High performance checksum computation for fault-tolerant MPI over InfiniBand

Alexandre Denis, François Trahay, Yutaka Ishikawa

High performance checksum computation for fault-tolerant MPI over InfiniBand

Alexandre DENIS[1], Francois TRAHAY[2], and Yutaka ISHIKAWA[3]

[1] INRIA Bordeaux – Sud-Ouest / LaBRI, France
alexandre.denis@inria.fr
[2] Institut Mines-Telecom, Telecom SudParis, 91011 Evry, France
francois.trahay@it-sudparis.eu
[3] University of Tokyo, Japan
ishikawa@il.is.s.u-tokyo.ac.jp

Abstract. With the increase of the number of nodes in clusters, the probability of failures and unusual events increases. In this paper, we present checksum mechanisms to detect data corruption. We study the impact of checksums on network communication performance and we propose a mechanism to amortize their cost on InfiniBand. We have implemented our mechanisms in the NEWMADELEINE communication library. Our evaluation shows that our mechanisms to ensure message integrity do not impact noticeably the application performance, which is an improvement over the state of the art MPI implementations.

Keywords: Checksum, Fault-Tolerance, High-performance networks, InfiniBand

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 failure [4]. Multiple causes of failures exists — software bugs, hardware failures, failed switch, electromagnetic perturbation, faulty cable shielding — leading to various types of failures — crashed nodes, lost packets, data corruption. Communication libraries implement a variety of mechanisms to detect and survive these failures.

We focus in this paper on the detection of data corruption in MPI network communication through the use of checksums.

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 [7]. To ensure end-to-end message integrity from sender memory through receiver memory, communication libraries use *checksums*: the sender computes a checksum of the message to be sent and its headers and sends it with the message headers. The receiver computes the checksum on the received messages; if it 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, the message is considered as lost and the communication library retransmits the packet.

In this paper, we study the impact of checksuming on communication performance and propose mechanisms to amortize their cost on InfiniBand.

The remainder of this paper is organized as follows: Section 2 presents related work. In Section 3, we analyze the cost of checksum on communication performance. Section 4 presents the technique we propose to amortize the cost of checksum computation on InfiniBand. Results are discussed in Section 5 and we draw a conclusion in Section 6.

2 Related work

Some works have focused on the effectiveness [13, 12] of error detection for various checksums algorithms, or on the performance [8, 9] of checksum computation. To our knowledge, these works have not been integrated into any MPI implementation.

Failure detection in MPI relies usually on heart beat technique [2] or on sender-based logging [16] that consist in detecting remote activity through the network. Such techniques detect node or link failures, not data corruption.

LA-MPI [11] and OPENMPI [15] ensure the integrity of messages by computing checksums. This allows to detect corrupted fragments and to retransmit them, but this technique suffers from a large overhead that significantly impacts the performance of applications. Since LA-MPI has been superseded by OPENMPI, in this paper we compare our approach against OPENMPI only.

We have implemented our proposed checksum mechanisms in NEWMADELEINE [1] since it was more convenient for us to work in our own communication library. However, these mechanisms are intended to be generic and not specific to NEWMADELEINE, thus they could probably be implemented in any other MPI implementation.

3 Checksum cost analysis

In this Section, we study the cost of various checksum algorithms and their impact on communication performance.

Computing checksums has a cost that may lower the available bandwidth. The precise cost depends on the checksum algorithm, the compiler, and the CPU. In this paper, we consider the following algorithms: *sum*– plain sum of 32 bits words; *XOR*– XOR all 32 bits words; *Adler-32*, *Fletcher-64* [9], *Jenkins One-at-a-time* [12], *FNV1a* [10], *Knuth hashing*, *MurmurHash2a*, *Paul Hsieh Superfast*– a collection of well-known fast hashing functions that can be used as error-detection (non-cryptographic) checksum; *CRC*– 32 bits CRC computed with SSE 4.2 (non-accelerated CRC is too slow to be considered here). Algorithms *sum* and *XOR* are given as performance reference only, but are not suitable [13] to detect reliably errors on more than one bit; *CRC* is expected to be slow but offers the best error detection; other algorithms are expected to be a good compromise [8].

