Message Relaying Techniques for Computational Grids and their Relations to Fault Tolerant Message Passing for the Grid

Michael Cadilhac, Thomas Herault, and Pierre Lemarinier

Laboratoire de Recherche en Informatique [⋆]
INRIA Futus - Equipe Grand-Large
Universite Paris XI, 91 405 Orsay
(cadilh,herault,lemarini)@lri.fr

Abstract. In order to execute without modification Message Passing distributed applications on a computational grid, one has to address many issues. The first to come is how let processes of two different clusters communicate. In this work, we study the performances of relaying techniques (passing messages to a middle-tier) to solve this issue. When using relays, messages and most of the nondeterministic behavior of nodes pass through the relays during the execution. This provides the ability to implement fault tolerance at the relay level using pessimistic message logging techniques. We also evaluate the overhead of this logging and study how relays should be designed and fault tolerance protocols composed to provide a full fault-tolerant Message Passing Interface library for computational grids.

1 Introduction

Message Passing is a paradigm widely used to design and implement communication-bound distributed applications. Computational grids aggregate entities (typically high-performance clusters) distributed over the Internet to build a parallel virtual machine. When porting message passing libraries to the computational grid, like MPICH [1] or other implementations of the MPI standard [2], one has to address two new issues: first the Relaying Techniques and second the Fault-Tolerance.

A major issue introduced by the grid is how communications between entities are handled. Each entity (cluster) is administrated independently and is generally protected from the rest of the Internet, for example behind firewalls. When such entities are assembled to build a computational grid, it is necessary to implement relays of communication between the clusters. These relays provide two features: they limit the number of nodes to monitor for the security, and they limit the number of connections to handle simultaneously. Potentially, in a message passing application, a process may communicate directly with all the

[⋆] This work uses the Grid5000 experimental grid founded by the ACI Masse de donnees from the french ministry of research others. Clusters typically handle this by allowing a process to establish a connection with all the nodes. For very large clusters and for grids, this approach introduces a system bottleneck: operating systems may introduce a non negligible overhead when multiplexing many simultaneous connections. Relays may be used to decrease this number of connections by merging the streams into a single one between two relays.

A typical Computation grid consists in a set of high performance clusters (including high performance networks) linked together through a lower performance network (e.g. ATM Internet links). The cost of a communication between clusters is much higher than the cost of a communication inside a cluster. Thus, adding another internal component through which inter-cluster communications must pass may not add measurable overhead. For small messages, the latency overhead is not significant simply because of the high difference between inter-cluster and intra-cluster communication. However, special care must be taken for long messages, and the implementation must ensure that pipeline techniques are used. Another implementation issue is to handle efficiently the available bandwidth.

Fault-Tolerance is a capital concern for message passing libraries over a grid, because of the high number of components, which decreases the mean time between failures of the whole set, and because of the heterogeneity of the components and their distributed administration. A typical workaround used today against the first problem is to calibrate the duration of the experiment for avoiding with high probability the hit of a failure during the execution. This calibration assumes a certain reliability of the hardware, or at least a failure behavior that is well described with a probabilistic law. Grids are less predictable by nature: they gather heterogeneous components from different administrations, and a user may be less aware of the reliability of a cluster she does not use frequently. Thus, this calibration phase is more difficult and instead of trying to avoid failures, one may want to cope with them.

A fault tolerant technique envisioned for large scale systems like grids [3] is based on message logging. It consists in saving information on communications and other non deterministic events during the execution, then in case of crash using this information to mend the failure. This information, and in particular the message payload, can be logged by the relays for the messages between clusters.

In this work, we study the impact of relays on message passing applications built over the MPICH-1.2.7 library, how relays should be implemented to introduce the smallest overhead, and how they can be used to help tolerating failures. This article is organized as follows: in the next section, we discuss previous works on relaying techniques and fault tolerance for message passing systems; in section 3, we present the common architecture for our relays, how they can be used to help fault tolerant protocols, and different relay implementations in section 4. Then, in section 5, we study and compare their performances, and conclude in section 6.

2 Related Work

Relaying techniques have been longly studied. As a basic principle to transmit some information from one point to another, routers and network gateways can be seen as relays. However, they operate at the packet level (usually at the IP level for Internet communications), and this level precludes passing different network kinds or logging efficiently. In this work, we focus our study on relaying technique at the MPI-message level.

Modern operating systems provide two main techniques to design a communicating application like a relay: multiplexing and multithreading/multiprocessing [4, 5]. On one hand, in a typical multiplexing-based network application, non-blocking communication system calls are issued by the application, which uses a single blocking system call to wait for observable events simultaneously on multiple file descriptor (e.g. select or poll and non blocking read and write). On the other hand, a typical multithreaded-based network application uses only blocking system calls to communicate, and multiple threads to ensure that lack of events from a file descriptor will not pertain another active file descriptor to be used. Usually, the application has one or more thread per communication stream, and uses the scheduling ability of the operating system to multiplex the different streams and provide fairness to the communication. Hybrid solutions mixing multiplexing and multithreading are also used, aiming at more efficiency.

ssh [6] can be seen as a message-level relay. Although it provides relaying for TCP connections through its tunnel functionality, messages and not packets are relayed through the tunnel. Its current implementation in OpenSSL [7] is purely multiplexed. It uses a single thread and a single process. Since all communications eventually come through a single TCP stream, this is the most efficient approach. This position is held by many other relay implementations, like [8].

