A Generic and High Performance Approach for Fault Tolerance in Communication Library

François Trahay[1] Alexandre Denis[2]
Yutaka Ishikawa[1]
[1] RIKEN, University of Tokyo,
{trahay,ishikawa}@il.is.s.u-tokyo.ac.jp
[2] INRIA, LABRI
alexandre.denis@inria.fr

Abstract

With the increase of the number of nodes in clusters, the probability of failures increases. In this paper, we study the failures in the network stack for high performance networks. We present the design of several fault-tolerance mechanisms for communication libraries to detect failures and to ensure message integrity. We have implemented these mechanisms in the NEWMADELEINE *communication library with a quick detection of failures in a portable way, and with fallback to available links when an error occurs. Our mechanisms ensure the integrity of messages without lowering too much the networking performance. Our evaluation show that ensuring fault-tolerance does not impact significantly the performance of most applications.*

1 Introduction

Since the development of large scale supercomputers have led to systems composed of hundreds of thousands of components, the likelihood of hardware or software failure becomes embarrassing. The design of future supercomputers foreshadows an increasing number of components, decreasing the mean time between failures [4]. Among the common causes of failures, the crash of computing nodes can be bypassed by using checkpoint and restart mechanisms. However, failures may also happen on the network – a switch can fail or a NIC may crash – and communication libraries thus have to support such failures and maintain the connectivity between nodes as much as possible.

In this paper, we present a generic retransmission mechanism that permits to reroute automatically the communication flow through surviving links when a network failure occurs. The originality of our work consists in two mechanisms: quick network failure detection, and low-overhead checksums to ensure data integrity.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 describes the design of the retransmission mechanism. The fault-detection mechanism is presented in Section 4 and the technique used for ensuring message integrity is described in Section 5. Results are discussed in Section 6 and we draw a conclusion in Section 7.

2 Related work

The advent of large scale supercomputers as well as grid computing have led to numerous researches on fault-tolerance. Various works have focused on surviving to node failures by using checkpointing mechanisms. However, the whole software stack has to be fault-tolerant and several researches have been conducted to make MPI implementations capable of supporting dynamic disconnection and reconnection. MPICH-V [3] uses a virtual channel mechanism that abstracts connections between nodes and permits to respawn a crashed process without losing messages. FT-MPI [5] supports process failures by allowing dynamic resizing of MPI communicators. These mechanisms permit to survive a process or network failure at the expense of high overhead. The causes of the invocation of these mechanisms – network or process failures – should thus be avoided.

Various researches were conducted on providing fault-tolerant communication libraries able to survive network failures. Some studies have focused on detecting network errors by using pinging or heart beat techniques [2]. However, this may lead to unnecessary messages on the network and it can be problematic for large number of processes. Most fault-tolerant communication libraries for high performance networks thus use a sender-based logging mechanism [15]: when a message is received, an acknowledgment is sent back. If the sender process does not receive the acknowledgment before a timeout, the network is assumed to have failed. Generic MPI implementations usually use a multi-second timeout, but since some network interfaces provide hardware timeout, specific MPI implementations can use them for detecting failures in a few milliseconds [7]. Detecting network failure rapidly while preserving the portability is thus difficult with current MPI implementations.

Since altered messages are likely to false the results of applications, various researches have focused on ensuring the integrity of MPI messages. LA-MPI [6] or OPEN MPI [12] compute checksums on transferred messages and are able to retransmit corrupted fragments. This ensures the integrity of messages, but the overhead introduced by the computation of checksums may significantly impact the performance of applications.

When a network failure is detected, the communication library needs to recover from this error. On IP networks, it is possible to use an alternative route that may be safe [10]. For Infiniband networks, MVAPICH2 [8] tries to reconnect or to reload the HCA driver, before rerouting messages to another Infiniband HCA. RI2N [11] can exploit several Ethernet links and fallbacks to surviving links when a network becomes unavailable. MPI implementations that support heterogeneous multirail can reroute messages to another network. Most supercomputer being equipped with both high speed network and Ethernet, this latter technique permits to maintain connections between nodes in case a network fails.

3 Surviving network failures

In order to avoid using heavy recovery mechanisms at the application level, it is necessary for the communication library to survive network failures. An issue occurring on the network – a switch crashing for example – or on a remote node – a process crashing or a remote NIC failure – should not cause the local process to crash. The communication library thus has to handle network errors and ensure that no message is lost.

