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{\text{first}}} \right\rfloor \right\}$.

Proof. The upper bound for accuracy will be shown first: It is known that $b_p(t) \geq b_{\text{max}}(t) - \pi + (1 - I_{\text{sync}}(t))$ and $b_p(t) \leq b_{\text{max}}(t)$ from Lemma 4.13 and Lemma 4.11. Thus $b_p(t_2) - b_p(t_1) \leq b_{\text{max}}(t_2) - b_{\text{max}}(t_1) + \pi - (1 - I_{\text{sync}}(t_1))$. By applying Lemma 4.11, $b_p(t_2) - b_p(t_1) \leq \left\lfloor \frac{\Delta}{T_{\text{first}}} \right\rfloor + 2I_{\text{sync}}(t_1) - 1 + \pi \leq \left\lfloor \frac{\Delta}{T_{\text{first}}} \right\rfloor + \pi + 1 \leq \left\lfloor \frac{\Delta}{T_{\text{first}}} \right\rfloor + \pi + 1$. Moreover, from Lemma 4.7 it follows that $b_p(t_2) - b_p(t_1) \leq \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{\text{first}}} \right\rfloor$. Hence, $b_p(t_2) - b_p(t_1) \leq \min \left\{ \left\lfloor \frac{\Delta}{T_{\text{first}}} \right\rfloor + \pi + 1, \left\lfloor \frac{\Delta}{D} - \frac{\Delta}{T_{\text{first}}} \right\rfloor \right\}$.

To prove the lower bound, first define $b_1 = b_p(t_1)$, $b_2 = b_p(t_2)$ and $t_{b_1} \leq t_2$, $t_{b_2} \leq t_2$ as the points in time when $p$ sends tick $b_1$ and $b_2$. Clearly $t_{b_2+1} > t_2$. 

U. Schmid

182.703 PRDC
DARTS Implementation
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 processes p and q with inter-step and end-to-end delays]

- 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:
  - Linearizable AsyncSHM can be simulated in AsyncMP only when a majority of processes \((n > 2f)\) do not crash
- MP is more elementary than SHM
- SHM is more powerful than MP

**Impossibility proof for** \(n \leq 2f\):

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

\[ \alpha_0: \begin{cases} 
R = 0, S_1 \text{ dead} \\
t_0 
\end{cases} \]

\[ \alpha_1: \begin{cases} 
R = 0, S_0 \text{ dead} \\
t_0 
\end{cases} \]

\[ t_1 \]

\[ \text{Read}_q R = 0 \]

\[ \text{Write}_p R := 1 \]

\[ \text{Merge } \alpha_0 \& \alpha_1: \text{Indistinguishable for } S_0, S_1! \]

\[ \text{Write}_p R := 1 \]

\[ \sim \text{linearizable!} \]

\[ \alpha_2: \begin{cases} 
R = 0 \\
t_1 
\end{cases} \]

\[ \text{Read}_q R = 0 \]
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

\[ \begin{align*}
   p & \xrightarrow{\text{Write}_p R[p] := 1} \quad x := \text{Read}_p R[q] \\
   q & \xrightarrow{\text{Write}_q R[q] := 1} \quad y := \text{Read}_q R[p] \\
   p & \xrightarrow{\text{Write}_p O[p] := x} \\
   q & \xrightarrow{\text{Write}_q O[q] := y}
\end{align*} \]
Correctness Proofs

• Global state transitions
  – Configuration $C = \text{vector of processor local states} [+ \text{in-transit messages for MP}]$
  – 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 \((init, p_s, m_s)\) to all processors

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

- if got \((init, p_s, m_s)\) from \( p_s \)
  \( \rightarrow \) send \((echo, p_s, m_s)\) to all [once]
- if got \((echo, p_s, m_s)\) from \( f + 1 \)
  \( \rightarrow \) send \((echo, p_s, m_s)\) to all [once]
- if got \((echo, p_s, m_s)\) from \( 2f + 1 \)
  \( \rightarrow \) call \( \text{accept}(p_s, m_s) \)

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/init 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 $t+d+2\varepsilon$
A Note on Formal Verification

• Typical distributed algorithms proofs are definitely „handwaving“, compared to verification standards

• Making proofs amenable to theorem-proving is tedious [SWR02]

• Model checking is challenging, even for simple problems like CB:
  – **Parameterization:** How to handle not just unspecified number of processes, but rather both unspecified \((n, f)\)?
  – **Failures:** How to exhaustively incorporate allowed faulty behaviors?
  – **Synchrony assumptions:** How to deal with asynchronous/synchronous/timed/partially synchronous systems?

• We are working on this in the context of RiSE [JKSVW13]: „Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms“
Consensus
A Classic Problem: Distributed Agreement (Consensus)

Yes

Yes

No

Yes

Yes

No

No

No

Yes

Yes

None meet

All meet

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

Yes

Yes

Yes

No

No

Yes

None meet

All meet

No

32
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 …
“Easy Impossibility Proofs“ [FLM86] (II)

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 $y_3 = y_4 = 1 \ldots$
„Easy Impossibility Proofs“ [FLM86] (IV)

Local view of $p_2, p_3$:

By Agreement: Decision should be $y_2 = y_3 \rightarrow$ Contradicion
Food for Thoughts
Communciation Failures

• Correct processes with link failures:
  1. Per communication round: Omission and/or 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:
  – \( n > f_l^r + f_l^s \) necessary & sufficient for solving consensus with pure link omission failures
  – \( n > f_l^r + f_l^{ra} + f_l^s + f_l^{sa} \) necessary & sufficient for solving consensus with link omission and arbitrary failures

\[ f_l^s \geq f_l^{sa} \quad \text{Send link failures} \]
\[ f_l^{ra} \leq f_l^r \quad \text{Receive link failures} \]
Exercises

1. Find the smallest values for $S, R, S', R', S'', R''$ in the CB implem. below for arbitrary link failures ($f_l^r = f_l^{ra}$ and $f_l^s = f_l^{sa}$):

   \begin{verbatim}
   if got (init, p_s, m_s) from p_s
       \rightarrow send (echo, p_s, m_s) to all [once]
   if got (echo, p_s, m_s) from $Sf_l^{sa} + Rf_l^{ra} + f + 1$
       \rightarrow send (echo, p_s, m_s) to all [once]
   if got (echo, p_s, m_s) from $S'f_l^{sa} + R'f_l^{ra} + 2f + 1$
       \rightarrow call accept(p_s, m_s)
   \end{verbatim}

   Required number of procs:
   - $n \geq S''f_l^{sa} + R''f_l^{ra} + 3f + 1$

   Link failure lower bound:
   - $n > f_l^r + f_l^{ra} + f_l^s + f_l^{sa}$

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