Relaying messages in a MPI application enforces however to provide more functionality: messages should be represented canonically (a coherent bit order of integer and other meaningful data provides the possibility to execute the application over heterogeneous components); identifiers are abstracted at the MPI level as continuous rank numbers, which must be translated into identifiers at the communication level (e.g. IP addresses); multiple network medium should be supported by the relay to use the most efficient network medium between two peers.

MPICH-G [9] is the first work to provide all these functionalities. It uses the Nexus [10] toolkit for multiple-networks communication methods. This work also focuses on the integration of the other grid-specific issues, like co-allocation of nodes, authentication, executable staging. More integration with the Globus [11] toolkit was done in MPICH-G2 [12]. For the relaying issue, the authors address the performance hit introduced by the Nexus toolkit which introduces too many copies and data manipulation. However, the techniques used to gain the performance introduces many nondeterministic choices which inhibit the fault-tolerance feature of the relay we present in this work.

A notion related to the relay we present here, the tuple-space [13], can be used to deposit data in a shared replicated space and retrieve it. However, the approach is very far from the communication pattern used in an MPI application, and significant overhead is to be expected.

Automatic and transparent fault tolerant techniques for message passing distributed systems have been studied for a long time. We can distinguish few classes of such protocols: replication protocols, rollback recovery protocols and self-stabilizing protocols. In replication techniques, every process is replicated f times, f being an upper bound on the number of simultaneous failures. As a consequence, the system can tolerate up to f concurrent faults but divides the total computation resources by a factor of f. Although this technique is well-suited for RPC-based Grid Computing, in a MPI-based computation, this would lead to a complete replication of the distributed application, which may be to expensive.

Self-stabilizing techniques are used for non terminating computations, like distributed system maintenance. In this work, we focus on rollback recovery protocols which consist in taking checkpoint images of processes during initial execution and rollback some processes to their last images when a failure occurs. These protocols take special care to respect the consistency of the execution in different manners. Rollback recovery is the most studied technique in the field of fault tolerant MPI. Several projects are working on implementing a fault tolerant MPI using different strategies. An overview can be found in [14].

Rollback recovery protocols include global checkpoint techniques and message logging protocols. Extended descriptions of these techniques can be found in [15]. Three families of global checkpoint protocols have been proposed in the literature. The first family gathers uncoordinated checkpoint protocols: every process checkpoints its own state without coordination with other processes. When a fault occurs, the crashed processes restart from previous checkpoints. Processes that have received a message from a rolled back process have also to rollback if this message was initially received before the checkpoint image of this process. This may lead to a domino effect where a single fault makes the whole system rollback to the initial state. As a consequence, this kind of protocols is not used in practice.

The second family of global checkpoint protocols gathers the coordinated checkpoint protocols. Coordinated checkpoint consists in taking a coherent snapshot of the system at a time. A snapshot is a collection of checkpoint images (one per process) with each channel state [16]. A snapshot is said to be coherent if for all messages m from process P to process Q, if the checkpoint on Q has been made after reception of m then the checkpoint on P has been made after emission of m. When a failure occurs, all processes are rolled back to their last checkpoint images.

The third family of global checkpoint protocols gathers Communication Induced Checkpointing (CIC) protocols. Such protocols try to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, they piggyback causality dependencies in all messages and detects risk of inconsistent state. When such a risk is detected, some processes are forced to checkpoint. While this approach is very appealing theoretically, relaxing the necessity of global coordination, it turns out to be inefficient in practice. [17] presents a deep analysis of the benefits and drawbacks of this approach. The two main drawbacks in the context of cluster computing are 1) CIC protocols do not scale well (the number of forced checkpoints increases linearly with the number of processes) and 2) the storage requirement and usage frequency are unpredictable and may lead to checkpoint as frequently as coordinated checkpoint.

Message logging consists in forcing the reexecution of crashed processes from their last checkpoint image to reach the state immediately preceding the crashing state, in order to recover a state coherent with non crashed ones. All message logging protocols suppose that the execution is *piecewise deterministic*. This means that the execution of a process in a distributed system is a sequence of deterministic and non deterministic events and is led by its nondeterministic events. Most protocols suppose that the reception events are the only possible non deterministic events in an execution. Thus message logging protocols consists in logging all reception events of a crashed process and replaying the same sequence of receptions.

