Distributed Graph Algorithms and Very Large Scale Integrated Circuits

Danny Dolev

Specific work jointly with Matthias Függer, Christoph Lenzen, Martin Perner, Ulrich Schmid and Noam Teomim
Distributed Systems

In this talk we focus on fault tolerance of distributed functionalities -- to be implemented in hardware.
Some Systems Must Not Fail

- space
- military
- power grid
- medical science
- security
- vehicles
Defects (Electromigration)

Wiskers

Hillock

M. Ohring, Reliability and Failure of Electronic Materials and Devices, 1998

ASM Corp. Shanghai
Process and Operational Variations

Even if there isn’t a complete short or open, resistance and capacitance variations can lead to trouble.

Chip temperature map
Heat affects unstable connections and clock drifts

Small transistor dissipating 5mW in an SOI wafer; University of Bolton
Decades of Fault Tolerance

- Nasa initiated these lines of research
  - Byzantine Faults
  - Self-stabilized systems (transient faults)
- Utopia
  - Efficient protocols that overcome both types of faults
- DARTS project of Austrian Aerospace implemented Byzantine tolerant protocols in hardware
Model

- Local clocks with bounded drifts
- A bound “d” on the time it takes two correct nodes to communicate
- Message passing
- FIFO and authenticated channels
- Transient faults
- A fraction of the nodes may be permanently faulty
  - Omission
  - Byzantine
- The number of nodes and their IDs are common knowledge
Typical Objectives

- Simulate synchronous rounds
  - Common numbering of rounds is another objective
- Produce “coordinated” pulsing
- Run consensus protocols
- Synchronize clocks
- Fault tolerant routing

- We discuss such protocols in a fully connected system and on various communication graphs
Vast majority of previous research was in this model.

Aim: “round separation”

- Synchronize the nodes such that:
  - All messages sent by correct nodes in a given round is received by all correct nodes within that round.

Thus, no correct node sends a message of the new round if any correct node is still willing to accept messages of the previous round.
Round Separation

\[ P_1 \quad \text{sending} \quad \text{sending} \quad \text{sending} \]

\[ P_2 \quad \text{sending} \quad \text{sending} \quad \text{sending} \]

\[ P_3 \quad \text{sending} \quad \text{sending} \quad \text{sending} \]
The challenge

- Nodes start out “of the blue”
  - No common clock
  - No sense of round numbering or any other consistent relative states

- The only means nodes have is to -
  - send and receive messages
  - measure duration of time passed

- Assume first transient faults and permanent omission faults
Node’s State Machine

Rule: Every d switch states

1. Send received messages
2. Accept received messages
3. Clear receive buffer
4. But how to get there?
Dry Run

\[ P_1 \]

\[ P_2 \]

\[ P_3 \]
Node’s State Machine

Rule: if within d there are n-f in your state – switch states

1 → 2

1

Clear receive buffer

2

Send

3

4

Accept received messages
Lower Layer Protocol

1) If you see (n-f) messages of the same state in the last d:
   a) If it’s your state move one state ahead.
   b) Otherwise, move to that state
      Either way send “I moved to X” (where X is the state you moved to).
2) If you received the message “I moved to X”, move to X and send “I jumped to X” message.
3) If you received “I jumped to X” message, move to X (and don’t announce anything).
4) If you didn’t receive any “move” message within 2d move to state 1.
5) If you sent a “move” message but didn’t receive (n-f) “move”/”jump” messages within 2d of the sending, stop sending “move”/”jump” messages (but follow the protocol) until you see a steady state (n-f messages from a single state within 4d).
6) Every d announce your state.

The actual protocol is complicated – We will not cover the details.
Dealing with Byzantine nodes

- Aim: “round counting”
  - Synchronize the nodes such that:
    - All nodes periodically increase their round count by 1 within a small time window of each other
Why is this difficult?

...Byzantine tolerant, but not self-stabilizing
Main Result (complete graph)

There exists a “wrapper” protocol $A(\cdot)$:

**Theorem**
Given: synchronous consensus protocol $P$
- resilient to rushing
- terminates in $R$ rounds (w.h.p.)
- sends $B$ bits per node

Then: $A(P)$ simulating synchronous rounds
- global round counter modulo $M$
- stabilizes in $O(R)$ time (w.h.p.)
- sends $O(nB+n^2+n \log M)$ bits per node (for stabilization)
- $O(1)$ broadcasted bits overhead per node when stabilized

Result by D. Dolev and C. Lenzen
At the heart of the previous solution:

A self stabilizing and Byzantine pulse generating process ensures recovery from arbitrary transient faults despite $f<n/3$ permanent faults.

