Failure Detection and Propagation in HPC systems

George Bosilca[1], Aurélien Bouteiller[1], Amina Guermouche[1],
Thomas Hérault[1], Yves Robert[1,2], Pierre Sens[3] and Jack Dongarra[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 ($\approx 20mn$)
  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

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 τ
- 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

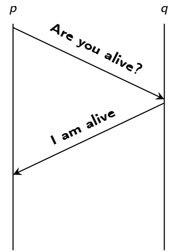
- Node = physical resource
- Process = program running on node
- Thread = part of a process that can run on a single core
- Failure detector will detect both process and node failures
- Failure detector mandatory to detect some node failures

Outline

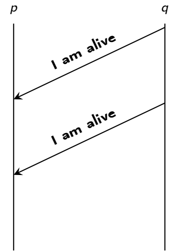
1. Model
2. Failure detector
3. Worst-case analysis
4. Implementation & experiments

Timeout techniques: p observes q

- Pull technique
  - Observer p sends a *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)



[1]: W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. IEEE Trans. Computers, 2002

Timeout techniques: platform-wide

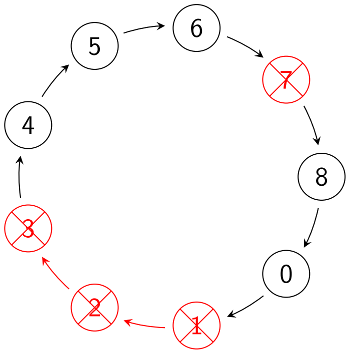
- All-to-all:
  - ☺ Immediate knowledge propagation
  - ☹ Dramatic overhead

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

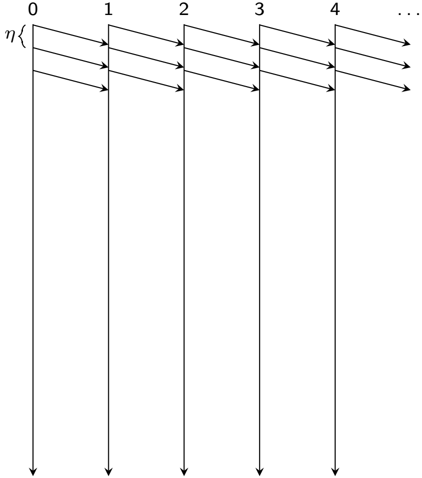
Algorithm for failure detection

- Processes arranged as a ring
- Periodic heartbeats from a node to its successor

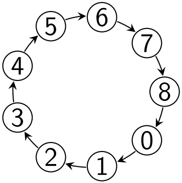
- Maintain ring of alive nodes
  - → Reconnect ring after a failure
  - → Inform all processes



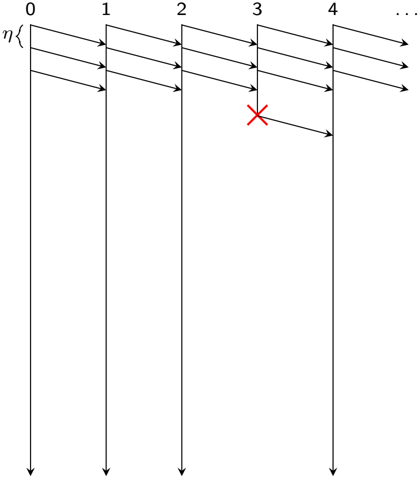
Reconnecting the ring



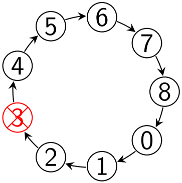
η: Heartbeat interval



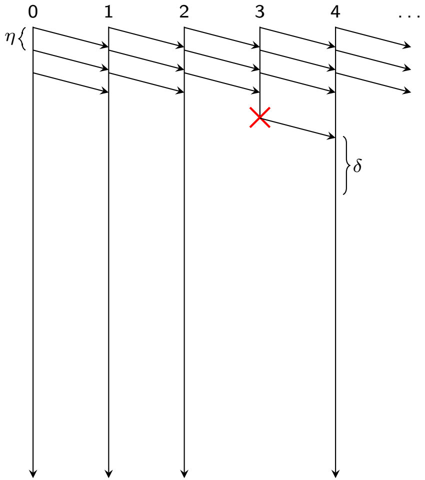
Reconnecting the ring



η: Heartbeat interval



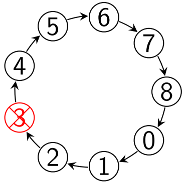
Reconnecting the ring



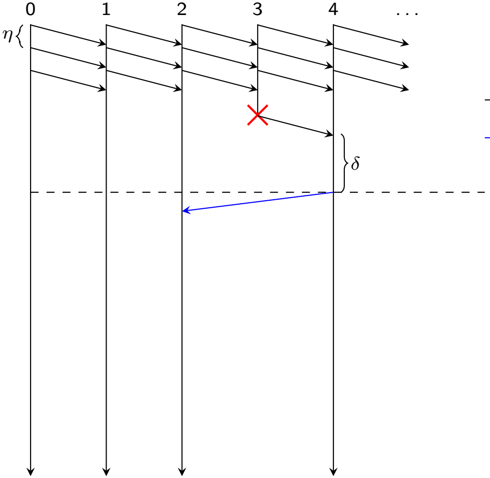
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