In this Section, we present the recovery mechanism implemented in the NEWMADELEINE communication library [1] that permits to ensure these properties. Since NEWMADELEINE can exploit several networks simultaneously, a failure happening on a network can be overridden by rerouting the traffic to other links. Since most supercomputers are connected through one or more high speed networks as well as through Ethernet, this permits to survive at least one network failure: the application benefits from the fast network and can rvert to the slow one in case of failure.

Recovering from a network failure requires to stop using the faulty link and to retransmit all the messages that may have been lost. Since NEWMADELEINE maintains a list of available links for each connected process, this can be done by changing the state of the faulty link. When communicating with the remote process for which the error was

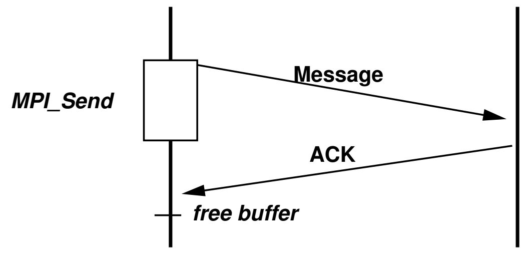

Figure 1. Reliability protocol for eager messages

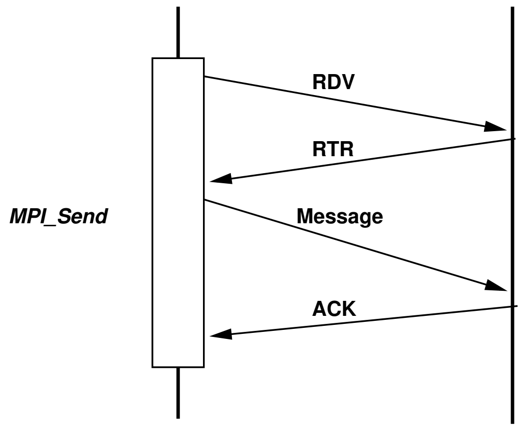

Figure 2. Reliability protocol for *rendezvous* **messages**

detected, the faulty driver will not be use. However, the driver remains available for other processes. The retransmission of messages requires to know which one were actually received, thus NEWMADELEINE uses a sender-based message logging: messages are kept in memory until the receiving process notifies their reception. As it is depicted in Figure 1, for messages transmitted using a *eager* protocol NEWMADELEINE copies the data into a pre-registered buffer, the data is thus freed when the corresponding acknowledgment message is received. Figure 2 illustrates the case of large messages that use the *rendezvous* protocol: since NEWMADELEINE does not copy the data, the application is notified that the message was sent only when the acknowledgment is received. It is to be noted that, since NEWMADELEINE can aggregate messages, acknowledgments can be merged into data messages, reducing the overhead to a message aggregation.

When a network failure is detected, NEWMADELEINE retransmits all pending messages through one of the remaining links. The receiver detects duplicate messages using their sequence number.

This generic retransmission mechanism thus permits to survive to network errors as long as a route to the remote process exists. If a remote NIC crashes, messages are rerouted using another NIC. If a network fails – cable or switch failure – messages are rerouted and other errors may be detected from links that use the faulty hardware. If a remote process crashes, messages are rerouted using available NICs until all the links to this node are disabled. The process is then considered as unreachable and the application should be notified.

4 Detecting network failures

The retransmission mechanism described in Section 3 being based on message logging, the memory consumption depends on how fast acknowledgment messages are sent back. Ensuring fast reply to messages and detecting network failures quickly thus permits to reduce the impact of fault tolerance on memory consumption.

The low-level drivers used in NEWMADELEINE can provide useful information on network failures. Most network interfaces can report errors that happen during message transmission. However the reactivity to these network errors can be slow – for instance, the MX interface only supports multi-second timeouts – and such mechanism cannot detect every errors. Reacting to the drivers notifications is thus not sufficient.

