Failure Detection and Propagation in HPC systems

George Bosilca\textsuperscript{1}, Aurélien Bouteiller\textsuperscript{1}, Amina Guermouche\textsuperscript{1}, Thomas Hérault\textsuperscript{1}, Yves Robert\textsuperscript{1,2}, Pierre Sens\textsuperscript{3} and Jack Dongarra\textsuperscript{1,4}

1. University Tennessee Knoxville
2. ENS Lyon, France
3. LIP6 Paris, France
4. University of Manchester, UK

SC’16 – November 15, 2016
Failure detection: why?

Nodes do crash at scale (you’ve heard the story before)

Current solution:
1. Detection: TCP time-out (∼ 20mn)
2. Knowledge propagation: Admin network

Work on fail-stop errors assumes instantaneous failure detection

Seems we put the cart before the horse
Resilient applications

- Continue execution after crash of one node
Resilient applications

- Continue execution after crash of **several** nodes
Resilient applications

• Continue execution after crash of **several** nodes
• Need *rapid* and *global* knowledge of group members
  ① **Rapid**: failure detection
  ② **Global**: failure knowledge propagation
Resilient applications

- Continue execution after crash of **several** nodes
- Need *rapid* and *global* knowledge of group members
  1. **Rapid**: failure detection
  2. **Global**: failure knowledge propagation
- Resilience mechanism should **come for free**
Resilient applications

- Continue execution after crash of **several** nodes
- Need *rapid* and *global* knowledge of group members
  1. **Rapid**: failure detection
  2. **Global**: failure knowledge propagation
- Resilience mechanism should **have minimal impact**
Contribution

- Failure-free overhead constant per node (memory, communications)
- Failure detection with minimal overhead
- Knowledge propagation based on fault-tolerant broadcast overlay
- Tolerate an arbitrary number of failures (but with bounded frequency of occurrence)
- Logarithmic worst-case repair time
Outline

1. Model
2. Failure detector
3. Worst-case analysis
4. Implementation & experiments
Outline

1. Model
2. Failure detector
3. Worst-case analysis
4. Implementation & experiments
Framework

- Large-scale platform with (dense) interconnection graph (physical links)
- One-port message passing model
- Reliable links (messages not lost/duplicated/modified)
- Communication time on each link: randomly distributed but bounded by $\tau$
- Permanent node crashes
Failure detector

**Definition**

**Failure detector:** distributed service able to return the state of any node, alive or dead. **Perfect if:**

1. any failure is eventually detected by all alive nodes and
2. no alive node suspects another alive node of being dead

**Definition**

**Stable configuration:** all dead nodes are known to all processes (nodes may not be aware they are in a stable configuration).
Vocabulary

- Node = physical resource
- Process = program running on node
- Thread = part of a process that can run on a single core
- Failure detector will detect both process and node failures
- Failure detector mandatory to detect some node failures
Outline

1. Model
2. Failure detector
3. Worst-case analysis
4. Implementation & experiments
Timeout techniques: $p$ observes $q$

- **Pull technique**
  - Observer $p$ sends an *Are you alive* message to $q$
  - 😞 More messages
  - 😞 Long timeout

- **Push technique** [1]
  - Observed $q$ periodically sends heartbeats to $p$
  - 😊 Less messages
  - 😊 Faster detection (shorter timeout)

Timeout techniques: platform-wide

• All-to-all:
  - ☀ Immediate knowledge propagation
  - ☹ Dramatic overhead

• Random nodes and gossip:
  - ☀ Quick knowledge propagation
  - ☹ Redundant/partial failure information (more later)
  - ☹ Difficult to define timeout
  - ☹ Difficult to bound detection latency
Algorithm for failure detection

- Processes arranged as a ring
- Periodic heartbeats from a node to its successor
- Maintain ring of alive nodes
  → Reconnect ring after a failure
  → Inform all processes
Reconnecting the ring

\( \eta \): Heartbeat interval

\( \delta \): Timeout,
\( \delta > \tau \)

