Article

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High
Performance Computing Applications
1–20

DOI: 10.1177/1094342017711505
journals.sagepub.com/home/hpc
SAGE

A failure detector for HPC platforms

George Bosilca[1], Aurelien Bouteiller[1], Amina Guermouche[2],
Thomas Herault[1], Yves Robert[1,3], Pierre Sens[4]
and Jack Dongarra[1,5,6]

Abstract

Building an infrastructure for exascale applications requires, in addition to many other key components, a stable and efficient failure detector. This article describes the design and evaluation of a robust failure detector that can maintain and distribute the correct list of alive resources within proven and scalable bounds. The detection and distribution of the fault information follow different overlay topologies that together guarantee minimal disturbance to the applications. A virtual observation ring minimizes the overhead by allowing each node to be observed by another single node, providing an unobtrusive behavior. The propagation stage uses a nonuniform variant of a reliable broadcast over a circulant graph overlay network and guarantees a logarithmic fault propagation. Extensive simulations, together with experiments on the Titan Oak Ridge National Laboratory supercomputer, show that the algorithm performs extremely well and exhibits all the desired properties of an exascale-ready algorithm.

Keywords
MPI, failure detection, fault tolerance

1. Introduction

Failure detection (FD) is a prerequisite to failure mitigation and a key component of any infrastructure that requires resilience. This article is devoted to the design and evaluation of a reliable algorithm that will maintain and distribute the updated list of alive resources with a guaranteed maximum delay. We consider a typical high-performance computing (HPC) platform in steady-state operation mode. Because in such environments the transmission time can be considered as bounded (although that bound is unknown), it becomes possible to provide a *perfect failure detector* according to the classical definition of Chandra and Toueg (1996). A failure detector is a distributed service able to return the state of any node, alive or dead (subject to a crash).[1] A failure detector is *perfect* if any node death is eventually suspected by all surviving nodes and if no surviving node ever suspects a node that is still alive. Critical fault-tolerant algorithms for HPC and implementations of communication middleware for unreliable systems rely on the strong properties of perfect failure detectors (see e.g. Bland et al., 2013a, 2013b, 2015; Egwutuoha et al., 2013; Herault et al., 2015; Katti et al., 2015). Their cost in terms of computation and communication overhead, as well as their properties in terms of latency to detect and notify failures and of reliability, have thus a significant impact on the overall performance of a fault-tolerant HPC solution. A major factor to assess the efficacy of an FD algorithm is the trade-off that it achieves between scalability and the speed of information propagation in the system.

Although we focus primarily on the most widely used programming paradigms, the message passing interface (MPI), the techniques, and algorithms proposed have a larger scope and are applicable in any resilient distributed programming environment. We consider the platform as being initially composed of N nodes, but with a high probability, some of these resources will become unavailable throughout the execution. When exposed to the death of a node, traditional applications would abort. However, the applications that we consider are augmented with fault-tolerant extensions that allow them to continue across

[1]ICL, University of Tennessee Knoxville, Knoxville, TN, USA
[2]Telecom SudParis, Évry, France
[3]LIP, École Normale Supérieure de Lyon, Lyon, France
[4]LIP6, Université Paris 6, Paris, France
[5]Oak Ridge National Lab, Oak Ridge, TN, USA
[6]Manchester University, Manchester, UK

Corresponding author:
Yves Robert, ICL, University of Tennessee Knoxville, Knoxville, TN, USA; LIP, École Normale Supérieure de Lyon, Lyon, France.
Email: Yves.Robert@ens-lyon.fr

failures (e.g. Bland et al., 2013), either using a generic or an application-specific fault-tolerant model. The design of this model is outside the scope of this article, but without loss of generality, we can safely assume that any fault-tolerant recovery model requires a robust fault detection mechanism. Our goal is to design such a *robust* protocol that can detect all failures and enable the *efficient repair* of the execution platform.

By *repairing the platform*, we mean that all surviving nodes will eventually be notified of all failures and will therefore be able to compute the list of surviving nodes. The state of the platform where all dead nodes are known to all processes is called a *stable configuration* (note that nodes may not be aware that they are in a stable configuration).

By *robust*, we mean that regardless of the length of the execution, if a set of up to f failures disrupt the platform and precipitate it into an unstable configuration, the protocol will bring the platform back into a stable configuration within $T(f)$ time units—we will define $T(f)$ later in the article. Note that the goal is not to tolerate up to f failures overall. On the contrary, the protocol will tolerate an arbitrary number of failures throughout an unbounded-length execution, provided that no more than f successive overlapping failures strike within the $T(f)$ time window. Hence, f induces a constraint on the frequency of failures, but not on the total number of failures.

By *efficiently*, we aim at a low-overhead protocol that limits the number of messages exchanged to detect the faults and repair the platform. Although we assume a fully connected platform (any node may communicate with any other), we use a realistic *one-port* communication model (Bhat et al., 2003) where a node can send and/or receive at most one message at any time step. Independent communications, involving distinct sender/receiver pairs, can take place in parallel; however, two messages sent by the same node will be serialized. Note that the one-port model is only an assumption used to model the performance and provide an upper bound for the overheads. In real situations where platforms support multiport communications, our algorithm is capable of taking advantage of such capabilities. All these goals seem contradictory, but they only call for a carefully designed trade-off. As shown in the studies by Ferreira et al. (2008), Hoefler et al. (2010), and Kharbas et al. (2012), system noise created by the messages and computations of the fault detection mechanism can impose significant overheads in HPC applications. Here, *system noise* is broadly defined as the impact of operating system and architectural overheads onto application performance. Hence, the efficiency of the approach must be carefully assessed. The overhead should be kept minimal in the absence of failures, while FD and propagation should execute quickly, which usually implies a robust broadcast operation that introduces many messages. The major contributions of this work are as follows:

- It provides a proven algorithm for FD based on a robust protocol that tolerates an arbitrary number of failures, provided that no more than f consecutive failures strike within a time window of duration $T(f)$.
- The protocol has minimal overhead in failure-free operation, with a unique observer per node.
- The protocol achieves FD and propagation in logarithmic time for up to $f_{\max} = \lfloor \log n \rfloor - 1$ where n is the number of alive nodes. More precisely, the bound $T(f_{\max})$ is deterministic and logarithmic in n, even in the worst case.
- All performance guarantees are expressed within a realistic one-port communication model.
- It provides a detailed theoretical and practical comparison with randomized protocols.
- Extensive simulations and experiments with user-level failure mitigation (ULFM; Bland et al., 2013) show very good performance of the algorithm.

The rest of the article is organized as follows: We start with an informal description of the algorithm in Section 2. We detail the model, the proof of correctness, and the time-performance analysis in Section 3. Then, we assess the efficiency of the algorithm in a practical setting, first by reporting on a comprehensive set of simulations in Section 4, and then by discussing experimental results on the Titan Oak Ridge National Laboratory (ORNL) supercomputer in Section 5. Section 6 provides an overview of related work. Finally, we outline conclusions and directions for future work in Section 7.

2. Algorithm

This section provides an informal description of the algorithm (for a list of main notations, see Table 1). We refer to Section 3 for a detailed presentation of the model, a proof of correctness, and a time-performance analysis. We maintain two main invariants in the algorithm:

1. Each alive node maintains its own list of known dead resources.
2. Alive nodes are arranged along a ring, and each node observes its predecessor in the ring. In other words, the successor/observer receives heartbeats from its predecessor/emitter (see below).

When a node dies, its observer broadcasts the information and reconnects the ring: From now onward, the observer will observe the last known predecessor (accounting for locally known failures) of its former

Table 1. List of notations.

Platform parameters	
N	Initial number of nodes
t	Upper bound on the time to transfer a message
Protocol parameters	
h	Period for heartbeats
d	Time-out for suspecting a failure

predecessor. The rationale for using a ring for detection is to reduce the overhead in the failure-free case: With only one observer, a minimal number of heartbeat messages have to be sent. We use the protocol suggested by Chen et al. (2002) for fault detection. Consider a node q observing a node p. The observed node p is also called the emitter, because it emits periodic heartbeat messages m_1, m_2, … at time s_1, s_2, … to its observer q, every h time units. Now let $s_i' = s_i + d$. At any time $t \in [s'_i, s'_{i+1})$, q trusts p if it has received heartbeat m_i or higher. Here, d is the time-out after which q suspects the failure of p. Assume there are initially N alive nodes numbered from 0 to $N - 1$, and node $i + 1 \bmod N$ observes node i according to the previous protocol, for all $0 \leq i \leq N - 1$. Tasks T1 and T2 in algorithm 1 execute this basic observation node, with the time-out delay being reset upon reception of a heartbeat. Note that Chen et al. (2002) show that this protocol, where the emitter spontaneously sends heartbeats to its observer, exhibits better performance than the variant where observers reply to heartbeat requests.

What happens when an observer (node i) suspects the death of its predecessor in the ring? Task T3 in algorithm 2 implements two actions. First, it updates the local list D_i of dead nodes with the identity of its emitter and then reconnects the ring (lines 19–23); and second, it initiates a reliable broadcast informing all nodes in its current list of alive nodes about the death of its predecessor (line 24).

The first action, namely the reconnection of the ring, is taken care of by the procedure FindEmitter(D_i): Node i searches its list of dead resources D_i and finds the first (believed) alive node, j, preceding it in the ring. It assigns j as its new emitter and sends a message NewObserver informing j that i has become its observer. Node i also sets a time-out to 2d time units, a period after which it will suspect its new emitter, j, if it has not received any heartbeat. Task T4 implements the corresponding action at the emitter side.

The second action for node i is the broadcast of the death to all alive nodes (according to its current list). A message BcastMsg(dead, i, D_i) containing the identity of the dead node dead, the source of the broadcast i, and the locally known list of dead nodes D_i is broadcast to all alive nodes (according to the current knowledge of node i). We now detail how this procedure works. Let A be the complement of D_i in $\{0, 1, \ldots, N-1\}$ and let $n = |A|$. The elements of A are labeled from 0 to $n - 1$, where the source i of the broadcast is labeled 0. The broadcast is tagged with a unique identifier and involves only nodes of the labeled list A (this list is computable at each participant as D_i is part of the message). Because n is not necessarily a power of 2, we have a complication.[2] Letting $k = \lfloor \log n \rfloor$ (all logarithms are in base 2), we have $2^k \leq n < 2^{k+1}$. We use twice the reliable hypercube broadcast algorithm (HBA) of Ramanathan and Shin (1988). The first HBA call is from the source (label 0) to the subcube of nodes j, where $0 \leq j \leq 2^k$, and the second HBA call is from the same source (label 0) to the subcube of nodes $n - j \bmod n$, where $0 \leq j \leq 2^k$. Each HBA call thus involves a complete hypercube of 2^k nodes, and their union covers all n nodes (with some overlap). The HBA algorithm delivers multiple copies of the broadcast message through disjoint paths to all the nodes in the system. Each node executes a recursive doubling algorithm and propagates the received information to up to k participants ahead of it, located at distance 2^k for $0 \leq j \leq 2^k$. For simplicity, we refer to both HBA calls as a single broadcast in our algorithm.