Since the retransmission mechanism uses message logging, each outgoing message induces an

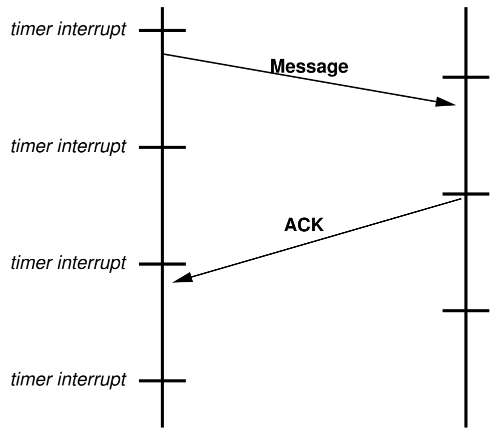

Figure 3. Detection of network failures in NEWMADELEINE

acknowledgment. If NEWMADELEINE detects that an acknowledgment was not received before a timeout expires, it assumes that a network failure happened. Choosing the timeout value can be tricky since a small value may lead to many false positives while waiting for too long may cause high memory consumption. Moreover, if the receiver process entered a computing phase, it may not let the communication library poll the network, delaying the acknowledgment for several seconds and leading to an incorrect network failure detection.

Using the PIOMan event manager, we showed [14] that NEWMADELEINE is guaranteed to detect network events within one scheduler timeslice. Even during heavy computing phases, NEWMADELEINE is scheduled through a timer interrupt, typically every 10 ms. As depicted in Figure 3, a message sent to a process is detected during the next timer interrupt in the worst case, and immediately acknowledged. Thus, when sending a message through the network, NEWMADELEINE can guarantee that acknowledgments arrive within 3 timeslices, unless a network failure occurs. Pending messages are periodically checked for expired timeouts. In that case, we assume that a network failure occurred and use the retransmission mechanism described in Section 3.

5 Ensuring message integrity with low overhead

On their way from the sender memory through the receiver memory, messages may be corrupted with some bits flipped. It may occur on the wire, in the NIC, or on the PCIe bus. Most network hardware use checksums internally to ensure message integrity on the wire, but corruption may occur at any other given point.

5.1 Adding checksums on the data path

To ensure end-to-end message integrity, in NEWMADELEINE we use a classical approach based on *checksums*. The sender computes a checksum of the message to be sent, and sends this checksum with the message headers. The receiver then computes the checksum on the received messages. If the computed checksum doesn't match the one received alongside the data, it means corruption occurred: either the data, the headers, or the checksum itself have been corrupted during the transfer. In this case, we ask the generic layer to retransmit the packet, as described in Section 3.

However, computing checksums has a cost that may lower the available bandwidth. Our approach consists in amortizing the cost of checksum computation by combining the checksum and the memory copy wherever it happens.

Figure 4 shows the bandwidth of some checksums and hashing functions (that may be used as checksums) on our `jack` cluster, equipped with dual-core Xeon X5650 at 2.67 GHz, on 32 kB blocks that fit the L1 cache. On this machine, the memory bandwidth for reading is 9700 MB/s and the copy bandwidth is 4530 MB/s. Thus, the simplest checksum algorithms are memory-bound. We propose to compute the checksum on

algorithm	bandwidth
plain sum	24102.98 MB/s
xor	15964.92 MB/s
Adler-32	1764.71 MB/s
Fletcher-64	6088.44 MB/s
Jenkins one-at-a-time	3489.67 MB/s
Fowler/Noll/Vo hashing	6102.05 MB/s
Knuth hashing	4463.70 MB/s
MurmurHash2a	6091.84 MB/s
Paul Hsieh SuperFast	1696.15 MB/s
SSE4.2 CRC32	8129.00 MB/s

Figure 4. Bandwidth of some checksum algorithms on 32 kB blocks.

the fly at every place where data is copied in NEWMADELEINE; once data has been fetched in cache during the copy, we can expect the checksum to get the same bandwidth as results of Figure 4.

The plain sum is the fastest, and is likely to always be on any hardware. However, it cannot reliably detect corruption beyond a single bit. If the user wants a better checksum than plain sum to detect corruption larger than a single bit, we use the SSE 4.2 CRC32 checksum if available on the given hardware, or the Fletcher-64 checksum else [9].