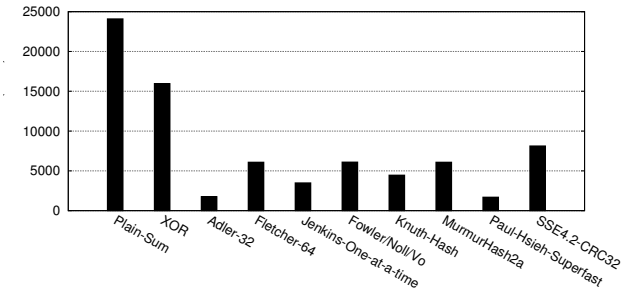


Fig. 1. Bandwidth of some checksum algorithms on 32 kB blocks.

Figure 1 shows the bandwidth of these checksums on our `jack` cluster, equipped with dual-core Xeon X5650 at 2.67 GHz, on 32 kB blocks that fit the L1 cache. The plain *sum* and *XOR* are the fastest, and will likely always be on any hardware. However, it cannot reliably detect corruption beyond a single bit. For a better error detection, *Fletcher-64*, *FNV1a*, *MurmurHash2a* and *SSE4.2 CRC* are good candidates on this particular machine and compiler. They perform around 6 GB/s which makes 1.5 ns/word, *i.e.* 4 cycles per 32-bit word.

We have observed a huge performance discrepancy from one CPU to another, and from one compiler to another, e.g. *Fletcher-64* is 60 % faster with `icc` than with `clang` on Nehalem, and with `gcc` *Fletcher-64* is slower than *FNV1a* on Nehalem but the reverse is true on Dunnington. Therefore we use auto-tuning [3] to choose dynamically the best performing checksum algorithm.

Even when selecting the fastest checksum algorithm, checksum computation has a huge impact on network performance. Let L be the length of a given message, we model the checksum time as a linear function in the form $T_{csum}(L) = \frac{L}{B_{csum}}$, and the network as $T_{net}(L) = \lambda_{net} + \frac{L}{B_{net}}$ with λ_{net} and B_{net} the latency and bandwidth of the network. Both sender and receiver must compute the checksum to ensure data integrity. For a naive approach — the sender computes the checksum, then sends data, then the receiver computes the checksum — the total transfer time is: $T(L) = \frac{L}{B_{csum}} + \lambda_{net} + \frac{L}{B_{net}} + \frac{L}{B_{csum}}$. The apparent bandwidth converges asymptotically towards $\frac{1}{\frac{1}{B_{net}} + \frac{2}{B_{csum}}}$ On the `jack` cluster, we have $B_{net} = 3\,GB/s$ and $B_{csum} = 6\,GB/s$ for *Fletcher-64*, thus the apparent bandwidth of the naive approach is $1.5\,GB/s$ which is 50 % of the network bandwidth. We get results in this order of magnitude on most contemporary hardware.

4 Amortizing the cost of checksum computation

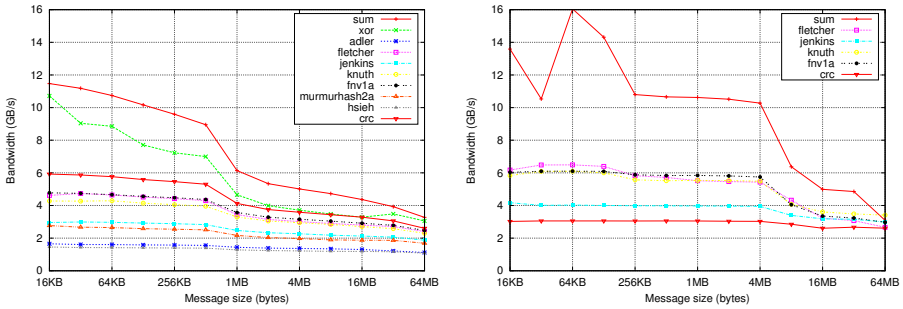
In this Section, we present our approach which consists in amortizing the cost of checksum computation by combining the checksum and the memory copy wherever it happens, and in overlapping computation and network transfer.

We have implemented our mechanisms in the NEWMADELEINE communication library, which decouples upper layers communication requests from network interface.