Upon reception of a broadcast message including a source s and a list of dead nodes D, any alive node i can reconnect the complement list A of nodes involved in the broadcast operation and their labels, and then compute the ordered set of neighbors Neighbors(s, D) to which it will then forward the message. We stress that the same list D or equivalently the same set of participating nodes, is used throughout the broadcast operation, even though some intermediate nodes might have a different knowledge of dead and alive nodes. This feature is essential to preserving fault tolerance in the algorithm of Ramanathan and Shin (1988). Indeed, we know from Ramanathan and Shin (1988) that each hypercube broadcast is guaranteed to complete provided that there are no more than $k - 1$ dead nodes within participating nodes (set A) while the broadcast executes.

3. Model and performance analysis

This section provides a detailed presentation of the model and a proof of correctness of the algorithm, together with a worst-case time-performance analysis. We also present a comparison with randomized protocols for observing processes and detecting failures.

3.1. Model

3.1.1. General framework. Nodes can communicate by sending messages in communication channels, expected to be lossless and not ordered. Any node can send a message to any other node. Messages in the communication channel (p, q) take a random time $T_{p,\ q}$ to be delivered, which has an upper bound t. We consider executions where nodes can die permanently at any time. If a node p dies, then all communication channels to p are emptied; p does not send any message nor execute any local assignment.

Note that t is a property of the platform that represents the maximal time that separates a process entering a send operation and the destination process having the corresponding message ready to read in its memory. Although the exact value for t is generally unknown, it can be bounded in our case using the techniques described in Section 5.1. The algorithm uses d - t as a bound to define the limit after which a node is suspected dead. Tuning the value of d as close as possible to t—without underestimating t to guarantee that false positives are not detected—is an operation that must be fitted for each target platform. Thus, in the theoretical analysis, we use t to evaluate the worst case of a communication that succeeds, while the algorithm must rely on d to detect a failure.

3.1.2. Using the one-port model. Although we assume a fully connected platform (any node may communicate with any other), we use a realistic one-port communication model (Bhat et al., 2003) where a node can send and/or receive at most one message at any time step. Independent communications involving distinct sender/receiver pairs can take place in parallel; however, two messages involving the same node will be serialized. Using the one-port model while aiming at a low-overhead protocol is a key motivation to this work. It is not realistic to assume that each node would observe any other node, or even a large subset of nodes. While this would greatly facilitate the diffusion of knowledge about a new death and speed up the transition back to a stable configuration, it would also incur a tremendous overhead in terms of heartbeat messages and in the end dramatically impact the throughput of the platform.

Because all messages within our algorithm have a small size, we model our communications using a constant time t to send a message from one node to another. We could have used a traditional model such as LogP or a start-up overhead plus a time proportional to the message size, but since we use this only as an upper bound, this would unnecessarily complicate the analysis. Under the one-port model, the HBA algorithm (Ramanathan and Shin, 1988) with 2^k nodes executes in $2k$t, provided that no more than k-1 deaths strike during its execution. The time for one complete broadcast algorithm in algorithm 1 would then be (upper bounded by) 4t log n in the absence of any other messages, since we use two HBA calls in sequence. But our algorithm also requires heartbeats to be sent along the ring, as well as NewObserver messages when ring reconnection is needed. Assuming that h2 3t (where h is the heartbeat period), we can always insert broadcast and NewObserver messages in between two successive heartbeats, thereby guaranteeing that a broadcast in algorithm 2 will always execute within $B(n)$= 8t log n, assuming no new failure interrupts the broadcast operation.

Algorithm 1. Sketch of the failure detector for node *i*.

```
task Initialization
    emitter_i     (i – 1) mod N
    observer_i     (i – 1) mod N
    HB-Timeout   h
    Susp-Timeout   d
    D_i    [
end task

task T1: When HB-Timeout expires
    HB-Timeout   h
    Send heartbeat(i) to observer_i
end task

task T2: upon reception of heartbeat(emitter_i)
    Susp-Timeout   d
end task

task T3: When Susp-Timeout expires
    Susp-Timeout   2d
    D_i D [ _i emitter_i
    dead    emitter_i
    emitter_i    FindEmitter(D _i)
    Send NewObserver(i) to emitter_i
    Send BcastMsg(dead, i, _i D) to Neighbors(i, _i) D
end task

task T4: upon reception of NewObserver(j)
    observer_i     j
    HB-Timeout    0
end task

task T5: upon reception of BcastMsg(dead, s, D)
    D_i D [ _i {dead}
    Send BcastMsg(dead, s, ) D to Neighbors(s, ) D
end task

function FindEmitter(D)
    k    emitter_i
    while k 2 D_i do
        k    (k – 1) mod N
    return k
end function
```

3.1.3. Stable configuration and stabilization time. Here we consider executions that, from the initial configuration, reached a steady state before a failure hit the system and made it leave that steady state. To prove the correctness of our algorithm, we show that in a given time the system returns to a steady state, assuming that no more than a bounded number of failures strike during this time.

Connected node. A node p is *connected with its successor* in a configuration, if p is alive and $\mathrm{emitter}_p$ is the closest predecessor of p that is alive (on the ring). It is *connected with its predecessor* if it is alive, and $\mathrm{observer}_p$ is the closest successor of p that is alive in that configuration. It is *reconnected* if it is connected with both its successor and predecessor. If all processors are reconnected, we say the ring is reconnected.

Stable configuration. A configuration C is the global state of all processes plus the status of the network. A configuration is declared as *stable*, if any alive node p is reconnected in C and for any node q, q2 D$_p$ ¸ q is dead in C.

Stabilization time. $T(f)$, with f being the number of overlapping failures, is the duration of the longest sequence of nonstable configurations during any execution, assuming at most f failures during the sequence.

3.2. Correctness and performance analysis

The main result is the following proof of correctness that provides a deterministic upper bound on the *stabilization time* $T(f)$ of the algorithm with at most f overlapping faults:

Theorem 1. With n S N alive nodes, and for any f S blog nc- 1, we have

$$T(f)\ \mathrm{S}\ f(f+1)\mathrm{d}+f\mathrm{t}+\frac{f(f+1)}{2}B(n) \qquad \text{ð1Þ}$$

where $B(n)$= 8t log n.

This upper bound is pessimistic for many reasons, which are discussed after the proof. But the key point is that the algorithm tolerates up to blog nc − 1 overlapping failures in logarithmic time $O\left((\log n)^3\right)$.

Proof. Starting from a nonstable configuration, the next stable configuration will be reached when (i) all nodes are informed of the different failures via the broadcast and (ii) processes of the ring are reconnected. Recall that every time a node has detected a failure, it initiates a broadcast that executes within $B = B(n)$= 8t log n time units and that is guaranteed to reach all alive nodes as long as fS blog nc- 1. Because we interleave reconnection messages within the broadcast, B encompasses both the broadcast and the reconnection. However, due to the one-port model, we cannot assume anything about the pipelining of several consecutive broadcast operations. In this proof, we make a first simplification by overapproximating $T(f)$ as the maximum time $R(f)$ to reconnect the ring after f overlapping failures, plus the time to execute all the broadcasts that were initiated, in sequence (assuming no overlap at all). We prove an upper bound on $R(f)$ by induction, letting $R(0)=0$:

Lemma 1. For 1 S f S blog nc- 1, we have

$$R(f)\ \mathrm{S}\ R(f-1)+2f\mathrm{d}+\mathrm{t} \qquad \text{ð2Þ}$$

Proof. We first prove equation (2) when $f = 1$. Assume that node p, observed by node q, fails. After receiving the last heartbeat, q needs d time units to detect the failure (line 2 of algorithm 2). Thus, the worst possible scenario is when p fails right after sending a heartbeat, which will take t time units to reach q. Thus, q detects the failure after t + d time units. Finally, q sends the reconnection message to the predecessor of p, which will take t, hence $R(1)$S2t + d. We keep the overapproximation $R(1)$St + 2d to simplify the formula in the general case.

Assume now that equation (2) holds for all f S log n 2. Now consider an execution with $f+1$ overlapping failures, the first of them striking at time 0 (see Figure 1). The $(f+1)$-th failure strikes at time t. Necessarily, tS$R(f)$; otherwise, the ring would have been reconnected after f failures, and the last one would not be overlapping. There are f dead nodes just before time t among the original n alive nodes, which define k S f segments I_i, 1 S i S k. Here, segment I_i is an interval of d_i 2 1 consecutive dead nodes (see Figure 1). Of course, $\overset{\mathrm{P}}{\underset{i=1}{}}\ d_i = f$, and there remain $n-f$ alive nodes. There are multiple cases depending upon which node is struck by the $(f+1)$th failure at time t:

- The new failure strikes a node that is neither a predecessor nor a successor of a segment (e.g. the

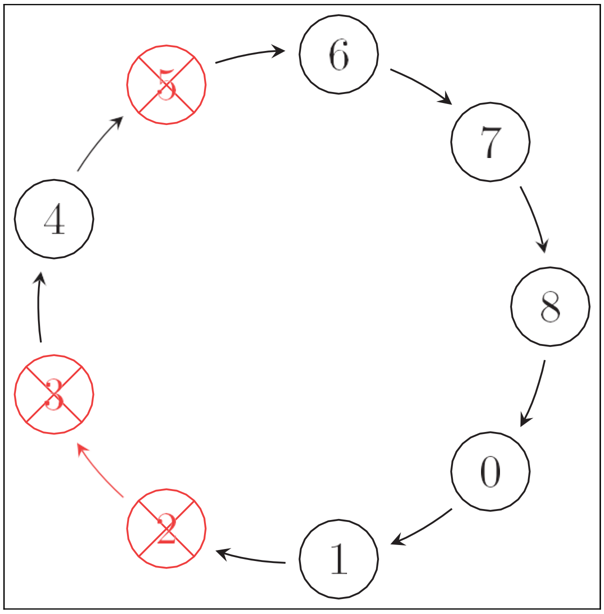


Figure 1. Segments of dead nodes after $f=3$ failures: $n=9$, $k=2$, $I_1 =$ f2, 3g, $I_2 =$ f5g, $d_1 = 2$, and $d_2 = 1$.

failure strikes node 7 in Figure 1). In that case, a new segment of length 1 is created, and the ring is reconnected at time $t+R(1)$.

- The new failure strikes a node p that precedes a segment I_i. Let q be the successor of the last dead node in I_i. By definition, q 6p. There are two subcases:

– (i) The predecessor ¼p^0 of p is still alive (e.g. the failure strikes node 1 preceding segment I_1 in Figure 1, $q=4$ and $p^0=0$ is alive). Then, the size of segment I_i is increased by one. In the worst case, q is not aware of the death of any node in I_i at time t and needs to probe all these nodes one after the other before reconnecting with p^0 (in the example, $q=4$ needs to try to reconnect with 2 and 1 since it is not aware of their death). This costs at most $(d_i+1)(2\mathsf{d})+\mathsf{t}\ \mathsf{S}\ 2(f+1)\mathsf{d}+\mathsf{t}$, because $d_i+1\mathsf{S}f+1$, hence the ring is reconnected at time $t+2(f+1)\mathsf{d}+\mathsf{t}$.