FATAL+: Robust Pulse Generation. Dolev, Fuegger, Lenzen, and Schmid. Under submission to JACM.
VLSI Circuits

and many more

It is a distributed system with all its challenges

Core Architecture

On-chip router

Core-Control

Core Architecture
Metastability

Bistable element (memory cell) with positive feedback

\[ u_{i,2} = u_{o,1} \]

stable (HI)

metastable

\[ u_{i,1} = u_{o,2} \]

stable (LO)
Further challenges

- Uncertainty regarding signals spreading around the chip
- Time difference between local and distant events
- Identifying node’s internal events

- Tight synchronization
- Simple state machines = less logic
- Tolerance to significant clock drifts
Resulting implementation (FATAL$^+$)

• Complexity bounds:
  • optimal resilience (n=3f+1 nodes for f faults)
  • full connectivity (linear in n necessary)
  • few 1-bit channels per link
  • $O(n)$ worst-case stabilization time (~$10^{-3}s$)
  • $O(1)$ stabilization time for most cases (~$10^{-5}s$)
  • gate complexity $O(n \log n)$ per node

• Clock distribution local for each node
  • no fault-tolerance required
  • clock *generation* network covers less area
Scalability – clock trees?

recovery from arbitrary transient faults ⇐
“correct” state reached from arbitrary initial state

Clock trees are self-stabilizing!
(But they cannot mask faults)
Scalability Goals

- Distribute clock signal (from multiple sources)
- Small connectivity
- Uniform edge length
- Small clock skew between neighbors
- Byzantine fault-tolerance
- Self-stabilization
Use Robust Generation + Distribute

The HEX Grid

- Small connectivity
- Uniform edge length

Direction of clock propagation
Fault-Tolerance

- one Byzantine fault per neighborhood
- must wait for pulse from two neighbors
Fault-Tolerance

- one Byzantine fault per neighborhood
- must wait for pulse from two neighbors
Fault-Tolerance

- one Byzantine fault per neighborhood
- must wait for pulse from two neighbors
Fault-Tolerance

- one Byzantine fault per neighborhood
- must wait for pulse from two neighbors
Fault-Tolerance

- wave propagates around faults
Fault-Tolerance

- wave propagates around faults
Fault-Tolerance

- wave propagates around faults
Fault-Tolerance

- wave propagates around faults
Nature deals with local Byzantines
Fault-Tolerance

- small connectivity
- uniform edge length
- Byzantine Tolerance
Skew: Fault-Free Case

- assume that link delays are from $[1, 1+\varepsilon]$
- neighbors in layer $\ell$ trigger at most $\varepsilon \ell$ time apart
Skew: Fault-Free Case

between 1 and $1+\varepsilon$
time per layer
Skew: Fault-Free Case

- but: links within layers keep skew in check

between 1 and 1+\varepsilon
time per layer
Skew: Fault-Free Case

- but: links within layers keep skew in check
- we show a worst-case bound of $1 + O(\varepsilon^2 \ell)$

between 1 and $1 + \varepsilon$
time per layer
Some Plotting

width

layer
Skew: Fault-Free Case

worst-case execution
Skew: Fault-Free Case

random delays
Skew and Faults

- faulty nodes can influence propagation by $O(1)$
Skew and Faults

- faulty nodes can influence propagation by $O(1)$
Skew and Faults

- faulty nodes can influence propagation by $O(1)$
Skew and Faults

- faulty nodes can influence propagation by $O(1)$
- worst-case skew of $O(f + \varepsilon^2 \ell)$ with $f$ faults
Skew and Faults

wave propagation with one fault
Skew and Faults

wave propagation with multiple faults
Skew and Faults

- small connectivity
- uniform edge length
- Byzantine tolerance
- small clock skew
Multiple Pulses

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

\begin{verbatim}
\textbf{once} received 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 some time within $[T^-, T^+]$
  \item forget previously received trigger messages
\end{itemize}
\end{verbatim}

- go to sleep once triggered pulse
- wake up & clear memory once wave has passed
Self-Stabilization (assuming no permanent fault)

If nodes in a layer are awake when a wave arrives:
- they are triggered
- they will go to sleep
- they will clear memory upon waking up
- they will be awake when the next pulse arrives

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

Does not work with worst-case faults:

- pulse memorized
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
- pulse arrives on left
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
- pulse arrives on left
- node goes to sleep
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
- pulse arrives on left
- node goes to sleep
- node wakes up
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
- pulse arrives on left
- node goes to sleep
- node wakes up
- pulse arrives on right
Self-Stabilization Despite Faults

Does not work with worst-case faults:

- pulse memorized
- faulty node triggers
- pulse arrives on left
- node goes to sleep
- node wakes up
- pulse arrives on right
- repeat
Self-Stabilization Despite Faults

Fix: “forget” pulse signals after a while

=> self-stabilization with faults

also: improves stabilization time

---

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

upon receiving trigger message from neighbor do
    memorize message for $\tau \in [T_{\text{link}}^-, T_{\text{link}}^+]$ time;
upon having memorized 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 $\tau \in [T_{\text{sleep}}^-, T_{\text{sleep}}^+]$ time;
    forget previously received trigger messages;
HEX Summary

- HEX has many promising features:
  - few edges of similar length
  - fault containment
  - self-stabilization
  - $O(f+\varepsilon^2\ell)$ worst-case skew, better on average
HEX Summary

- HEX has many promising features:
  - few edges of similar length
  - fault containment
  - self-stabilization
  - $O(f+\varepsilon^2\ell)$ worst-case skew, better on average

- Future work:
  - implementation in state-of-the-art hardware
  - reduce skews further
  - formal verification of HEX and FATAL$^+$
“Wild” Synchronization...
“The benefit of being synchronized...”
Questions?