It applies optimization strategies inbetween in order to use more efficiently the network [1]. For instance, multiple small messages from the application may be aggregated and sent as a single packet on the network. Another optimization consists in using simultaneously multiple links by splitting large messages. To survive network failures, a sender-based logging mechanism [16] was implemented in NEWMADELEINE. When a data corruption is detected, the message sent through the faulty link is retransmitted.

4.1 Combining checksum and memory copy

On the `jack` machine used in the previous Section, 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 and the others are in the same order of magnitude as memory bandwidth. It is then expected that a large part of the cost of a naive approach for checksums will be actually memory access. For multiplexing and to apply optimization strategies, NEWMADELEINE always copies small packets. Even large packets sent with *rendez-vous* over InfiniBand go through a super-pipelined protocol [6] using a copy.



(a) copy, then checksum

(b) combined checksum and copy

Fig. 2. Performance of copy and checksum on cluster `jack`.

We propose to take benefit from these copies to amortize the cost of checksum, *i.e.* reduce the memory accesses needed for checksuming by combining copy and checksum, and overlap memory accesses and checksum computation thanks to CPU being superscalar. We propose to compute the checksum on the fly at every place where data is copied in NEWMADELEINE. Two approaches are possible: copy data then compute checksum, relying on data having been fetched in cache by the copy; combine checksum computation and memory copy, *i.e.* for each word fetch from memory, compute checksum, store at destination.

We have implemented the first approach with the full collection of checksum functions; the benchmarks results are presented in Figure 2(a). We have implemented the second approach with a selection of checksums; the benchmarks results are depicted in

Figure 2(b). We observe that combining the checksum and the memory copy is always beneficial, except for SSE 4.2 CRC where the checksum-only implementation is optimized in assembly where the combined version is written in C with compiler intrinsics for SSE. Once again, we rely on auto-tuning to dynamically decide which version to use.

4.2 Checksums for small messages (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. To add checksuming, we simply change this copy into the combined checksum and copy. 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. Here again we change the copy into a combined checksum and copy.

Let λ_{net} and B_{net} be the latency and bandwidth of the network; $B_{csum+copy}$ the bandwidth of the combined memory copy and checksum computation, then the total transfer time for a message of length L sent with eager mode is $T(L) = \frac{2\times L}{B_{csum+copy}} + \lambda_{net} + \frac{L}{B_{net}}$

On the `jack` cluster, equipped with ConnectX2 InfiniBand QDR HCA, we have $\lambda_{net} = 1.4\,\mu s$; $B_{net} = 3\,GB/s$; $B_{csum+copy} = 6\,GB/s$. Then we can compute the expected overhead of checksums to be 34 % on 4 kB messages. This cost is quite high, but lower than the asymptotic cost since network latency cannot be neglected for small packets. A pipeline to overlap checksum computation and network transfers wouldn't be beneficial since fragmentation overhead would not compensate for the checksum cost on such small packets.

4.3 Checksums for large messages (*rendez-vous*)

Large messages are sent through a *rendez-vous* protocol in NEWMADELEINE. On InfiniBand, we use a variable depth super-pipeline [6] to fetch data into registered memory. We propose to combine the checksum computation with the copy performed by the super-pipeline on both sender and receiver sides. We expect it would amortize the memory transfers needed for checksum, and overlap checksum computation and network transfers.

As depicted in Figure 3, this protocol overlaps copy and RDMA. The chunk size $C_n = q^n$ is growing from chunk to chunk, as a geometric series with a ratio q being equal to the bandwidth ratio between network and copy. The size of the first chunk C_0 is determined so as its copy perfectly overlaps the *rendez-vous* round-trip ($\frac{C_0}{B_{copy}} = 2\lambda_{net}$ computed by auto-tuning). A sub-blocking mechanism amortizes the cost of the copy of the last chunk.

We have shown [6] that the total transfer time of the superpipeline protocol is:

$$T_{superpipeline}(L) = \frac{b}{B_{copy}(L)} + g \times n + \lambda_{net} + \frac{L}{B_{net}}$$

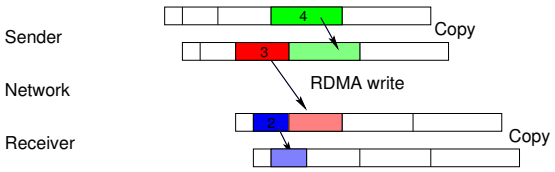


Fig. 3. Super-pipeline for memory copy: a pipeline with a variable chunk size.

with the number of gaps:

$$n = \log_q \left(1 + \frac{L}{C_0}(q-1)\right)$$

and L is the message length, λ_{net} the network latency, B_{net} the network bandwidth, g the gap as in the LogP model [5], q the ratio of the finite geometric series of chunk size, and b the sub-block size. The overhead of this protocol compared to the raw network performance is comprised of: the copy of the first sub-block of size b; the n gaps between packets.