Reconnection message
Broadcast message

Heartbeat
Reconnecting the ring

$\eta$: Heartbeat interval

Heartbeat

$\delta$: Timeout, $\delta \gg \tau$

Reconnection message

Broadcast message
Reconnecting the ring

\( \eta \): Heartbeat interval
\( \delta \): Timeout, \( \delta \gg \tau \)

Heartbeat
Reconnecting the ring

η: Heartbeat interval
δ: Timeout, δ >> τ

Heartbeat

Reconnection message

η\{ 0 1 2 3 4 ... \}
\δ
Reconnecting the ring

\[ \eta \]: Heartbeat interval
\[ \delta \]: Timeout, \( \delta \gg \tau \)

Heartbeat

Reconnection message

Diagram showing the reconnection process with heartbeat intervals and timeout.
Reconnecting the ring

\[ \eta \{ \]

\[ \delta \{ \]

\[ 2\delta \{ \]

\[ Ring \ reconnected \]

\[ \eta: \ Heartbeat \ interval \]

\[ \delta: \ Timeout, \ \delta \gg \tau \]

- Heartbeat
- Reconnection message
Reconnecting the ring

\( \eta \): Heartbeat interval

\( \delta \): Timeout, \( \delta \gg \tau \)

- \( \rightarrow \) Heartbeat
- \( \rightarrow \) Reconnection message
- \( \rightarrow \) Broadcast message

-Ring reconnected
Algorithm

**task** Initialization

- emitter\(_i\) ← \((i - 1)\) mod \(N\)
- observer\(_i\) ← \((i + 1)\) mod \(N\)
- HB-Timeout ← \(\eta\)
- Susp-Timeout ← \(\delta\)
- \(D_i\) ← \(\emptyset\)

**end task**

**task** T1: When HB-Timeout expires

- HB-Timeout ← \(\eta\)
- Send heartbeat\((i)\) to observer\(_i\)

**end task**

**task** T2: upon reception of heartbeat\((emitter_i)\)

- Susp-Timeout ← \(\delta\)

**end task**

**task** T3: When Susp-Timeout expires

- Susp-Timeout ← \(2\delta\)
- \(D_i\) ← \(D_i\) \(\cup\) emitter\(_i\)
- dead ← emitter\(_i\)
- emitter\(_i\) ← \(\text{FindEmitter}(D_i)\)
- Send NewObserver\((i)\) to emitter\(_i\)
- Send BcastMsg\((\text{dead}, i, D_i)\) to Neighbors\((i, D_i)\)

**end task**

**task** T4: upon reception of NewObserver\((j)\)

- observer\(_i\) ← \(j\)
- HB-Timeout ← 0

**end task**

**task** T5: upon reception of BcastMsg\((\text{dead}, s, D)\)

- \(D_i\) ← \(D_i\) \(\cup\) \{\text{dead}\}
- Send BcastMsg\((\text{dead}, s, D)\) to Neighbors\((s, D)\)

**end task**

**function** \(\text{FindEmitter}(D_i)\)

- \(k\) ← emitter\(_i\)
- while \(k \in D_i\) do
  - \(k\) ← \((k - 1)\) mod \(N\)
- return \(k\)

**end function**
• Hypercube Broadcast Algorithm [1]
  • Disjoint paths to deliver multiple broadcast message copies
  • Recursive doubling broadcast algorithm by each node
  • Completes if $f \leq \lfloor \log(n) \rfloor - 1$
    ($f$: number of failures, $n$: number of alive processes)

Failure propagation

- Hypercube Broadcast Algorithm
  - Completes if \( f \leq \lfloor \log(n) \rfloor - 1 \) (\( f \): number of failures, \( n \): number of alive processes)
  - Completes within \( 2\tau \log(n) \)

- Application to failure detector
  - If \( n \neq 2^l \)
    - \( k = \lfloor \log(n) \rfloor \)
    - \( 2^k \leq n \leq 2^{k+1} \)
    - Initiate two successive broadcast operations
  - Source \( s \) of broadcast sends its current list \( D \) of dead processes
  - No update of \( D \) during broadcast initiated by \( s \)
    (do NOT change broadcast topology on the fly)
Quick digression