– (ii) The predecessor p^0 of p is dead (e.g. the failure strikes node 4 preceding segment I_2 in Figure 1, $q=6$ and $p^0=3$ is dead). Then, p^0 is the last node of another segment I_j. In that case, segments I_i and I_j are merged into a new segment of size $d_i+d_j+\mathsf{S}f+1$. Just as before, in the worst case, q is not aware of the death of any node in that new segment, and the reconnection costs at most $(d_i+d_j+1)(2\mathsf{d})+\mathsf{tS}2(f+1)\mathsf{d}+\mathsf{t}$ (for an illustration, see Figure 1). Hence, the ring is reconnected at time $t+2(f+1)\mathsf{d}+\mathsf{t}$.

- The new failure strikes a node p that follows a segment I_i. Let q be the successor of p. If q is alive, it now follows a segment of size d_i+1. If q is the first dead node of segment I_j, let r be the node that follows I_j. Now r follows a segment of size d_i+d_j+1. In both cases, we conclude just as before.

This completes the proof of Lemma 1.

From Lemma 1, we easily derive by induction that

$$R(f)\ \mathsf{S}\ f(f+1)\mathsf{d}+f\mathsf{t}$$

for all values of fSlog n -1. During the ring reconnection, processes that discover a dead process initiate a broadcast of that information. We need to count, in the worst case, how many broadcasts are initiated to compute how long it takes for the information to be delivered to all nodes.

Lemma 2. Let p_i, lS iS fS blog nc- 1 be the ith process subject of a failure. In the worst case, at most $f-i+1$ processes can detect the death of p_i.

Proof. A process p is discovered dead by process q in task T3, if $\mathtt{emitter}_q=p$. In that case, p is added to D_q, and $\mathtt{emitter}_q$ is recomputed using *FindEmitter*. That function cannot return any process in D_q, and p is never removed from $_q$. Thus $\mathsf{D}q$ will never discover the death of p again. As long as q lives, no other process q^0 will execute the task T3 with $\mathtt{emitter}_{q^0}=p$, because q is an alive process between q^0 and p in the ring. Thus, q must fail after p, for p to be discovered once more. Since there are at most f faults, p_i, the ith dead process can thus be discovered dead by at most f i– + 1 processes.

We immediately have that:

Corollary 1. At most $\mathbf{P}_{i=1}^{f}(f-i+1)=(f(f+1))=2$ broadcasts are initiated.

Finally, the information on the f dead nodes must reach all alive nodes. For each segment I_i, there is a last failure after which the broadcast initiated by the observing process is not interrupted by new failures. That broadcast operation thus succeeds in delivering the list of newly discovered dead processes to all others ð$_i$ lSg n 1c-InÞ the worst case, that broadcast operation is the last to complete. As already mentioned, we conservatively consider that all the broadcast operations execute in sequence, and since there are at most $(f(f+1))=2$ broadcast operations initiated (Corollary 1), we derive that

$$T(f)\ \mathsf{S}\ R(f)+\frac{f(f+1)}{2}B(n)$$

which leads to the upper bound in equation (1) and concludes the proof of theorem 1.

We derive from lemma 2 that at most $\mathbf{P}_{i=1}^{f}(f-i+1)=(f(f+1))=2$ broadcasts are initiated. Finally, the information on the f dead nodes must reach all alive nodes. For each segment I_i, there is a last failure after which the broadcast initiated by the observing process is not interrupted by new failures. That broadcast operation thus succeeds in delivering the list of newly discovered dead processes to all others ð$_i$ Slg n lc-. Þn the worst case, that broadcast operation is the last to complete. As already mentioned, we conservatively consider that all the broadcast operations execute in sequence. Since there are at most $(f(f+1))=2$ broadcast operations initiated, we obtain $T(f)\mathsf{S}R(f)+((f(f+1))=2)B(n)$, which leads to the upper bound in equation (1) and concludes the proof of theorem 1.

The bound on $T(f)$ given by equation (1) is quite pessimistic. We can identify three levels of complexity with their corresponding bounds on $T(f)$. In the most likely scenario, where the time between two consecutive faults is larger than $T(1)$, the system has time to return to a stable configuration before the second fault, in which case all faults can be considered as independent, and the average stabilization time is $T(1)=R(1)+B(n)=O(\log n)$. If the system suffers quickly overlapping faults, the location of impacted nodes becomes important. However, the larger the platform, the smaller the probability that successive faults strike

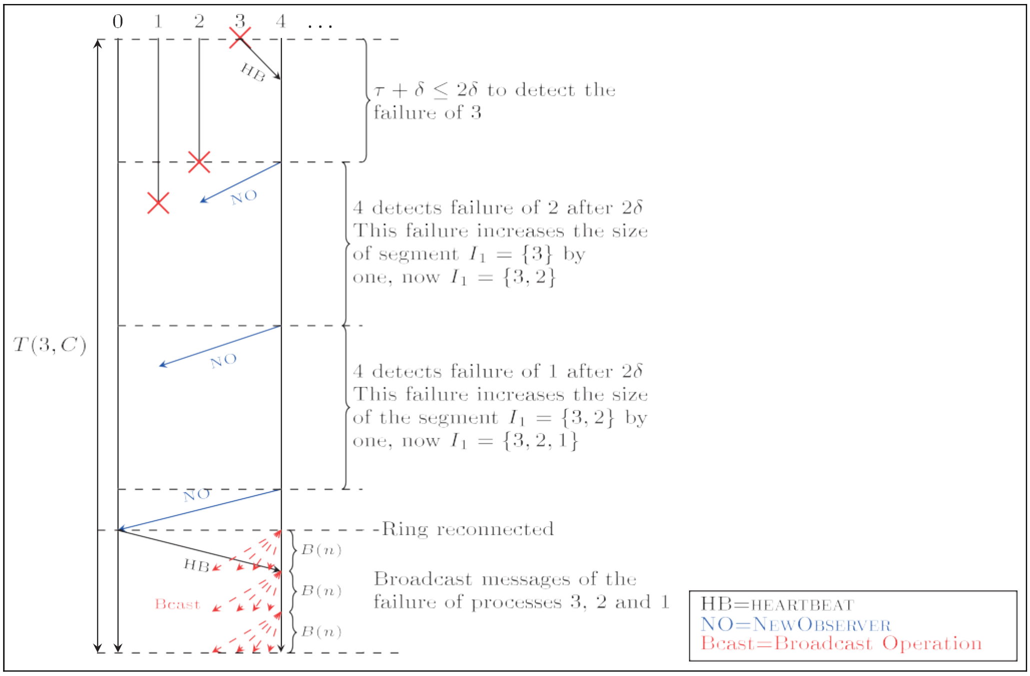

0 1 2 3 4 ...
HB
$\tau+\delta \le 2\delta$ to detect the failure of 3
NO
4 detects failure of 2 after 2δ
This failure increases the size of segment $I_1 = \{3\}$ by one, now $I_1 = \{3, 2\}$
$T(3, C)$
NO
4 detects failure of 1 after 2δ
This failure increases the size of the segment $I_1 = \{3, 2\}$ by one, now $I_1 = \{3, 2, 1\}$
NO
-Ring reconnected
$B(n)$
HB
$B(n)$
Bcast
Broadcast messages of the failure of processes 3, 2 and 1
$B(n)$
HB=HEARTBEAT
NO=NEWOBSERVER
Bcast=Broadcast Operation

Figure 2. From stable configuration C, growing segment I_1 of Figure 1: first failure on node 3, next two failures striking its ring predecessors.

consecutive nodes ($2/n$, where n is the number of alive nodes). Thus, on large platforms, overlapping failures are more likely to strike nonconsecutive nodes in the ring. If overlapping faults hit nonconsecutive nodes rapidly (i.e. faster than the time needed by the system to reach the next stable configuration), each error is detected once, but due to the one-port model, the upper bound on $T(f)$ becomes $R(1) + fB(n) = O\left(\log^2 n\right)$. Finally, in the unlikely scenario where f quickly overlapping faults hit f consecutive nodes in the ring, theorem 1 provides the upper bound for $T(f) \le R(f) + (f(f+1)/2)B(n) = O\left(\log^3 n\right)$.

Remark about stabilization time: $f/T(f)$ is the maximum number of faults per time unit that the algorithm can tolerate to guarantee that we pass by a stable configuration infinitely often. However, $T(f)$ is not a period to optimize: $T(f)$ is just the time it takes, in the worst case, after f failures, for the ring to be reconnected, and the failure information to be propagated to all alive nodes.

3.3. Nonstabilization risk control

To guarantee convergence within $T(f)$ time units, algorithm 2 assumes that $f \le \lceil\log(n)\rceil - 1$. In order to evaluate the risk behind this assumption, consider that failures strike following an Exponential distribution of parameter λ. Let $P_T(f)$ be the probability of the event ‘‘more than f failures strike within time T.’’ Then,

$$P_T(f) = 1 - \sum_{k=0}^{f} \left((\lambda T)^k / k!\right) e^{-\lambda T}.$$

Consider a platform of n nodes: if μ_{ind} is the Mean Time Between Failures (MTBF) of a single node, then $\lambda = n/\mu_{ind}$ (Hérault and Robert, 2015). Let $M = \lceil\log(n)\rceil - 1$, the assumption that there will not be more than M failures before stabilization is then true with probability $1 - P_{T(M)}(M)$. In Figure 2, we represent this relation by showing the upper bound of δ to enforce $P_{T(M)}(M) \le 10^{-9}$, at variable machines scale (n), and for different values of μ_{ind}, with a message time bound of $\tau = 1$ ms. Figure 2 illustrates that for all values of δ lower than the bound shown for a given system size and individual node reliability, the probability that failures strike fast enough to prevent algorithm 2 from converging in $T(f)$ is negligible (less than 0.000000001). As already mentioned, this bound on δ is a loose upper bound, because the bound on $T(f)$ in equation (1) is loose itself. Furthermore, it captures the risk that enough failures would strike during stabilization time to make the appearance of the worst-case scenario *possible*, even though this worst-case scenario has itself a very low probability of happening (as shown in Sections 4 and 5). Still, for the largest platforms with $n = 256,000$ nodes, we find that $\delta \le 22$ s for the most pessimistic $\mu_{ind} = 20$ years, and $\delta \le 60$ s if $\mu_{ind} = 45$ years results in timely convergence. With such large values, the detector generates negligible noise to the applications, as shown in Section 5.3.

3.4. FD with randomized protocols

In this section, we provide a comparison of our algorithm with randomized protocols such as SWIM (Das et al., 2002; Gupta et al., 2001; Snyder et al., 2014). We first provide some background in Section 3.4.1, and then proceed to a detailed comparison in Section 3.4.2

3.4.1. Background. FD techniques based on randomized protocols detect failures through periodic *observation rounds*. Within a round, each node randomly chooses another node to observe. This entails the observer sending an *are you alive?* message to the observed node and waiting for its answer. This pull technique (also called pinging) is different from the push technique based on heartbeats (Chen et al., 2002) and used in the deterministic algorithm of Section 2. Pinging is inherent to randomly choosing the observed process because this latter process does not know in advance whom to send its alive message to. Pinging is known to be less efficient than using heartbeats (Chen et al., 2002) because it requires twice as many messages and leads to increasing the time-out, to accommodate for a round-trip message.