The addition of checksum to the copy has an impact on the first term (copy of the first sub-block) and on q. The impact on the first term consists in the checksum of a 4 KB sub-block, which is half a micro-second on our `jack` cluster with *FNV1a*. The impact on q used as the base of a logarithm is limited, e.g. with the parameters of the `jack` cluster for a 1 MB message, it adds an overhead of one gap, *i.e.* 300 ns. The total overhead of checksuming on this example is less than 1 % according to the theoretical model.

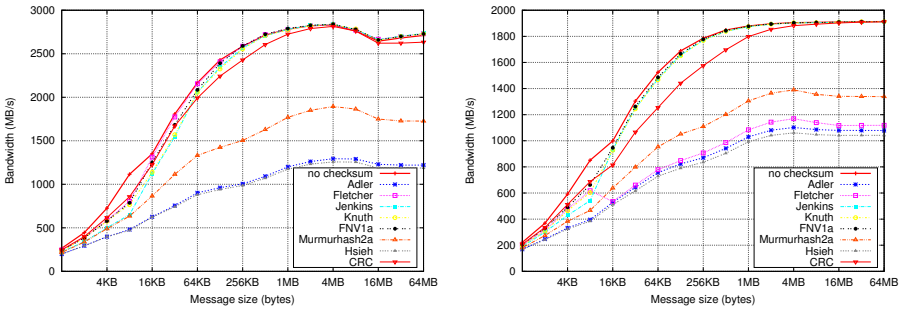
5 Evaluation

In this Section, we present the experimental results obtained by comparing the checksum-enabled NEWMADELEINE with the original NEWMADELEINE and OPENMPI (`ob1` and `csum`). We used MPICH2-nmad [14] as an MPI interface over NEWMADELEINE, and compared against latest stable release OPENMPI 1.4.5. We evaluate the raw overhead of checksums computation as well as their impact on NAS Parallel Benchmarks.

The results we present were obtained on the `jack` and `graphene` clusters. Cluster `jack` is equipped with dual-core Xeon X5650 at 2.67 GHz and ConnectX2 QDR (MT26428) InfiniBand; compiler is `icc` 12.1. Cluster `graphene` features ConnectX DDR (MT26418) InfiniBand cards on quad-core nodes equipped with Intel Xeon X3440; compiler is `gcc` 4.4.

5.1 Raw checksum overhead

We used Netpipe to measure the raw MPI performance on InfiniBand on both clusters. Bandwidth results for NEWMADELEINE with various checksum algorithms are de-



(a) Cluster `jack`, QDR (b) Cluster `graphene`, DDR

Fig. 4. NEWMADELEINE bandwidth with various checksums algorithms

picted in Figure 5.1. On both clusters, for small packets before the *rendez-vous* threshold (16 KB), the impact of checksums is quite high, around 30 %, which is consistent with our model in Section 4. For these packet sizes, there is no pipelining nor any mechanism to amortize the cost of checksuming except the combination of copy and checksum. The performance of these combined operations cannot be higher than the peak checksum performance, which is much lower than copy for such packet size that fit the cache.

For messages larger than 16 KB, the bandwidth overhead ranges from 3 % for 64 KB to less than 0.5 % asymptotically for the fastest checksum algorithms. *FNV1a* is a sensible default choice on most machines and compilers if auto-tuning has not been performed yet, but auto-tuning may still improve performance by a few percents, e.g. *Fletcher* is 2 % faster than *FNV1a* on cluster `jack` (but *Fletcher* is 40 % slower on `graphene`).