- Need fault-tolerant overlay with small fault-tolerant diameter and easy/fast one-port routing

- Known only for specific values of $n$:
  - Hypercubes: $n = 2^k$
  - Binomial graphs: $n = 2^k$
  - Circulant networks: $n = cd^k$
  - ...
Outline

1 Model

2 Failure detector

3 Worst-case analysis

4 Implementation & experiments
Worst-case analysis

Theorem

With \( n \leq N \) alive nodes, and for any \( f \leq \lfloor \log n \rfloor - 1 \), we have

\[
T(f) \leq f(f + 1)\delta + f\tau + \frac{f(f + 1)}{2}B(n)
\]

where \( B(n) = 8\tau \log n \).

- 2 sequential broadcasts: \( 4\tau \log(n) \)
- One-port model: broadcast messages and heartbeats interleaved
Worst-case scenario

\[ T(f) \leq f(f+1)\delta + f\tau + \frac{f(f+1)}{2}B(n) \]

- \( T(f) \leq \text{ring reconstruction + broadcasts (for the proof)} \)

- Process \( p \) discovers the death of \( q \) at most \textbf{once}

  \[ i - \text{th dead process discovered dead by at most } f - i + 1 \text{ processes} \]

  \[ \Rightarrow \text{at most } \frac{f(f+1)}{2} \text{ broadcasts} \]

- \( R(f) \) ring reconstruction time

  For \( 1 \leq f \leq \lfloor \log n \rfloor - 1 \),

  \[ R(f) \leq R(f - 1) + 2f\delta + \tau \]
$R(f) \leq R(f - 1) + 2f\delta + \tau$

- $R(1) \leq 2\tau + \delta \leq 2\delta + \tau$
- $R(f) \leq R(f - 1) + R(1)$ if next failure non-adjacent to previous ones
- Worst-case when failing nodes consecutive in the ring
- Build the ring by “jumping” over platform to avoid correlated failures

$\tau + \delta \leq 2\delta$ to detect the failure of 3

4 detects failure of 2 after $2\delta$

4 detects failure of 1 after $2\delta$

Ring reconnected

Broadcast messages of the failure of processes 3, 2 and 1

HB=heartbeat

NO=NewObserver

Bcast=Broadcast Operation
Worst-case scenario

\[ T(f) \leq f(f + 1)\delta + f\tau + \frac{f(f + 1)}{2}B(n) \]
Worst-case scenario

\[ T(f) \leq (f + 1)\delta + f\tau + \frac{f(f+1)}{2}B(n) \]

Too pessimistic!?
Worst-case scenario

1. If time between two consecutive faults is larger than $T(1)$, then average stabilization time is $T(1) = O(\log n)$

2. If $f$ quickly overlapping faults hit non-consecutive nodes, $T(f) = O(\log^2 n)$

3. If $f$ quickly overlapping faults hit $f$ consecutive nodes in the ring, $T(f) = O(\log^3 n)$

Large platforms: two successive faults strike consecutive nodes with probability $2/(n-1)$
Risk assessment with $\tau = 1\mu s$

\[ P(\geq \lfloor \log_2(n) \rfloor \text{ failures in } T(\lfloor \log_2(n) \rfloor - 1)) < 0.000000001 \]

- With $\mu_{\text{ind}} = 45$ years, $\delta \leq 60s \Rightarrow$ timely convergence
- Detector causes negligible overhead to applications (e.g., $\eta = \delta/10$)
Simulations

Average stabilization time ⇒ see paper!