In fact, actual protocols such as SWIM, which stands for *Scalable Weakly-consistent Infection-style Process GroupMembership* (Das et al., 2002; Gupta et al., 2001; Snyder et al., 2014; for more details, see Section 5.4), request that after a time-out, other processors are required by the observer, say P_i, to ping the nonresponding process, say P_j. Specifically, k randomly chosen processors (for details on how to choose k, see Das et al., 2002; Gupta et al., 2001) would ping P_j on behalf of P_i and forward any answer back to P_i. Only after this confirmation step would P_j's death become suspected. This confirmation step is not needed in our framework, since we assume that network links are reliable and we place an upper bound, **t**, on the time to transmit a message.

A single observation round is not enough to detect failures with high probability. During a round, some nodes will not be observed, while other nodes will receive many *are you alive?* messages from different observers, and will need to answer them all. Setting the value of the time-out then becomes a complicated task: indeed, to avoid false positives (alive nodes unduly suspected of death), one has to account for the maximum number of *are you alive?* messages that are received by the same node. The next section proposes a simplified analysis of the number of rounds and time-out values needed to limit the risk of such false positives.

Finally, just as with our algorithm, after detecting a failure, the knowledge of that failure must be propagated to every alive node. In a nutshell, this propagation can be done in many ways, including a reliable diffusion mechanism similar to the one presented in this article. Other solutions include using a gossip mechanism flooding the network in logarithmic time or piggybacking *are you alive?* messages with the current knowledge of all dead processes (for details, see Katti et al., 2015).

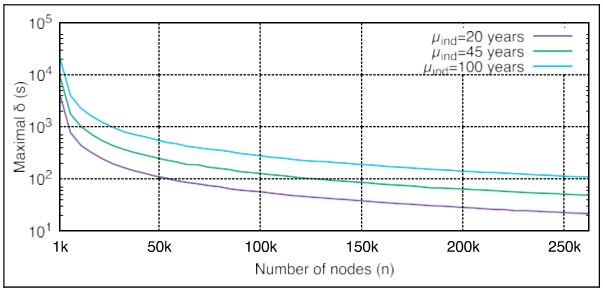


Figure 3. Maximal value for d to ensure that $P_{T(M)}(M)$╲10^{-9} with **t** = 1 ms and $M =$ b $\log_2(n)$c.

3.4.2. Comparison. In this section, we estimate the FD time for a randomized protocol. We assume a platform with $N = 100,000$ nodes and fix the risk of missing the death of a node to 10^{-9}. Note that this is the same value as the risk $P_{T(M)}(M)$ used in Section 3.3; however, our deterministic algorithm detects a single failure with probability 1, as long as the time-out value d is correctly set. On the contrary, the worst-case detection time of a randomized protocol is infinite, by construction: There are some (very unlikely) scenarios in which a dead node will never be pinged (Figure 3).

Consider a single observation round with $N = 100,000$ nodes. The probability that a given node is not pinged is $p(N) = ((N - 1)$=$N)^{N-1}$ ʼ 0:367881. In fact lim$_N$! ʻ $p(N) = 1$=e, where e is the Euler constant, and $(1$=$e)$ ʼ 0:367894. The expected number of nodes that are not pinged within a round tends to N/e. With $N = 100,000$, expect that 36,788 nodes will be ignored. Then how many rounds are needed to guarantee that all nodes are pinged with probability $1 - 10^{-9}$? The solution is x where $p(N)^x = 10^{-9}$, and by deriving, we obtain x ʼ 20:7, so that 21 rounds are needed to achieve the desired probability.

We now have to account for contention within a round. As already mentioned, some nodes will not be pinged, while some others will be pinged several times. What is the largest number $L(N)$ of ping messages that a node will receive? Of course, the largest number is $L(N) = N$-1 if all nodes ping the same one, but this is very unlikely, and we need to estimate $L(N)$ with high probability. The problem can be modeled as a balls and bins problem (Mitzenmacher and Upfal, 2005), where we throw N balls into N bins randomly and independently. The only difference is that a given node does not ping itself, but this does not modify the analysis. It is known (Mitzenmacher and Upfal, 2005: Ch 5) that (ln N = ln ln N) S $L(N)$ S 3(ln N = ln ln N) with high probability $1 -$ (1=N). Here we obtain 4:7 S L (100, 000) S 14:1, so we need to account for at least five ping messages being possibly sent to the same node (simulations in Section 4.3 show that in fact we need to account for up to 11 ping messages to be on the safe side).

Altogether, this calls for multiplying the time-out for a round-trip message by (at least) 5 to account for contention, and then by 21 to account for the number of rounds, leading to a 100 Ʒ increase. Altogether, with $N = 100, 000$, we conclude that detection can be achieved with probability 10^{-9} only with a huge time-out of magnitude two orders higher than that of our deterministic algorithm.

4. Simulations

We conduct simulations and experiments to evaluate the performance of the algorithm under different execution scenarios and parameter settings. We instantiate the model parameters with realistic values taken from the literature. The code for all algorithms and simulations is publicly available[3] so that interested readers can build relevant scenarios of their choice. In this section, we report simulation results (for experiments, see Section 5).

4.1. Simulation settings

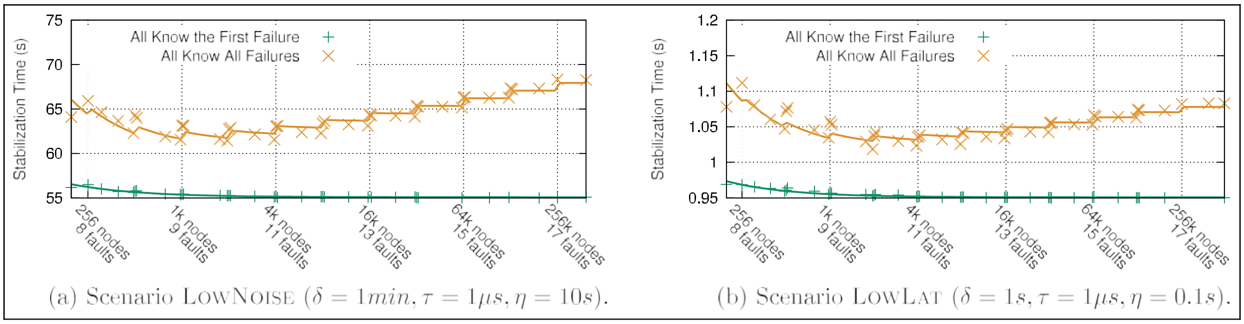
The discrete-event simulator imitates how the protocol of algorithm 2 would behave on a distributed machine of size n. Messages between a pair of alive nodes in this machine take a uniformly distributed time in the interval (0, t]. Failures are injected following an exponential law of parameter l $=n$=m$_{\text{ind}}$ (see Section 3.3). To generate a manageable amount of events, each heartbeat message and the corresponding time-outs are not simulated, but the simulator asserts that a time-out should have expired on the observer after the death of its emitter if the observer is alive at that time; otherwise, the observer's observer is going to react, following the protocol. The simulator computes (i) the average time to reach a stable configuration (all processes know all faults) starting from a configuration with a single failure injected at time 0, (ii) the average time to reach a configuration where all processes know about the initial failure, and (iii) the average number of failures striking during the time it takes to reach a stable configuration over a set of 10,000 independent runs.

We consider two main scenarios for the simulations. In both scenarios, we target a large-scale machine (up to 256,000 computing nodes) with a low-latency interconnect (t = 1 ms). In the scenario LowNoise, we set the failure detector so as to minimize the overhead in the failure-free case: h is set to 10 s and d to 1 min. We consider this case significant for platforms where nodes are expected to be reliable, or where alternative methods to detect most failures exist; the heartbeat mechanism is then used as a last resort solution (e.g. when special hardware providing a Baseboard Management Controller and controlled through a protocol like intelligent platform management interface is connected to the application notification system; Wung, 2009). We also considered a scenario LowLat, with the opposite assumptions, where active check through heartbeats is the primary method to detect failures, and a low latency of detection is required for the application: h = 0:1 s, and d = 1 s.

4.2. Simulation results

In Figure 4, we force the simulator to inject the maximum number of failures tolerated by the algorithm for a given platform size ðb log$_2$(n)c - þ in a very short time, smaller than d, in order to evaluate the average stabilization time in the most volatile environment. Varying the system size (n), and the number of injected failures simultaneously, we evaluate the time taken for the first failure to be notified to all processes and for all the processes to be notified of all the failures that struck since the last stable configuration.

The figure considers scenario LowNoise. Points on the graph show times reported by the simulator, while lines represent functions fitted to these points, Oð(1=n)+ bog$_2$(n)c þor *all know all failures* (orange lines) and $O(1/n)$ for *all know the first failure* (green lines).



(a) Scenario LowNoise ($\delta = 1min, \tau = 1\mu s, \eta = 10s$).

(b) Scenario LowLat ($\delta = 1s, \tau = 1\mu s, \eta = 0.1s$).

Figure 4. Average stabilization time, when the maximal number of failures strike a platform of varying size.

On average, the first failure, striking at time 0, is detected d –(h=2) seconds later, and this is the observed baseline for detecting the first failure at all nodes. The reliable broadcast overhead in this case is negligible, because t≪d and h. There are a few executions in which, within the first d seconds, another failure hits the observer of the first failure, introducing another d delay to actually detect the first failure and broadcast it. As the size of the machine increases, this probability decreases. Such overlapping failure cases contribute to a longer detection and notification time that can be fitted with a function inversely proportional to the platform size, but have a low probability to happen, introducing a measurable but small overhead at small scale. For general stabilization, where all processes need to know all failures, the reliable broadcast remains as fast as for the initial failure. However, if any failure strikes before that broadcast phase is complete, this delays reaching stabilization by another d followed by a logarithmic phase. As we observe in both figures, this shows at large scale, where failures have a high probability of striking successively, each introducing a constant overhead. The fitting function thus shows the same *inversely proportional* property in the beginning, and then the logarithmic behavior starts to dominate at large scale.

We conducted the same set of simulations on the LowLat scenario. The evaluation presents the exact same characteristics, shifted by the ratio between the two values for d.

We then consider the average case, when failures are not forced to strike quasi-simultaneously. We set the MTBF of independent components to a very pessimistic value ($m_{ind} = 1$ year), making the MTBF of the platform decrease to a couple of minutes at 256,000 nodes. Although we do not expect such a pessimistic value in real platforms, we evaluate this case in order to ensure that failures may occur before the initial one is detected and broadcast (or stabilization would be reached immediately after). Figure 5 presents the average number of failures observed at different scales, the average time for all nodes to know about the first failure, and the average time for all nodes to know about all failures. Points represent values given by the simulator, while lines represent fitting functions: $O(1)$ for the time for all to know the first failure, $O(n)$ for the average number of failures and the average time for all to know all failures. We present here the scenario LowNoise, although the result also holds for scenario LowLat, at a different scale.

This figure shows that, on average, and even with extremely low MTBFs, the probability that two independent failures hit the system in an overlapping manner—before the first failure is known by all nodes—is very low. This happens when the MTBF of the system becomes comparable to d. In that case, the first failure still takes close to a constant time to be notified to all. The reason is that t $\log_2(n)$ remains very small compared to d, and once the broadcast is initiated, it completes in t $\log_2(n)$. The successive failures may strike anytime between ½0, d], delaying the time to reach the stable configuration by another d + t $\log_2(n)$. On average, at 256,000 nodes, this happens in the middle of the initial FD interval, delaying the completion by d/2. Each failure, however, is independent in that case, and each is detected almost d time units after it strikes.