We compared our checksum-enabled MPI implementation against OPENMPI. The bandwidth results are depicted in Figure 5 and 6. On cluster `jack` (Figure 5), NEWMADELEINE and OPENMPI get roughly the same bandwidth without checksums. When checksums are enabled, the bandwidth is lowered by 20 % for OPENMPI, and by at most 3 % for NEWMADELEINE, thanks to the super-pipeline protocol. On cluster `graphene` (Figure 6), OPENMPI is slightly faster than NEWMADELEINE when checksums are disabled. When checksums are enabled, OPENMPI suffers a performance drop of 60 % while the overhead is below 2 % for NEWMADELEINE.

5.2 NAS Parallel Benchmarks

We also run the NAS Parallel Benchmarks on the `graphene` cluster. Table 1 reports results for class B on 16 nodes. We report raw performance results (median time from 10 runs) as well as time differences as percentage.

The results show that OPENMPI is slightly faster than MPICH2-nmad when checksums are disabled. This can be explained by NEWMADELEINE optimization strategies

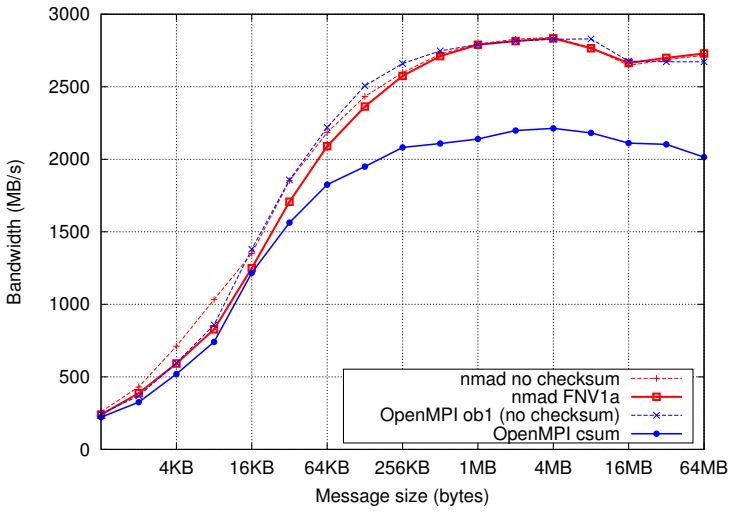


Fig. 5. Bandwidth over QDR InfiniBand on cluster `jack`

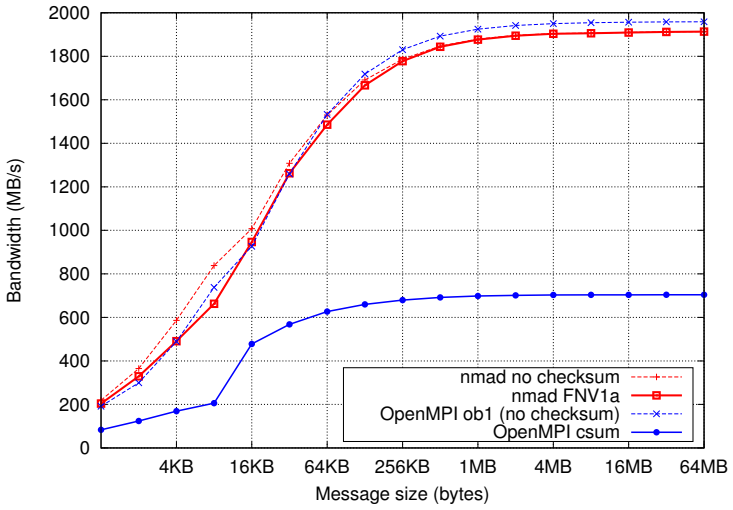


Fig. 6. Bandwidth over DDR InfiniBand on cluster `graphene`

	is.B.16	lu.B.16	ft.B.16	cg.B.16	mg.B.16
MPICH2-nmad (no checksum)	0.37 s	18.54 s	5.06 s	5.72 s	0.71 s
MPICH2-nmad *FNV1a*	0.37 s	18.57 s	5.05 s	5.69 s	0.72 s
OPENMPI `ob1`	0.35 s	17.89 s	4.89 s	5.60 s	0.71 s
OPENMPI `csum`	0.43 s	19.30 s	5.45 s	6.59 s	0.79 s
OPENMPI `csum` / OPENMPI `ob1`	+22.86%	+7.88%	+11.45%	+17.68%	+11.27%
MPICH2-nmad *FNV1a* / MPICH2-nmad no checksum	+0%	+0.16%	-0.20%	-0.52%	+1.41%
MPICH2-nmad *FNV1a* / OPENMPI `csum`	-13.95%	-3.78%	-7.34%	-13.66%	-8.86%

Table 1. NAS results on cluster `graphene`

causing a longer software stack, thus a higher latency, with no gain when there is a single communication flow as in the NAS Parallel Benchmarks.