→ Heartbeat



Reconnecting the ring

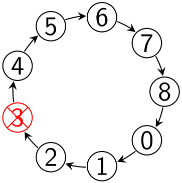


η: Heartbeat interval

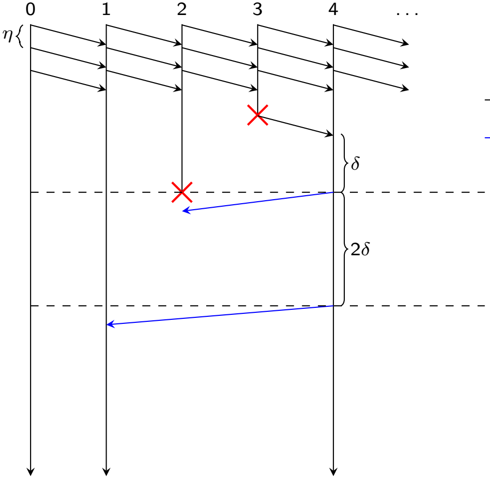
δ: Timeout, $\delta >> \tau$

→ Heartbeat

→ Reconnection message



Reconnecting the ring

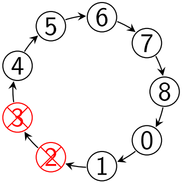


η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

Reconnection message



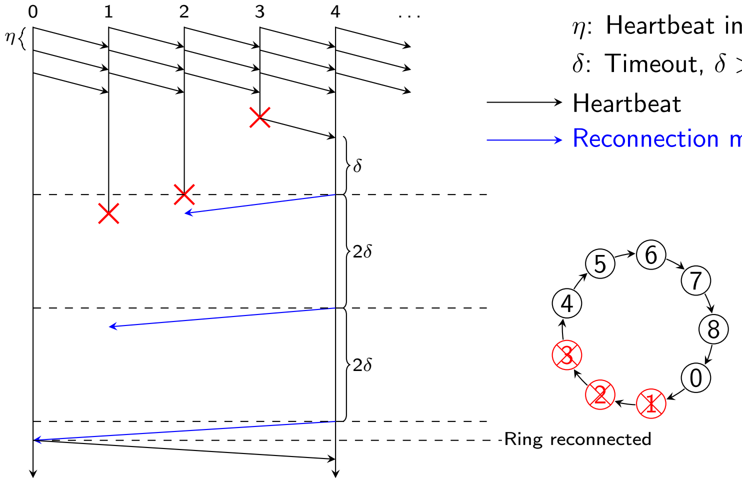
Reconnecting the ring

η: Heartbeat interval

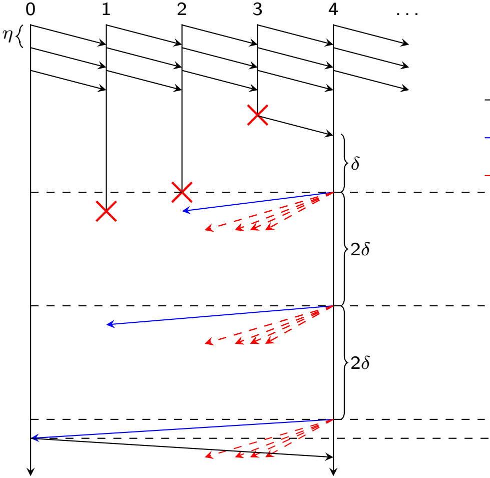
δ: Timeout, $\delta >> \tau$

Heartbeat

Reconnection message



Reconnecting the ring



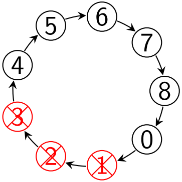
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