5.2 Checksums for eager send

In NEWMADELEINE, small packets are sent with an eager protocol. Data is copied to add the headers and to apply optimization strategies such as aggregation of multiple messages into one packet. We add the checksum computation immediately following the copy, when data is still in cache. On the receiver side, NEWMADELEINE receives packets in its internals buffers, then parses headers, performs matching, and unpacks data to its final destination in the user buffers. We add the checksum computation for the received data immediately after this copy, while data is still in cache.

Let λ_{net} and B_{net} be the latency and bandwidth of the network; B_{copy} the bandwidth of memory copy, and B_{csum} the bandwidth of checksum computation for data already in cache, then the total transfer time for a message of length L sent with eager mode is:

$$T(L) = \frac{2 \times L}{B_{copy}} + \frac{2 \times L}{B_{csum}} + \lambda_{net} + \frac{L}{B_{net}}$$

On the `jack` cluster, equipped with ConnectX2 Infiniband QDR HCA, we have $\lambda_{net} = 1.4\,\mu s$; $B_{net} = 3\,GB/s$; $B_{copy} = 4.5\,GB/s$; $B_{csum} = 24.1 GB/s$. Then we can compute the expected overhead of checksums to be 7 % on 4 kB messages and 10 % on 32 kB messages. We get the same order of magnitude for the overhead of checksum – from 5 % to 10 %– on other clusters.

5.3 Checksum with rendezvous

Large messages are sent through a *rendezvous* protocol in NEWMADELEINE. In this case, our strategy to compute checksums depends on the underlying driver.

For Infiniband, NEWMADELEINE implements two different strategies for memory registration: dynamic registration of data, with a registration cache (*rcache*); or memory copy through a pre-registered memory buffer, with a variable depth super-pipeline to overlap copy and send (*pipeline*). Since this later method uses a memory copy, it is very easy to add the checksum computation on the fly on both sender and receiver sides. We actually interleave memory copy and checksum computation with sub-blocking at the size of L1 cache, to reduce the overhead of checksum. To ensure overlap of copy and send operations, this pipeline relies on the copy bandwidth to be higher than the network bandwidth. When we add checksums, the bandwidth of copy+checksum is still higher than network, so the impact of checksums on pipeline performance is expected to be low.

On other networks, NEWMADELEINE performs no memory copy. In this case, our approach to amortize the cost of checksums consists in computing checksums in PIOMan asynchronous ltasks [13] as soon as the users posts the send request, without waiting for the receiver reply. There is thus an overlap of the *rendezvous* roundtrip and the checksum computation. Moreover, we compute the checksum in parallel in multiple ltasks so as to fully exploit the available CPU cores.

6 Evaluation

In this Section, we present the results obtained by comparing the original NEWMADELEINE with the fault-tolerant NEWMADELEINE. We evaluate the raw overhead of fault-tolerance mechanisms as well as their impact on NAS Parallel Benchmarks. We also present results obtained with the current development version of OPEN MPI. Its `csum` component performs data integrity checks but cannot correct corruption errors, and the `dr` component also detects network failures and supports message retransmission.

The results that we present were obtained on the PLAFRIM and CLUSTER0 platforms. Each node of the PLAFRIM cluster is composed of two quad-core Xeon CPUs (X5550) running at 2.66 GHz. Each node features 24 GB of RAM and one Infiniband QDR HCA. The CLUSTER0 testbed is composed of nodes equipped with two dual-core Opteron CPUs (2214 HE), 4 GB of RAM and one Myrinet Myri-10G NIC.

6.1 Raw overhead

We now present the results obtained with a ping pong program on Infiniband and Myrinet. The latency obtained on Infiniband is reported in Figure 5. OPEN MPI `csum` causes an overhead of 100 ns, while the fault-tolerant version degrades the latency by 1.3 μs. The message retransmission mechanism implemented in NEWMADELEINE causes an overhead of 500 ns. This lighter overhead is due to the ability to aggregate messages: the acknowledgment is aggregated with the application message so that only one packet