(results confirm that:
- overlapping failures are rare
- overlapping failure strike independently
- average stabilization time remains close to \( \delta \))
Outline

1. Model
2. Failure detector
3. Worst-case analysis
4. Implementation & experiments
• Observation ring and propagation topology implemented in Byte Transport Layer (BTL)
• No missing heartbeat period:
  • Implemented in MPI internal thread independently from application communications
  • RDMA put channel to directly raise a flag at receiver memory
    → No allocated memory, no message wait queue
• Implementation in ULFM / Open MPI
Case study: ULFM

- Extension to the MPI library allowing the user to provide its own fault tolerance technique
- Failure notification in MPI calls that involve a dead process
- ULFM requires an agreement
  → All **alive** processes need to participate
- Examples: MPI_COMM_AGREE and MPI_COMM_SHRINK
Experimental setup

- Titan ORNL Supercomputer
  - 16-core AMD Opteron processors
  - Cray Gemini interconnect

- ULFM
  - OpenMPI 2.x
  - Compiled with MPI_THREAD_MULTIPLE

- One MPI rank per core

- Up to 6,000 cores

- Average of 30 times
Titan (Cray XK7); 256 MPI ranks on 256 cores; $\delta=10\times10$; ULFM MPI, uGNI/SM transports, Tuned collective module
Detection and propagation delay

Titan (Cray XK7); 1 MPI rank/core; δ=ηx10; ULFM MPI, uGNI/SM transports, Tuned collective module

Stabilization delay δ=2.5s

Latency (s) vs Number of cores

Reliable Broadcast and Congestions

Overhead% over 1 fault vs Number of cores

Agreement with Detected Failures

Latency (ms) vs Number of failures
Consensus in ULFM without fault detector

- Provided by the system
  1. Timeout: Large to avoid false positive 😞
  2. Failures detected by ORTE, which informs mpirun, which then broadcasts
     - Non-resilient binary tree structure
     - Delays on the mpirun level to start the propagation

50X improvement with failure detector 😊😊😊
Related work

- Some have a logical ring (Chord, Gulfstream, ...)
- Some separate detection and propagation (SWIM, consensus algorithms, ...)
- Most have non-deterministic strategies
  - at best: expectation of detection/propagation time for single failure
  - no quantitative assessment for several consecutive failures

- Our work is 100% deterministic
  - detection with single observer and easy-to-define time-out
  - minimal impact on failure-free execution of the application
  - logarithmic worst-case propagation
  - logarithmic worst-case repair time with consecutive failures
Did you say random?

**Failure detection**

- Periodic rounds of observation
- Need several rounds to detect with high probability 😞
- Observation round with 100,000 nodes selecting random target:
  - ⇒ expect 36,788 nodes ignored
  - ⇒ need 21 rounds for probability to miss one node \( \leq 0.000000001 \)
  - ⇒ contention with likely \( 5 \leq \#msgs-per-node \leq 15 \)
  - ⇒ 100X increase in stabilization time for one failure
- No need to maintain the ring 😊

**Information propagation**

- Flooding algorithm with randomized targets
- Hard to find criteria to stop propagation 😞
- No need to maintain any broadcast structure 😊
Did you say random?

**Failure detection**

- Periodic rounds of observation
- Need several rounds to detect with high probability 😞
- Observation round with 100,000 nodes selecting random target:
  - $\Rightarrow$ expect 36,788 nodes ignored
  - $\Rightarrow$ need 21 rounds for probability to miss one node $\leq 0.000000001$
  - $\Rightarrow$ contention with likely $5 \leq \#msgs-per-node \leq 15$

**Our take**

Good for dynamic environments (with new nodes joining, intermittent failures, unreliable routing)

Unfit for HPC platforms

- Hard to find criteria to stop propagation 😞
- No need to maintain any broadcast structure 😊
Conclusion and future work

Conclusion

- Failure detector based on timeout and heartbeats
- Tolerate arbitrary number of failures (but not too frequent)
- Complicated trade off between overhead, detection and risks (of not detecting failures)
- 100% deterministic
  - First worst-case analysis of repair time with cascading failures
  - 100X faster detection time over random rounds
- Unique implementation in ULFM
  - Negligible overhead, quick failure information dissemination
  - 50X improvement for consensus

Future work

- Failure detector service provided by MPI process manager (PMIx) instead of MPI library
- Investigate link/switch failures