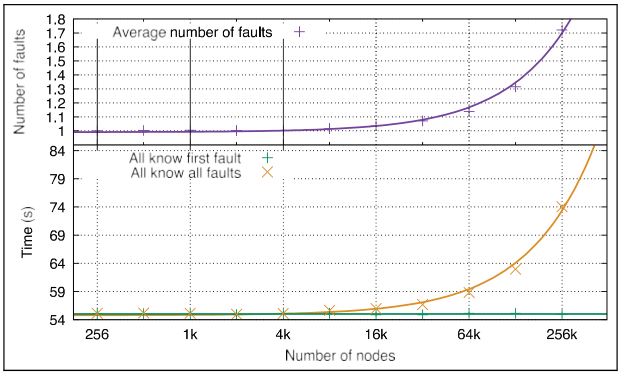


Figure 5. Average stabilization time, with random overlapping failures in scenario LowNoise (d = 1 min, t = 1 ms, h = 10 s), with $m_{ind} = 1$ year.

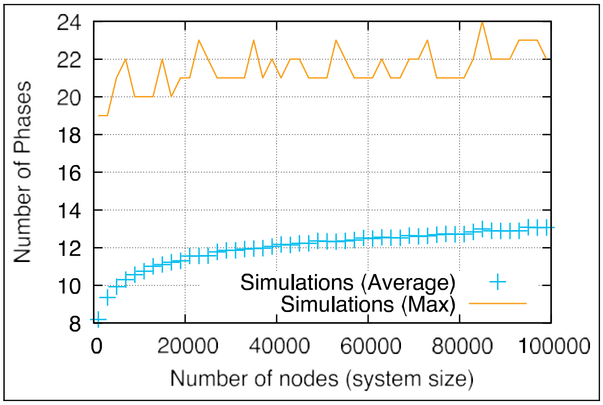


Figure 6. Number of random probing rounds to ensure that all nodes have been detected by the randomized pinging protocol.

4.3. Comparison with randomized protocols

Finally, in Figures 6 and 7, we use the discrete event simulation to expose the quality of detection with randomized gossiping protocols, such as SWIM. As described in Section 3.4.1, these protocols execute successive rounds. During a round, a process randomly selects another one to ping and uses a push mechanism to check if this selected process is still responsive. Determining when a failure will be detected with such

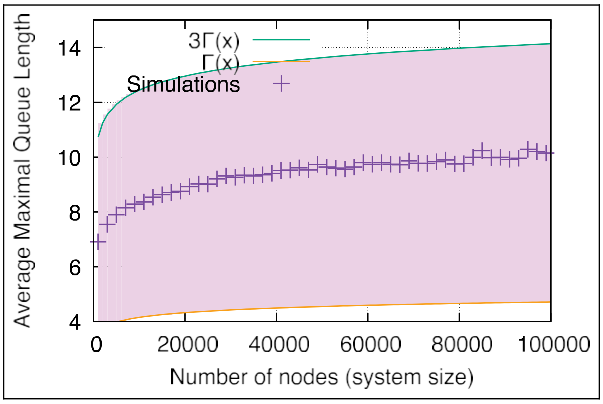


Figure 7. Average of the maximum length of queue size $L(N)$ during a single round, for the randomized pinging protocol.

an approach is more subtle than for the deterministic pull algorithm that we presented in this work. As mentioned in Section 3.4.2, there are two main reasons for this:

- The duration of a round must be higher than the value of the time-out to detect a failure. That value is a function of the network latency and of the number of heartbeat requests messages that a single target can receive during a single round. Otherwise, a process might suspect another one falsely after it did not receive a heartbeat in time, not because the target is not responsive but because it is busy responding to other ping requests.
- The number of rounds to ensure (with high probability) that all processes have been probed needs to be determined. As processes select targets independently, it is predictable that two (or more) processes select the same target and thus—as each selects a single target—that some processes are not observed during a single round.

To quantify these two parameters experimentally—in complement of the theoretical study of Section 3.4.2—we have simulated the behavior of a the probing part of a randomized gossiping protocol like SWIM. We report the following two critical measures:

- During a single probing round, where each alive process selects a single target randomly and requests for a heartbeat, what is the maximal number of processes—denoted as $L(N)$ in Section 3.4.2, where N is the number of processes—that select the same target? Figure 7 gives average values for $L(N)$.
- Figure 6 shows how many rounds are needed to ensure (with high probability) that all processes have been targeted at least once. In both figures, we scale the system size, increasing the number of nodes, and report these average and maximum numbers over 10,000 simulations per parameter.

In theory, the average largest number of processes that select the same target during a single round is between $(\ln N = \ln \ln N)$ and $3(\ln N = \ln \ln N)$, where N is the number of processes (see Section 3.4.2). Simulations of Figure 7 are consistent with these bounds, showing that up to 11 processes have a high probability to select the same target during a random round at 100,000 nodes and up to 8 processes for a system of 20,000 nodes. This means that the time-out for the heartbeat must be set to 16–22 times the maximum network latency, to ensure that a nonresponsive process has indeed failed, rather than being too congested by messages to answer to the request in time. Comparatively, our solution deterministically ensures that only one process will be pinged by another, thereby eliminating the queue management pressure.

Similarly, Figure 6 shows that on average, between 11 rounds at 20,000 nodes and 13 rounds at 100,000 nodes are necessary to ensure, with high probability, that all nodes are targeted by at least one other. If one considers the worst case, over the 10,000 simulations considered, it is often necessary for at least one of these executions to wait until 22 rounds are executed to reach all processes.

Combining both factors, to detect with high probability the failure of a single process in a system of 100,000 elements, on average, 13 rounds of 22 times the maximum network latency each would be necessary. During this time, on average, 2,599,974 messages would have been exchanged over the network. This is in stark contrast with the ring algorithm presented in this article, which provides a deterministic bound function of the number of failures. It would detect the failure in one maximum network latency and see 99,999 heartbeat messages (one per alive process).

5. Experimental evaluation

This section presents an experimental evaluation of an operational implementation of the proposed failure detector on the Titan ORNL supercomputer. We have implemented the FD and propagation service in the reference implementation of the ULFM draft MPI standard (Bland et al., 2013), provided by Open MPI. ULFM is an extension of the MPI standard that empowers MPI users—applications, library developers, or parallel programming languages—to provide their own fault-tolerant strategy. ULFM defines a set of additional API to MPI that permits (i) the interruption of MPI operations that cannot complete due to the occurrence of failures through raising appropriate MPI error classes; (ii) the continuation of point-to-point MPI messaging between nonfailed processes after such error classes have been raised; (iii) the interruption of MPI operations at all ranks in a particular communication handle (e.g. MPI COMM REVOKE, MPI WIN

REVOKE, MPI FILE REVOKE), under the explicit control of the programmer; (iv) the fault-tolerant validation of algorithmic steps (MPI COMM AGREE); and (v) the recovery of full operational capabilities (including the ability to perform collective communications) by constructing replacements for damaged communication objects (with MPI COMM SHRINK and MPI COMM SPAWN to recreate isomorphic communicators and then derive windows and files as necessary). The general design of ULFM relies on local semantics: The user is notified of failure only in MPI calls that involve a failed process, and a correct ULFM implementation will try to make all operations succeed if it can complete locally. Although this relaxed design eases the implementation requirements and delivers higher failure-free performance, the fact that a failure is guaranteed to be detected only after an active reception from the dead process can lead to an increase of latency during failure recovery operations, because the same process failures may be detected sequentially by multiple processes, possibly at a much later time than when they were first reported. Moreover, several routines imply necessarily a communicator-wide knowledge of failures. Operations like MPI_COMM_AGREE and MPI COMM SHRINK need to build consistent knowledge on (sub)sets of acknowledged failures; a pending point-to-point reception from any source must eventually raise an error if it cannot complete because of the death of a processor. Therefore, the addition of the FD and propagation service provides an acceleration to such scenarios by eliminating delayed local observation of the failure, which can then be immediately reported to the upper level, which can in turn act upon it quickly.

5.1. Implementation

The failure detector has two components: the observation ring and the propagation overlay. The components operate on a group of processes that must be MPI consistent (i.e. identical at all ranks). The propagation topology is implemented at the Byte Transport Layer (BTL) level, which provides the portable low-level transport abstraction in OPEN MPI.

The propagation overlay takes advantage of the Active Message behavior of the OPEN MPI BTL's. Each message, with a size less than the ''eager'' protocol switch point, contains the index of the callback function to be analyzed by upon reception. This approach provides independence from the MPI semantic (including matching). Upon the reception of a propagation message, the message is forwarded according to two possible algorithms. In the case where the overlay is not corrected to incorporate the knowledge about failed processes and thus the group can be considered as an invariant during the entire execution, the message is forwarded as is through the propagation topology which is constructed every time a broadcast is initiated, according to the algorithm presented in Section 2, in order to guarantee the logarithmic propagation delay. When the upper level declares—through a runtime parameter—that it repairs its communicators after every stabilization phase, the reliable propagation overlay can reduce the size of the messages to include only the latest detected failures, and the overlay is then built considering all processes of the group.

The observation ring is also built at the BTL level. The emission of the heartbeats poses a particular challenge in practice. The timely activation and delivery of heartbeats is critically important in enforcing the perfection of the detector and the bound on t. Missing its h emission period deadlines puts the emitter process at risk of becoming suspected by its observer, even though it is still alive. If the heartbeats are emitted from the application context, they can only be sent when the application enters MPI routines, and consequently, a compute intensive MPI application would often miss the h period. In our implementation, the heartbeats are emitted from within a separate, library internal thread, to render their emission independent from the application's communication pattern. For ease of implementation, the MPI_THREAD_MULTIPLE support is enabled by default when the detector thread is enabled; however, future software releases will drop this requirement. An intricate issue also arises from a negative interaction between the emission and the reception of heartbeat messages. To check the liveliness of the emitter process (after the d time-out), the observer has to see if it has received heartbeats. From an implementation perspective, if the heartbeats are sent through the eager channel, the detector thread (in this case, the receive thread) has to be active and poll the BTL engine for progress. However, if the application has posted operations on large messages, the poll operation may start progressing these (long) operations before returning control to the detector thread, leading to an unsafe delay in the emission of heartbeats from that same thread. To circumvent that difficulty, the detector thread emits heartbeats using the ''RDMA put'' channel. Heartbeats are thus directly deposited by raising a flag in the registered memory at the receiver, using hardware accelerated put operations that do not require active polling. The observer can then simply check that the flag has been raised during the last d period with a local load operation, and reset the flag with a local store, which are mostly impervious to noise and do not delay the h period. This approach also allows the observer to miss d periods without endangering the correctness of the protocol (only increasing the time to detect and notify the failure, but no triggering a false positive).