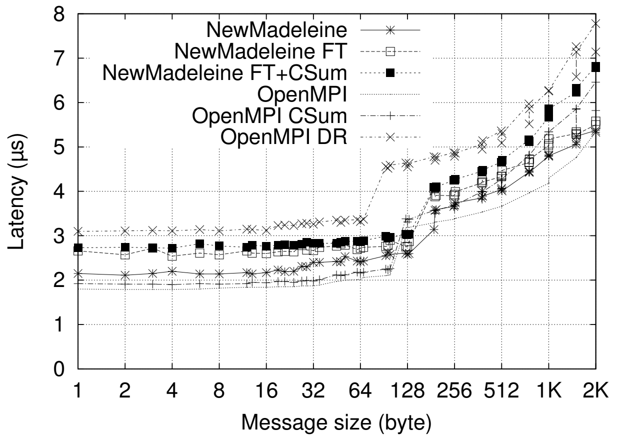

Figure 5. Latency over Infiniband

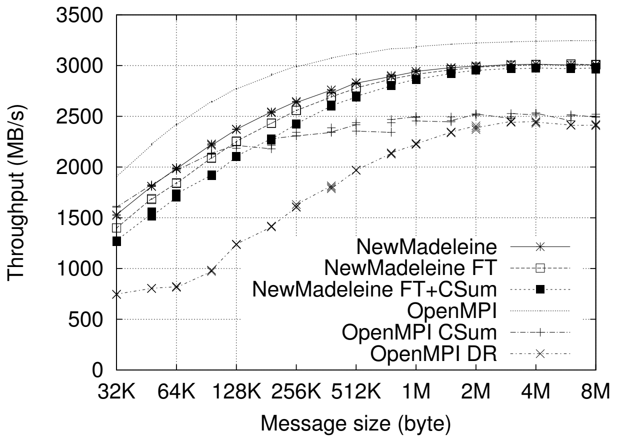

Figure 6. Throughput over Infiniband

is sent through the network. Adding message integrity checks in NEWMADELEINE causes an extra 100 ns overhead.

Figure 6 reports the throughput results obtained for Infiniband. OPEN MPI `csum` and `dr` mechanisms degrade the throughput by approximately 20 %. This is mainly due to the sequential computation of checksums. However the OPEN MPI `dr` suffers from an additional overhead for medium size messages due to the message retransmission mechanism. The results obtained with NEWMADELEINE show little overhead when the fault

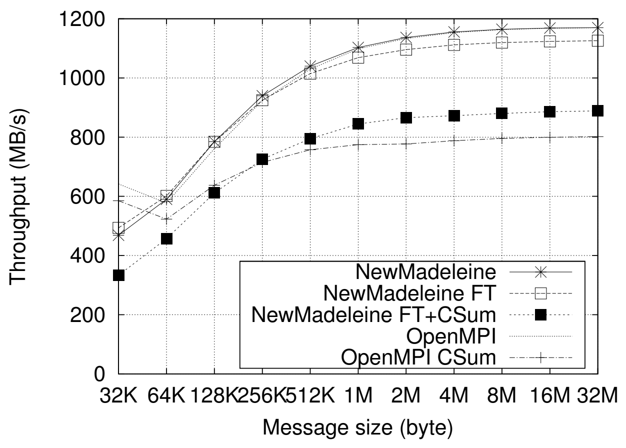

Figure 7. Throughput over MX

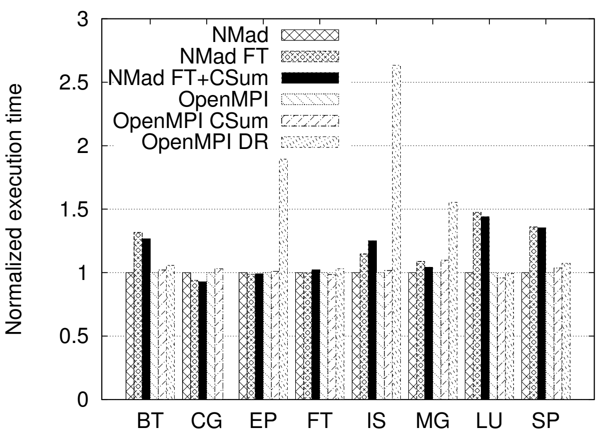

Figure 8. NAS Class B on 16 processes

tolerance mechanisms are enabled. This is due to the checksum being computed during the memory copy in the Infiniband driver.

The results obtained with Myrinet are reported in Figure 7. OPEN MPI `dr` mechanism causes an error during the measurement and we could not get performance results. As for Infiniband, the `csum` component causes an overhead of 30 %. The behavior of NEWMADELEINE on Myrinet is different from the case of Infiniband since the MX driver cannot compute checksums. The parallel checksum computation is thus used in that case. This mechanism degrades the performance by 20 %.