Reconnection message

Broadcast message



Algorithm

task Initialization
  $\texttt{emitter}_i \leftarrow (i-1) \bmod N$
  $\texttt{observer}_i \leftarrow (i+1) \bmod N$
  HB-Timeout $\leftarrow \eta$
  Susp-Timeout $\leftarrow \delta$
  $\mathcal{D}_i \leftarrow \emptyset$
end task

task T1: When HB-Timeout expires
  HB-Timeout $\leftarrow \eta$
  Send heartbeat(i) to $\texttt{observer}_i$
end task

task T2: upon reception of heartbeat($\texttt{emitter}_i$)
  Susp-Timeout $\leftarrow \delta$
end task

task T3: When Susp-Timeout expires
  Susp-Timeout $\leftarrow 2\delta$
  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \texttt{emitter}_i$
  $\texttt{dead} \leftarrow \texttt{emitter}_i$
  $\texttt{emitter}_i \leftarrow \mathit{FindEmitter}(\mathcal{D}_i)$
  Send NewObserver(i) to $\texttt{emitter}_i$
  Send BcastMsg($\texttt{dead}, i, \mathcal{D}_i$) to $\mathit{Neighbors}(i, \mathcal{D}_i)$
end task

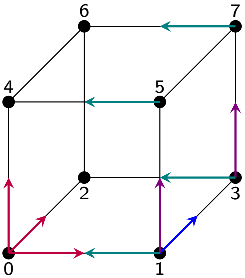
task T4: upon reception of NewObserver(j)
  $\texttt{observer}_i \leftarrow j$
  HB-Timeout $\leftarrow 0$
end task

task T5: upon reception of BcastMsg($\texttt{dead}, s, \mathcal{D}$)
  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{\texttt{dead}\}$
  Send BcastMsg($\texttt{dead}, s, \mathcal{D}$) to $\mathit{Neighbors}(s, \mathcal{D})$
end task

function $\mathit{FindEmitter}(\mathcal{D}_i)$
  $k \leftarrow \texttt{emitter}_i$
  while $k \in \mathcal{D}_i$ **do**
    $k \leftarrow (k-1) \bmod N$
  return k
end function

Broadcast algorithm

- 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)



Node	Node1	Node2	Node4
1	0	0-2-3	0-4-5
2	0-1-3	0	0-4-6
3	0-1	0-2	0-4-5-7
4	0-1-5	0-2-6	0
5	0-1	0-2-6-7	0-4
6	0-1-3-7	0-2	0-4
7	0-1-3	0-2-6	0-4-5

[1] P. Ramanathan and Kang G. Shin, 'Reliable Broadcast Algorithm', IEEE Trans. Computers, 1998

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

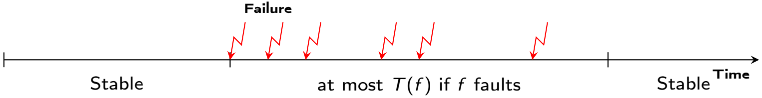
- Need fault-tolerant overlay with small fault-tolerant diameter and easy/fast one-port routing

- 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

$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{\textit{reconstruction}} + \underbrace{\frac{f(f+1)}{2}B(n)}_{\textit{broadcast}}$$

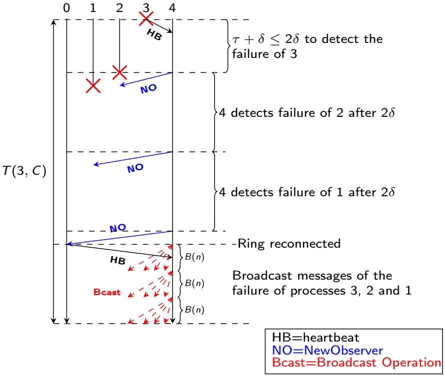
- $T(f) \leq$ ring reconstruction + broadcasts (for the proof)
- Process p discovers the death of q at most **once**
  $\Rightarrow$ $i-th$ dead process discovered dead by at most $f-i+1$ processes
  $\Rightarrow$ at most $\frac{f(f+1)}{2}$ broadcasts
- $R(f)$ ring reconstruction time
  For $1 \leq f \leq \lfloor \log n \rfloor - 1$,

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

Ring reconnection

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

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



Worst-case scenario

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

Worst-case scenario

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

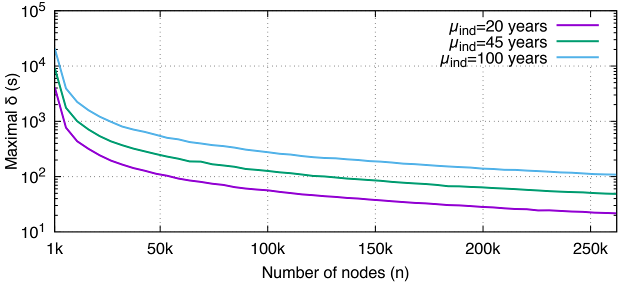
Too pessimistic!?