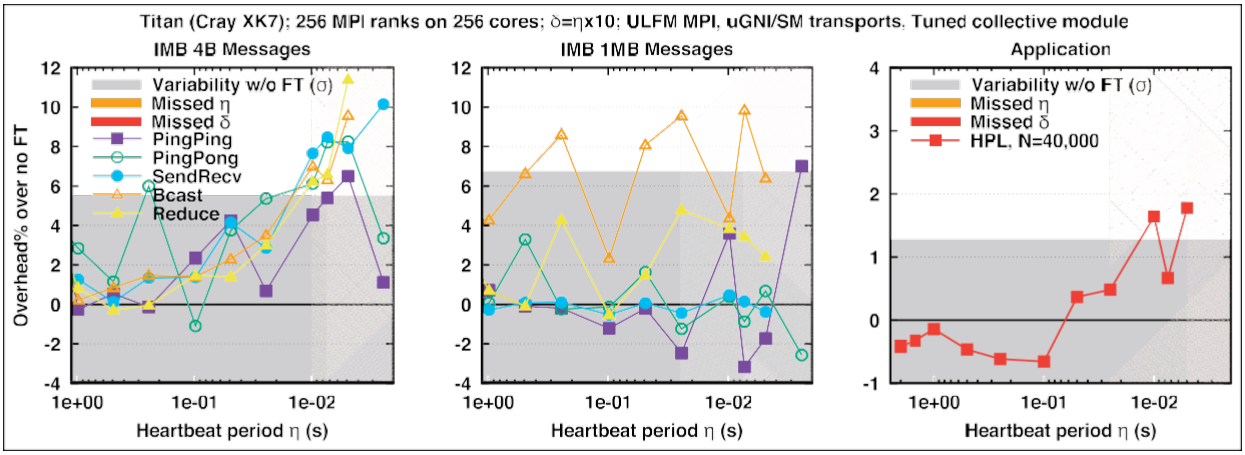


Figure 8. Sensitivity to noise resulting from the failure detector activity for varied workloads.

5.2. Experimental conditions

The experiments are carried out on the Titan ORNL Supercomputer (Titan, 2016), a Cray XK7 machine with 16-core AMD Opteron processors, and the Cray Gemini interconnect. The ULFM MPI implementation is based on a prerelease of Open MPI 2.x (r#6e6bbfd), which supports the optimized uGNI and shared-memory transports (without XPmem) and uses the Tuned collective module. The MPI implementation is compiled with the MPI_THREAD_MULTIPLE support. Every experiment is repeated 30 times and we present the average. The benchmarks are deployed with one MPI rank per core, and all threads of an MPI process are bound to that same core (application, detector, and driver threads when applicable, i.e. the detector thread does not require exclusive compute resources).

5.3. Noise and accuracy

The first set of experiments investigate the noise generated by the detector and its accuracy for different workloads when h and d vary, in a method similar to (Kharbas et al., 2012) that focused exclusively on measuring the noise generated by different FD strategies. The h and d periods are set so that d = 10 3 h. If the test is successful (i.e. no failure was detected, since none was injected in this experiment), then h is reduced, and the experiment is repeated, until a false positive is reported. We also collect the number of times an h deadline was missed, even when the d time-out is still respected. We first considered a noncommunicative, compute-only MPI application where each rank calls LAPACK DGEMM operations on local matrices, without calling MPI routines for extended periods of time. Without the detector thread, the noncommunicative benchmark reports false detections for all considered values of h. With the detector thread, this noncommunicative benchmark succeeds until h is set to 1 msec. However, starting from h\5 msec, messages indicating a missed h deadline are occasionally issued (although the d time-out is still respected). These observations are consistent with the scheduling time quantums (sched_min_granularity is set to 3 ms), and indicate that the thread scheduling latency is an absolute for the minimum h period. Smaller periods could be achieved with a real time scheduler, but such capabilities need administrative privileges, which is an undesirable requirement.

Next, in Figure 8, we present the noise incurred on a variety of communication and computation workloads, provided by the Intel MPI Benchmark (version 4.1) and HPL (version 2.2), respectively. Accuracy results are similar overall in the communicative benchmarks. All tests of the IMB-MPI1 suite can run without false detection for h210 ms. Notably, point-to-point only benchmarks can succeed with h value as low as 2.5 ms but occasionally report false suspicions. Collective communication benchmarks are more sensitive and report occasional heartbeat emission deadline misses until h 25 ms, due to contentions on the access to hardware network resources.

The latency performance (left graph) and bandwidth performance (center graph) are barely affected by low frequencies of heartbeat emissions. For higher frequencies, the overhead generated by the noise can reach approximately 10%. The bandwidth performance is less impacted overall than the latency, especially for point-to-point bandwidth, which remains unchanged for all but the most extreme values of h. The application performance (Linpack, right graph) exhibits no observable performance degradation for h2100 ms. For higher frequencies, the performance degradation remains contained under 2%.

5.4. Comparison with SWIM

This section compares our failure detector with SWIM (Das et al., 2002; Gupta et al., 2001; Snyder et al., 2014), the random protocol introduced in Section 3.4.1.

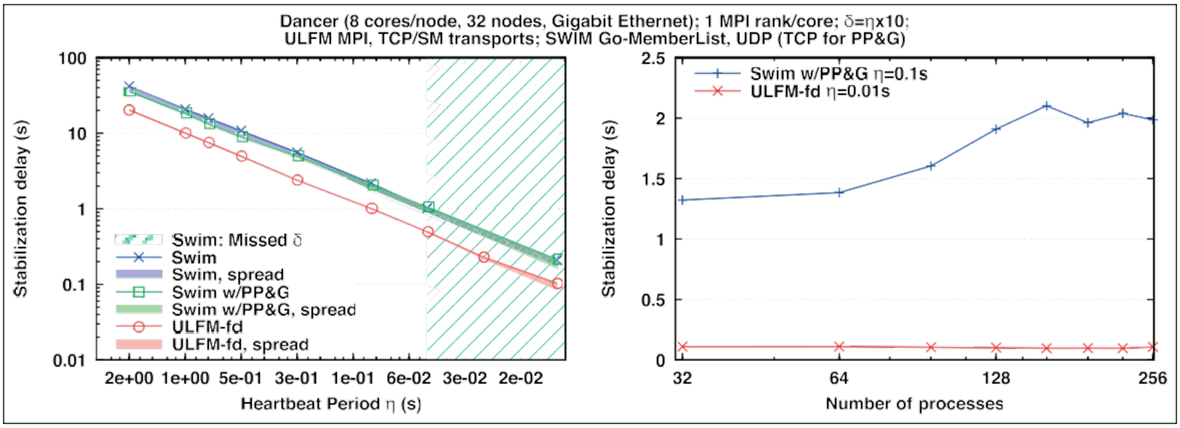


Figure 9. Detection and propagation delay compared to using the SWIM randomized failure detector from Memberlist.

SWIM scales by using a probabilistic approach: Nodes randomly choose a subset of neighbors to probe. To avoid false suspicions, SWIM relies on a collaborative approach. An initiator node invites *k* other nodes to form a group, pings them, and waits for their replies. If a node does not reply in time, the initiator then judges this node as suspicious and asks the other group members to check the potentially faulty node.

Figure 9 compares the detection delay (i.e. the stabilization delay) between the MPI failure detector and the SWIM failure detector, after a failure occurs at some process. For the MPI benchmark, after synchronizing, the desired number of MPI processes (whose ranks are chosen at random) simulate a failure. Any other process posts an *any-source* reception. When the reception raises a process failure exception (the only possible outcome for this nonmatched any-source reception), the process counts the number of locally known failed processes, and if it does not contain all injected failures, it repeats the reception. The SWIM benchmark also employs MPI to synchronize before injecting failures; however, the SWIM algorithm implementation—we used Go-Memberlist (r#d16b8b73)—is not integrated with MPI, and consequently the SWIM benchmark reports FDs directly through Go-Memberlist callbacks. In both cases (MPI and SWIM), the time at which all failures have been locally observed is reported at each rank. On the Titan platform, the Memberlist initialization over the ipogif interface (i.e. the Internaet protocol (IP) emulation layer over uGNI) suffers from a connection storm (a large group of simultaneous requests) and consequently often fails to initialize with more than 32 processes. A similar outcome has been observed on a different Linux cluster (called Dancer, a 32 nodes, 8 cores per node Xeon 7550, Ethernet Gigabit platform), but on that machine, the issue can be remediated by disabling the IP connection tracking kernel module (which supports iptables rules). With the contrack_nf module disabled, the message absorption rate is sufficient for the Memberlist benchmark to initialize and run to completion up to the maximum 256 processes that can be tested without oversubscription on that platform. Note that disabling the connection tracking module requires administrative privileges and severely limits the security of the system. Figure 7, therefore, presents results on the dancer platform, using Transmission Control Protocol (TCP) as the transport layer for both Memberlist and MPI.

The Memberlist implementation presents two variants of the SWIM protocol. The first one is the pure SWIM protocol, which relies exclusively on UDP heartbeats for both detecting and propagating the known suspected processes. Heartbeats are requested from random processes at the beginning of every period. The answer contains the list of currently suspected processes. If no answer is received before the time-out, the observed process itself becomes suspect. The second one expands on the SWIM protocol with the addition of requesting TCP handshakes with processes whose UDP heartbeats are not received in time and a periodic gossiping (with a random gossip algorithm) of the list of suspected processes. We refer this optimization as PP&G, for the Push-Pull and Gossip optimizations.

On the left graph in Figure 9, with 256 processes, the difference between pure SWIM and SWIM PP&G is minor. The PP&G optimization closes the spread between the first process suspecting a failure and the failure being reported at all processes (shaded area), especially for smaller values of h, resulting in marginally better stabilization delays. For values of h lower than 100 ms (which are, arguably, orders of magnitude more demanding than the default values selected for WAN SWIM deployments), false positive detections are reported for all variants of SWIM; the underlying reason lies in the loss of UDP messages due to occasional collisions; the failover TCP mechanism in the PP&G variant takes longer to establish the TCP connexion

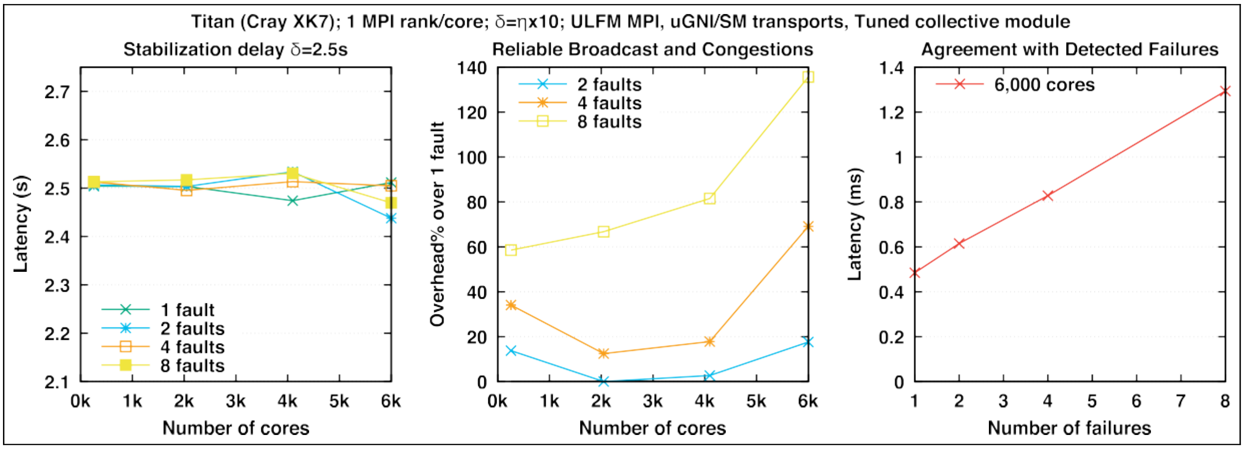


Figure 10. Detection and propagation delay and impact on completion time of fault-tolerant agreement operation.

than the detection time-out, which negates its advantages for such aggressive time-outs.