When checksums are enabled, OPENMPI suffers a performance penalty from 7 % to more than 22 %. On the other hand, enabling checksums in MPICH2-nmad (*FNV1a* is selected by auto-tuning here) has a negligible impact on performance.

When comparing both checksum-enabled OPENMPI and MPICH2-nmad, MPICH2-nmad is faster by 3 % to 14 %. This demonstrates that our approach to amortize the cost of checksum computation is competitive.

6 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. The classical solution to detect such errors is the use of checksums, which have an impact on network performance.

In this paper, we have proposed a mechanism that amortizes the cost of checksum computation in MPI implementations for InfiniBand. We have implemented and evaluated this mechanism. Our evaluation shows that it causes a performance degradation of at most a few percents in the worst case for micro-benchmarks, and the difference is negligible on NAS benchmarks. This is a huge improvement over the state of the art.

In the future, we plan to study the integration of these techniques in upper layers of the software stack. For instance, parallel file systems – 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.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

1. Aumage, O., Brunet, E., Furmento, N., Namyst, R.: NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In: CAC 2007: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007. http://hal.inria.fr/inria-00127356
2. Bertier, M., Marin, O., Sens, P.: Implementation and performance evaluation of an adaptable failure detector. In: International Conference on Dependable Systems and Networks (2002)
3. Brunet, E., Trahay, F., Denis, A., Namyst, R.: A sampling-based approach for communication libraries auto-tuning. In: International Conference on Cluster Computing (IEEE Cluster). pp. 299–307. IEEE Computer Society Press, Austin, Texas (Sep 2011), http://hal.inria.fr/inria-00605735/
4. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience. International Journal of High Performance Computing Applications 23(4) (2009)

5. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: Logp: towards a realistic model of parallel computation. In: ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 1–12. PPOPP '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/155332.155333
6. Denis, A.: A high performance superpipeline protocol for infiniband. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Proceedings of the 17th International Euro-Par Conference. pp. 276–287. No. 6853 in Lecture Notes in Computer Science, Springer, Bordeaux, France (Aug 2011), http://hal.inria.fr/inria-00586015/
7. Dinaburg, A.: Bitsquatting, DNS hijacking without exploitation. In: Black Hat Conference (Jul 2011)
8. Feldmeier, D.C.: Fast software implementation of error detection codes. IEEE/ACM Trans. Netw. 3(6), 640–651 (Dec 1995), http://dx.doi.org/10.1109/90.477710
9. Fletcher, J.: An arithmetic checksum for serial transmissions. IEEE Transactions on Communications 30(1), 247 – 252 (jan 1982)
10. Fowler, G., Noll, L.C., Vo, K.P., Eastlake, D.: The FNV non-cryptographic hash algorithm. IETF Internet-draft (Mar 2012)
11. Graham, R., Choi, S., Daniel, D., Desai, N., Minnich, R., Rasmussen, C., Risinger, L., Sukalski, M.: A network-failure-tolerant message-passing system for terascale clusters. International Journal of Parallel Programming 31(4) (2003)
12. Jenkins, B.: Hash functions. Dr Dobb's Journal (Sep 1997)
13. Maxino, T.C., Koopman, P.J.: The effectiveness of checksums for embedded control networks. IEEE Transactions on Dependable and Secure Computing 6(1) (Jan 2009)
14. Mercier, G., Trahay, F., Buntinas, D., Brunet, É.: NewMadeleine: An Efficient Support for High-Performance Networks in MPICH2. In: Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09). IEEE Computer Society Press, Rome, Italy (May 2009), http://hal.archives-ouvertes.fr/hal-00360275
15. Shipman, G., Graham, R., Bosilca, G.: Network fault tolerance in open MPI. Euro-Par 2007 Parallel Processing
16. Zwaenepoel, D., Johnson, D.: Sender-based message logging. In: 17th International Symposium on Fault-Tolerant Computing