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 $\tau = 1\mu s$



$$\mathbb{P}\left(\geq \lfloor \log_2(n) \rfloor \text{ failures in } T(\lfloor \log_2(n) \rfloor - 1)\right) < 0.000000001$$

- 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 $\Rightarrow$ see paper!

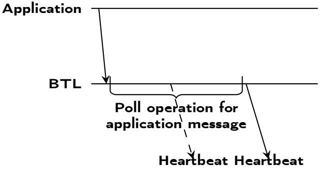
(results confirm that:
- overlapping failures are rare
- overlapping failure strike independently
- average stabilization time remains close to δ)

Outline

Implementation

- 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

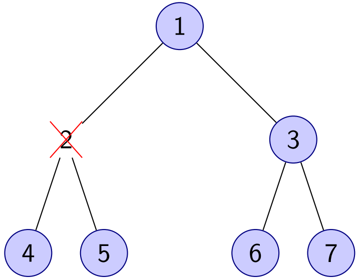


Case study: ULFM

- 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

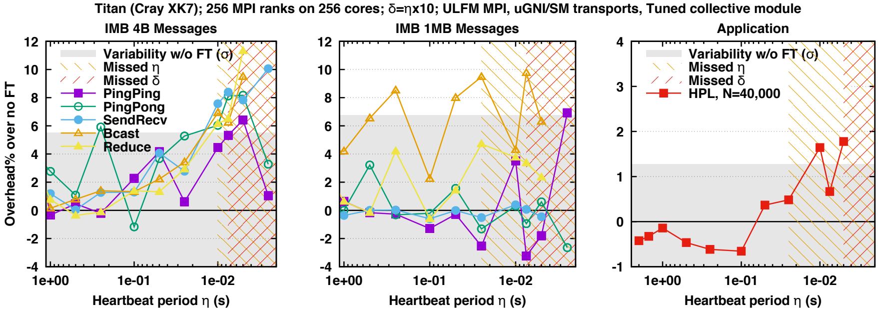
- Examples: `MPI_COMM_AGREE` and `MPI_COMM_SHRINK`



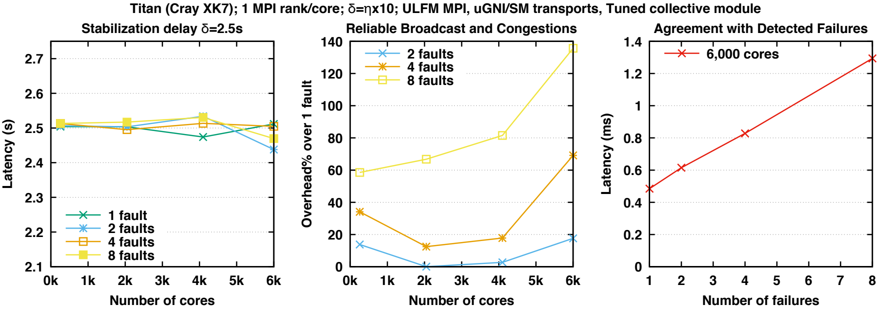
Experimental setup

- 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

Noise

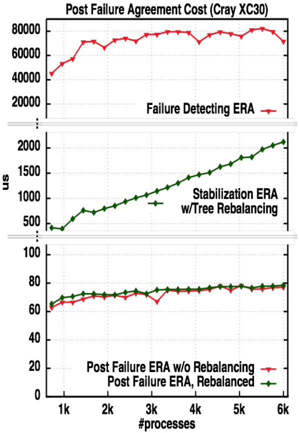

Titan (Cray XK7); 256 MPI ranks on 256 cores; δ=ηx10; ULFM MPI, uGNI/SM transports, Tuned collective module
IMB 4B Messages
IMB 1MB Messages
Application
Variability w/o FT (σ)
Missed η
Missed δ
PingPing
PingPong
SendRecv
Bcast
Reduce
HPL, N=40,000
Overhead% over no FT
Heartbeat period η (s)
1e+00
1e-01
1e-02

Detection and propagation delay



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:
  $\Rightarrow$ expect $36,788$ nodes ignored
  $\Rightarrow$ need 21 rounds for probability to miss one node ≤ 0.000000001
  $\Rightarrow$ contention with likely $5 \leq \#msgs\text{-}per\text{-}node \leq 15$
  $\Rightarrow$ $100X$ 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 ≤ 0.000000001
  $\Rightarrow$ contention with likely $5 \leq \#\textit{msgs-per-node} \leq 15$

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

  $\Rightarrow$ First worst-case analysis of repair time with cascading failures

  $\Rightarrow$ $100X$ faster detection time over random rounds
- Unique implementation in ULFM

  $\Rightarrow$ Negligible overhead, quick failure information dissemination

  $\Rightarrow$ $50X$ improvement for consensus

Future work

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