On the contrary, the ULFM failure detector is accurate for the entire range of h values (still subject to the kernel scheduler time quantum limitation discussed in the previous section). The spread between the first process detection and the stabilization delay is insignificant except for the smallest h considered, where it remains small nonetheless. Thanks to its deterministic behavior, the ULFM failure detector can remain accurate while reporting failures significantly faster than the SWIM algorithm employing the same heartbeat frequency. One has to consider that the number of messages exchanged for each heartbeat period is double in SWIM: After each heartbeat period, each process in the SWIM topology sends an observation request to a randomly selected process. This random selection process has the potential of creating hot spots, whenever many processes select to observe the same neighbor, which in turn increases the risk of message loss and consequently the risk of a false positive. Meanwhile, in our failure detector, a single message is sent, with a constant input and output degree of 1.

On the right graph of Figure 9, we compare the scalability of the detector with regard to the number of deployed processes. We selected the best performing PP&P variant for SWIM and employed the smallest safe value of h for each detector (which incidentally means that the h value for ULFM is smaller, thanks to its algorithm reporting fewer false positives). For a smaller number of processes, the ULFM failure detector is stabilizing in approximately 100 ms, while the SWIM algorithm stabilizes in 1.4 s. As the number of processes increases, the ULFM failure detector remains stable at 100 ms, while the stabilization delay of SWIM increases to over 2 s, an effect of the suspicion time-out, which is a logarithmic (in number of processes) delay added to the SWIM protocol to reduce the number of false positives.

5.5. FD time at scale

Figure 10 presents the behavior observed when injecting failures at scale. The first graph (left) presents the time to reach a stable state when injecting one to eight failures for a varying number of nodes. We observe that for small scales, the reported delay is consistently close to d. If emitters were sending heartbeats to their observer at random starting time, we would expect the detection time to be closer to d–h=2; however, as all processes start sending heartbeats to their observer at the end of the MPI_Init function, they are almost synchronized, and for all runs, we observe a consistent delay at small scale. At larger scale, processes leave MPI_Init at a more variable rate, and the average starts to converge toward the theoretical bound. This observation matches the model, considering that in this scenario, all failures are ‘‘simultaneous,’’ and that the random allocation of failures has a low probability of hurting observer/emitter pairs. Consequently, the detection and propagation of each of these failures progress concurrently and do not suffer from the cumulative effect of detecting multiple predecessors’ failures on the ring.

The second experiment (center in Figure 10) investigates the effect of collisions on the reliable broadcast propagation delay. The benchmark is similar to the previous experiment, except that before a process simulates a failure, it sends its observer a special ‘‘trigger heartbeat,’’ which initiates an immediate propagation reporting it dead, without waiting for the d time-out. The rest of the observation protocol remains unchanged (i.e. heartbeats are exchanged between alive processes with an h period, and the observer of the injection process switches to observing the predecessor). We then present the increase in the average duration of the reliable broadcast when multiple broadcasts are progressing concurrently. To simplify the proof of the upper bound on stabilization time (theorem 1), we have considered that successive broadcasts are totally sequential. This is an admittedly pessimistic hypothesis, and indeed, performing two concurrent propagations does not significantly increase the delay, as the two reliable broadcasts can actually overlap almost completely. However, starting from four, and, more prominently, for eight concurrent broadcasts, the average completion time is significantly increased. Considering the small size of the messages, the bandwidth requirements are small, and contention on port access is indeed the major cause of the imperfect overlap between these concurrent broadcasts, therefore vindicating the importance of considering a port-limited model during the design of the failure detector and propagation algorithms.

The last experiment (right in Figure 10) presents the performance of the agreement algorithm after failures have been injected. Herault et al. (2015) presented a similar performance result for their agreement algorithm. In their results, the agreement performance was severely impacted when failure was discovered during the agreement (with the failure-free performance of 80 ms increasing to approximatively 80 ms), an effect the authors claim is due to FD overhead. In their work, FD was delegated to an ORTE-based RAS[4] service, responsible for detecting and propagating failures. In this experiment, we strive to recreate as closely as possible this setup, except that we deploy our failure detector in lieu of the ORTE RAS service. We consider the same implementation of the agreement on 6000 Titan cores (the same number of cores they deployed on the generally similar Cray XC30 Darter system). Some in-band detection capabilities are active, in particular, failure of shared memory sibling ranks is reported by the node's local operating system. With the replacement of the ORTE RAS service by our failure detector algorithm, the time to completion of the agreement algorithm decreases to below 1.5 ms (a 50 3 improvement). This is due to the faster propagation of failure knowledge among the agreement participants: instead of waiting for (long) in-band time-outs or ORTE RAS notification, a process whose parent or children have failed can observe the condition much earlier, and start the online mending of the fan-in/fan-out tree topology at an earlier date. Interestingly, previously hidden performance issues become visible, as FD is not the dominant cost anymore: We observe that the performance of the agreement decreases linearly with the number of detected failures, a behavior that can be attributed to the agreement algorithm performing a linear scanning of the group when a failure is reported.

6. Related work

In this section, we survey related work on failure detectors and then on fault-tolerant broadcast algorithms.

6.1. Failure detectors

A number of FD algorithms have been proposed in the literature. Most current implementations of FDs are based on an all-to-all communication approach where each node periodically sends heartbeat messages to all nodes. Because they consider a fully connected set of known nodes that communicate in an all-to-all manner, these implementations are not appropriate for platforms equipped with a large number of nodes.

Several efforts have been made toward scaling up failure detectors implementations. Bertier et al. (2003) introduced a hierarchical organization suitable for grid configurations. They define a two-level organization to reduce message overhead. Local groups are cluster nodes, bound together by a global intercluster group. Every local group elects one leader that is member in the global group. Within each group, any member monitors all other members. While hierarchical approaches provide short local detection time, the cost of reconfiguration and the propagation of failure information both remain high. Larrea et al. (2000) also aim to diminish the amount of exchanged information in order to scale up. To do so, they use a logical ring to structure message exchanges. Thus, the number of messages to detect failures is minimal, but the time for propagating failure information is linear to the number of nodes.

An alternative approach for implementing scalable failure detectors is to use gossip-like protocols where nodes randomly choose a few other nodes with whom they exchange their failure information (Gupta et al., 2001; van Renesse et al., 1998). The idea is that, with high probability, eventually all nodes obtain every piece of information. The work of van Renesse et al. (1998) is one of the pioneering implementations of gossip-style failure detectors. In their basic protocol, each node maintains a list with a heartbeat counter for each known node. Periodically, every node increments its own counter and selects a random node which to send its list. A disadvantage is that the size of gossip messages grows with the size of the network, which induces a high-network traffic. The authors identified a variant specifically designed for large-scale distributed systems: the multilevel gossiping. They concentrate the traffic within subsets of nodes to improve the scalability. Hayashibara et al. (2002) explored a hybrid approach based on both dynamic clustering to solve the scalability issue and the gossiping technique to remove wrong suspicions. Horita et al. (2005) presented another scalable failure detector that creates scattered monitoring relations among nodes. Each node is intended to be monitored by a small number k of other nodes (with k set typically to 4 or 5). When a node dies, one of the monitoring nodes will detect the failure and propagate this information across the whole system. Similarly, as discussed in Section 5.4, SWIM (Das et al., 2002) scales by using a probabilistic approach. More recently, Tock et al. (2013) proposed a scalable membership service based on a hierarchical fast unreliable FD mechanism, where failure information can be lost, combined with a slower gossip protocol for eventual information dissemination. Finally, Katti et al. (2015) designed a scalable failure detector based on observing random nodes and gossiping information. In their protocol, each ping message transmits information on all currently known failures, either via a liveness matrix or in compressed form.

Practically, gossip approaches bring along redundant failure information which degrades their scalability. Furthermore, the randomization used by gossip protocols makes the definition of time-out values difficult, since the monitoring sets change often over time. In order to eventually avoid false detections, these techniques tend to oversize their time-outs, which results in longer detection times. Theoretically, gossip approaches introduce random detection and propagation times, whose worst case with a prescribed risk factor is hard to bound.[5] In contrast, our algorithm follows a deterministic detection and propagation topology with (i) constant-size heartbeats and well-defined delays, (ii) a single observer, (iii) a logarithmic-time propagation, and (iv) a guaranteed worst time to stabilization, thereby achieving all the goals of randomized methods with a deterministic implementation.

6.2. Fault-tolerant broadcast

Fault-tolerant broadcasting algorithms have been extensively studied, and we refer the reader to the surveys by Heydemann (1997) and Pelc (1996). A key concept is the fault-tolerant diameter of the interconnection graph, which is defined as the maximum length of the longest path in the graph when a given number of (arbitrarily chosen) nodes have failed (Krishnamoorthy and Krishnamurthy, 1987). The main objective in this context is to identify classes of overlay networks whose fault-tolerant diameter is close to their initial (fault-free) diameter, even when allowing a number of failures close to their minimal degree (allowing more failures than the minimal degree could disconnect the graph). Furthermore, these overlay networks should provide enough vertex-disjoint paths for broadcast algorithms to resist that many failures.

Research has concentrated on regular graphs (where all vertices have the same degree): hypercubes (Fraigniaud, 1992; Krishnamoorthy and Krishnamurthy, 1987; Ramanathan and Shin, 1988), binomial graphs (Angskun et al., 2007), or circulant networks (Liaw et al., 1998). For all these graphs, efficient broadcast algorithms have been proposed. These algorithms tolerate a number of failures up to their degree minus 1 and execute within a number of steps (in the one-port model) that does not exceed twice their original diameter. However, to the best of our knowledge, such algorithms require the number of nodes in the graph to be a power of 2, or a constant times a power of 2, while we need an algorithm for an arbitrary number of nodes. This motivates our solution based upon a double diffusion (see Section 2).

7. Conclusion

FD is a critical service for resilience. The failure detector presented in this work relies on heartbeats, time-outs, and communication bounds to provide a reliable solution that works at scale, independently of the type of faults that create permanent node failures. Our study reveals a complicated trade-off between system noise, detection time, and risks: A low-detection time would demand a low latency in the detection of failures, thus a tight approximation of the communication bound, increasing the risk of a false positive, and a frequent emission of heartbeat messages, increasing the system noise generated by the failure detector. We proposed a scalable algorithm capable of tolerating high-frequency failures and proved a theoretical upper bound to the time required to reconfigure the system in a state that allows new failures to strike; therefore, the algorithm can tolerate an arbitrary number of failures, provided that they do not strike with higher frequency. The algorithm was implemented in a resilient MPI distribution, which we used to assess its performance and impact on applications at large scale. The performance evaluation shows that for reasonable values of detection time, the ring strategy for detection introduces a negligible or nonmeasurable amount of additional noise in the system, while the high-performance reliable broadcast strategy for notification allows for quickly disseminating the fault information, once detected by the observing process.

Implementation considerations lead us to advocate that the detection part of the service should be provided at a lower levels of the software stack, either inside the operating system or inside the interconnect hardware. Active heartbeats to probe the activity of remote nodes could be handled by these lower levels without measurable noise, and with tighter bounds, since the other levels of the software stack would not introduce additional components to the noise. Future work should focus on providing this capability and on evaluating the approach to address the trade-off between detection time and risk.