6.2 NAS Parallel Benchmarks

We also run the NAS Parallel Benchmarks on the PLAFRIM testbed. Figure 8 reports the normalized results for class B on 16 processes. In order to evaluate the cost of each fault tolerance mechanisms, NEWMADELEINE results are normalized against vanilla NEWMADELEINE whereas OPEN MPI execution times are normalized against OPEN MPI. Since OPEN MPI `dr` component causes an error on the CG program, its performance is not reported.

The results show that OPEN MPI fault tolerance mechanisms cause a significant overhead for kernels EP, IS and MG while variations of performance for other programs are less important. Using NEWMADELEINE fault tolerance mechanisms cause an significant overhead for BT, LU and SP kernels while the other programs show little variation. In these kernels, most messages require a *rendezvous*; since our fault-tolerance mechanism needs the sender to wait for the acknowledgment, it may decrease the performance. When a process receives a message, it submits an acknowledgment to NEWMADELEINE packet scheduler and wakes up the application. Depending on the communication pattern, the packet scheduler may wait for aggregation opportunities before sending the acknowledgment, delaying the completion of the message at the sender side.

7 Conclusion and future work

The advent of large scale supercomputers composed of hundreds of thousands of components have raised reliability issues. Beside node failures, the interconnection system may suffer from errors leading to data corruption. Exploiting such supercomputers thus requires to use a fault-tolerant communication library.

In this paper, we have proposed various mechanisms for detecting and correcting network failures. We have designed a generic message retransmission mechanism that avoids packet loss when an error happens. Our communication library ensures that network failures are detected quickly, even during heavy computing phases. We also implemented a message integrity checker able to exploit the low level drivers capabilities as well as multicore CPUs for achieving high performance. Our evaluation show that these mechanisms only cause little overhead on most applications performance.

In the future, we plan to study the integration of these techniques in upper layers of the software stack. For instance, parallel filesystems – such as PVFS – that need reliable communication subsystems may also benefit from the message integrity mechanism we proposed.

Acknowledgments– This work was supported in part by the ANR-JST project *FP3C* and the PHC *Sakura*.

References

[1] O. Aumage, E. Brunet, G. Mercier, and R. Namyst. High-Performance Multi-Rail Support with the NewMadeleine Communication Library. In *HCW 2007: the Sixteenth International Heterogeneity in Computing Workshop*, 2007.

[2] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *International Conference on Dependable Systems and Networks (DSN 2002)*.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*.

[4] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4), 2009.

[5] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2000.

[6] R. Graham, S. Choi, D. Daniel, N. Desai, R. Minnich, C. Rasmussen, L. Risinger, and M. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4), 2003.

[7] M. Koop, R. Kumar, and D. Panda. Can software reliability outperform hardware reliability on high performance interconnects?: a case study with MPI over infiniband. In *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008.

[8] M. Koop, P. Shamis, I. Rabinovitz, and D. Panda. Designing high-performance and resilient message passing on InfiniBand. In *International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[9] T. C. Maxino and P. J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1), Jan. 2009.

[10] M. Menth and R. Martin. Network resilience through multi-topology routing. In *The 5th International Workshop on Design of Reliable Communication Networks*.

[11] T. Okamoto, S. Miura, T. Boku, M. Sato, and D. Takahashi. RI2N/UDP: High bandwidth and fault-tolerant network for a PC-cluster based on multi-link Ethernet. In *Parallel and Distributed Processing Symposium (IPDPS 2007)*.

[12] G. Shipman, R. Graham, and G. Bosilca. Network fault tolerance in open MPI. *Euro-Par 2007 Parallel Processing*.

[13] F. Trahay and A. Denis. A scalable and generic task scheduling system for communication libraries. In *IEEE International Conference on Cluster Computing and Workshops (Cluster'09)*.

[14] F. Trahay, A. Denis, O. Aumage, and R. Namyst. Improving reactivity and communication overlap in mpi using a generic i/o manager. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2007.

[15] D. Zwaenepoel and D. Johnson. Sender-based message logging. In *17th International Symposium on Fault-Tolerant Computing*.