Acknowledgements

The authors would like to thank the reviewers for their insightful comments. Yves Robert is with the Institut Universitaire de France.

Authors' note

A shorter version of this work has been published in the proceedings of SC'16 [1]. Preprint submitted to IJHPCA April 15, 2017.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research is partially supported by the CREST project of the Japan Science and Technology Agency (JST), by NSF grant #1339820, by the PIA ELCI (Bull Inria) project and partially supported by the NSF (award \#1564133).

Notes

1. We use the words *failure* and *death* indifferently.
2. Delay-bounded fault-tolerant broadcasts are not easily obtained for arbitrary values of n (see the discussion in Section 6.3).
3. http://icl.utk.edu/herault/ijhpca-failure-detector.tgz
4. ORTE stands for Open Run-Time Environment and RAS for Resource Allocation Subsystem.
5. Absolute worst-case times are infinite, as some nodes could be observed only after an unbounded delay (see the discussion of Section 3.4).

References

Angskun T, Bosilca G and Dongarra J (2007) Binomial graph: a scalable and fault-tolerant logical network topology. In: Stojmenovic I, Thulasiram RK, Yang LT, Jia W, Guo M and de Mello RF (eds) *Parallel and Distributed Processing and Applications ISPA*, Berlin: Springer, pp. 471–482.

Bertier M, Marin O and Sens P (2003) Performance analysis of a hierarchical failure detector. In: *Proceedings of the International Conference on Dependable Systems and Networks*, 22–25 June 2003 San Francisco, CA, 2003, pp. 635–644.

Bhat PB, Raghavendra CS and Prasanna VK (2003) Efficient collective communication in distributed heterogeneous systems. *Journal Parallel and Distributed Computing* 63(3): 251–263.

Bland W, Bouteiller A, Herault T, et al. (2013a) An evaluation of user-level failure mitigation support in MPI. *Computing* 95(12): 1171–1184.

Bland W, Bouteiller A, Herault T, et al. (2013b) Post-failure recovery of MPI communication capability: design and rationale. *International Journal of High Performance Computing Applications* 27(3): 244–254. arXiv: http://hpc.sagepub.com/content/27/3/244.full.pdf + html, doi:10.1177/1094342013488238. URL http://hpc.sagepub.com/content/27/3/244.abstract

Bland W, Lu H, Seo S, et al. (2015) Lessons Learned Implementing User-Level Failure Mitigation in MPICH. In: *Proceeding CCGrid*, 2015. 4–7 May 2015 IEEE. Shenzhen, China, DOI: 10.1109/CCGrid.2015.51

Chandra TD and Toueg S (1996) Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43(2): 225–267.

Chen W, Toueg S and Aguilera MK (2002) On the quality of service of failure detectors. *IEEE Transactions Computers* 51(5): 561–580.

Das A, Gupta I and Motivala A (2002) Swim: Scalable weakly-consistent infection-style process group membership protocol. In: *International Conference on Dependable Systems and Networks*, 23–26 June 2002 Washington, DC, USA, 2002, pp. 303–312.

Egwutuoha IP, Levy D, Selic B, et al. (2013) A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* 65(3): 1302–1326.

Ferreira KB, Bridges P and Brightwell R (2008) Characterizing application sensitivity to OS interference using kernel-level noise injection. In: *Proceeding SC'08, IEEE Computer Society Press*, 2008. 15–21 Nov. 2008, Austin, TX, USA: IEEE.

Fraigniaud P (1992) Asymptotically optimal broadcasting and gossiping in faulty hypercube multicomputers. *IEEE Transactions Computers* 41(11): 1410–1419.

Gupta I, Chandra TD and Goldszmidt GS (2001) On scalable and efficient distributed failure detectors. In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC'01*, New York, NY, USA, 2001, pp. 170–179. ACM. DOI: 10.1145/383962.384010.

Hayashibara N, Cherif A and Katayama T (2002) Failure detectors for large-scale distributed systems. In: *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, Osaka, Japan, 13–16 October 2002, pp. 404–409. DOI:10.1109/RELDIS.2002.1180218.

Herault T, Bouteiller A, Bosilca G, et al. (2015) Practical scalable consensus for pseudo-synchronous distributed systems. In: *Proceeding*. SC'15, *IEEE Computer Society Press*, 2015. Austin, Texas — November 15–20, 2015.

He´rault T and Robert Y (2015) Fault-tolerance techniques for high-performance computing. In: He´rault T and Robert Y (eds) *Computer Communications and Networks*. Verlag: Springer, 2015.

Heydemann MC (1997) Cayley graphs and interconnection networks. In: Hahn G and Sabidussi G (eds) *Graph Symmetry: Algebraic Methods and Applications*, Springer, 1997, pp. 167–224.

Hoefler T, Schneider T and Lumsdaine A (2010) Characterizing the influence of system noise on large-scale applications by simulation. In: *Proceeding. SC'10, IEEE Computer Society Press*, 2010.

Horita Y, Taura K and Chikayama T (2005) A scalable and efficient self-organizing failure detector for grid applications. In: *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, GRID'05*, Washington, DC, USA, 2005, pp. 202–210. IEEE Computer Society.

Katti A, Di Fatta G, Naughton T, et al. (2015) Scalable and fault tolerant failure detection and consensus. In: *Proceeding. EuroMPI'15*, 2015. ACM.

Kharbas K, Kim D, Hoefler T, et al. (2012) Assessing HPC failure detectors for MPI jobs. In: *Proceeding. PDP'12, IEEE Computer Society*, 2012.

Krishnamoorthy M and Krishnamurthy B (1987) Fault diameter of interconnection networks. *Computers & Mathematics with Applications* 13(5–6): 577–582.

Larrea M, Fema´ndez A and Are´valo S (2000) Optimal implementation of the weakest failure detector for solving consensus. In: *Proceedings, 2000, 19th IEEE Symposium on Reliable Distributed Systems, SRDS'00*, Nu¨rnberg, Germany, 16–18 October 2000, pp. 52–59.

Liaw SC, Chang GJ, Cao F, et al. (1998) Fault-tolerant routing in circulant networks and cycle prefix networks. *Annals of Combinatorics* 2(2): 165–172.

Mitzenmacher M and Upfal E (2005) *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge: Cambridge University Press.

Pelc A (1996) Fault-tolerant broadcasting and gossiping in communication networks. *Networks* 28(3): 143–156.

Ramanathan P and Shin KG (1988) Reliable broadcast in hypercube multicomputers. *IEEE Transactions Computers* 37(12): 1654–1657.

Snyder S, Carns PH, Jenkins J, et al. (2014) A case for epidemic fault detection and group membership in HPC storage systems. In: *5th Int. Workshop on Performance Modeling, Benchmarking, and Simulation (PMBS), LNCS 8966*, 2014, pp. 237–248. Springer.

Titan (2016) Oak Ridge National Laboratory. https://www.olcf.ornl.gov/titan/ (accessed 2016).

Tock Y, Mandler B, Moreira JE, et al. (2013) Design and implementation of a scalable membership service for supercomputer resiliency-aware runtime. In: *Proceedings, 2013, Processing – 19th International Conference Euro-Par 2013 Parallel*, Aachen, Germany, 26–30 August 2013, pp. 354–366. DOI:10.1007/978-3-642-40047-6_37.

van Renesse R, Minsky Y and Hayden M (1998) A gossip-style failure detection service. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware '98*, London, UK, 1998, pp. 55–70. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=1659232.1659238.

Wung DS (2009) *Intelligent platform management interface (IPMI)*. PhD Thesis, SLAC National Accelerator Laboratory.

Author biographies

George Bosilca is a research director and an adjunct assistant professor at the Innovative Computing Laboratory at University of Tennessee, Knoxville. His research interests evolve around distributed algorithms, parallel programming paradigms, and performance modeling and optimization, both from a theoretical and practical perspective. He is also interested in providing scalable and portable constructs for building resilience directly into the programming models.

Aurelien Bouteiller is a researcher at the University of Tennessee's Innovative Computing Laboratory. His research is focused on improving performance and reliability of distributed memory systems, mechanisms to improve communication speed and balance of many core clusters, one-sided communication on threaded systems, recoverable communication libraries to support fault tolerance, and emerging dataflow programming models.

Amina Guermouche received her PhD degree from University of Paris-Sud. She is currently an assistant professor at Telecom Paris-Sud. Her research interests evolve around fault-tolerance algorithms and energy minimization for large-scale platforms.

Thomas Herault is a research scientist at the Innovative Computing Laboratory at University of Tennessee, Knoxville. His research interests include fault tolerance, distributed algorithms, parallel programming paradigms, and performance modeling and optimizations. He focuses on bridging the gap between theoretical distributed systems and high-performance computing as it is practiced.

Yves Robert received the PhD degree from Institut National Polytechnique de Grenoble. He is currently a full-time professor in the Computer Science Laboratory LIP at ENS Lyon. He is the author of 7 books, 150 papers published in international journals, and 230 papers published in international conferences. He is the editor of 11 book proceedings and 13 journal special issues. He is the advisor of 30 PhD theses. His main research interests are scheduling techniques and resilient algorithms for large-scale platforms. He served on many editorial boards and currently is an editor of IEEE TPDS, JPDC, JoCS, and IJHPCA. He is a fellow of the IEEE. He has been elected a senior member of Institut Universitaire de France in 2007 and renewed in 2012. He has been awarded the 2014 IEEE TCSC Award for Excellence in Scalable Computing and the 2016 IEEE TCPP Outstanding Service Award. He holds a visiting scientist position at the University of Tennessee Knoxville since 2011.

Pierre Sens received his PhD in computer science in 1994 and the *Habilitation à diriger des recherches* in 2000 from Paris 6 University (UPMC), France. Currently, he is a full-time professor at UPMC. His research interests include distributed systems and algorithms, large-scale data storage, fault tolerance, and cloud computing. Since 2005, he is heading the Regal group which is a joint research team between LIP6 and Inria. He was member of the Program Committee of major conferences in the areas of distributed systems and parallelism (ICDCS, IPDPS, OPODIS, ICPP, Europar, etc.) and serves as general chair of SBAC and EDCC. Overall, he has published over 130 papers in international journals and conferences and has acted for advisor of 19 PhD theses.

Jack Dongarra received a bachelor of science in mathematics from Chicago State University in 1972 and a master of science in computer science from the Illinois Institute of Technology in 1973. He received his PhD in applied mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Electrical Engineering and Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL); turing fellow at Manchester University; an adjunct professor in the Computer Science Department at Rice University; and a faculty fellow of the Texas A&M University's Institute for Advanced Study. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research at the University of Tennessee which coordinates and facilitates IT research efforts at the University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing, and documentation of high-quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports, and technical memoranda and, he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches; in 2008, he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010, he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011, he was the recipient of the IEEE Charles Babbage Award; and in 2013, he was the recipient of the ACM/IEEE Ken Kennedy Award for his leadership in designing and promoting standards for mathematical software used to solve numerical problems common to high-performance computing. He is a fellow of the AAAS, ACM, IEEE, and SIAM, as well as a foreign member of the Russian Academy of Sciences and a member of the US National Academy of Engineering.