There are three classes of message logging protocols: pessimistic, optimistic and causal message logging. Pessimistic message logging protocols ensure that all events of a process P are safely logged on reliable storage before P can impact the system (send a message) at the cost of synchronous operations. Optimistic protocols assume faults will not occur between an event and its logging, avoiding the need of synchronous operations. As a consequence, when a fault occurs, some non crashed processes may have to rollback. Causal protocols try to combine the advantages of both optimistic and pessimistic protocols: low performance overhead during failure free execution and no rollback of any non crashed process. This is realized by piggybacking events to message until these events are safely logged. A formal definition of the three logging techniques may be found in [18].

In this work, we evaluate the performance cost of introducing pessimistic message logging in a relay-based Grid MPI.

3 Architecture

In this work, we consider computation grids which are independent high performance clusters bound by Internet links. Each cluster consists in a set of homogeneous computers (although this is not needed by our architecture, this is likely), communicating through a high performance network (like Gigabit Ethernet, Myricom, etc...). Communication between clusters pass through gateways (at least two) using a slower network (e.g. ATM).

3.1 Components of typical deployments

Figure 1 represents a typical deployment of the relay architecture. Relays are software component which runs on one computer of each cluster. They use the routing capability of the gateways to communicate with each other. Computing nodes of the same cluster communicate with the relay of this cluster through the high-speed network (high-speed networks are usually fully connected). Compared to Internet gateways, which route IP packets, relays communicate at the message level (application messages). This provides the capability to communicate over different network kinds (like sock-GM and IP over ethernet for example).

Internet gateways of the cluster are also used as firewall to implement a security policy. Confining the inter-cluster communication to relays limits the number of nodes which should be granted access to Internet. Moreover, relays can be coupled with secure tunnels to crypt the communications going through Internet and authenticate the peers.

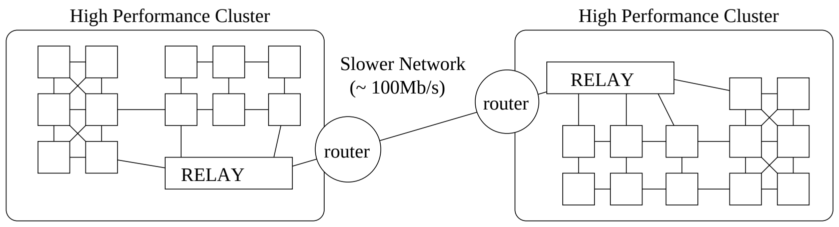


Fig. 1. Typical computation grid with relays

To evaluate different implementations of relays, we implemented a prototype inside the MPICH-1.2.7 message passing library. Figure 2 presents the software stack of the implementation and of the reference implementation P4 over TCP. At the highest level, a user communicates through the MPI language. The MPI routines are translated to point-to-point communications in the ADI. Then, messages are exchanged according three policies: short (the payload is included in the message), eager (the payload follows a control message) and rendez-vous (control messages are exchanged to synchronize the emitter and the receiver, then the payload is transmitted). MPICH devices implement the point-to-point communication over a medium. P4 is the reference device driver for communications over TCP. P4 comes in two parts : a driver inside the library and a communication daemon, which is part of the runtime. TCP communications are done between the communication daemons.

We implemented a specific new device, the R-device, to be used in a relay deployment. Compared to P4, it does not use a communication daemon but communicates directly with the relay. Moreover, as this work aims to evaluate the performance of the relay, we do not provide direct connection for intra-cluster communications. The R-device implements the minimal set of communication routines to provide a fully-functional MPI library, that is blocking emission and reception, probe of messages, initialization and finalization. Asynchrony and multiplexing of communications is provided at the relay level.

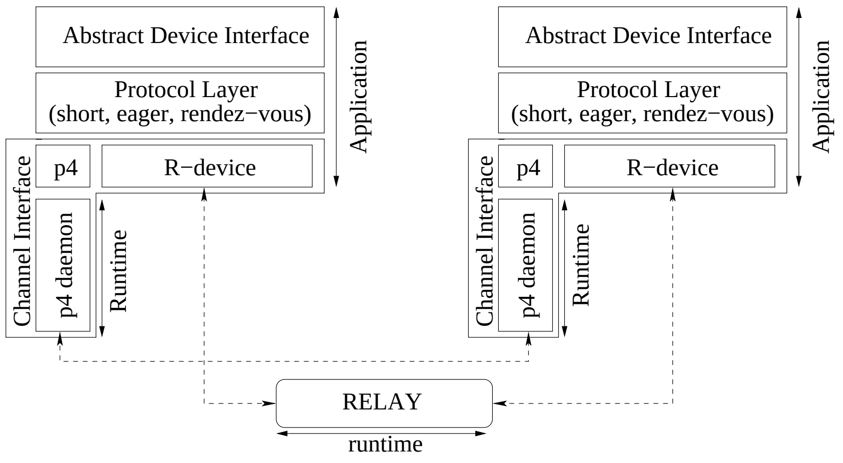


Fig. 2. Software stack of MPICH-relay and MPICH-P4

All computing nodes communicate only with relays. Each node has a preferred relay, which handles the messages sent to the node. Multiple nodes may use the same relay as a preferred relay. A typical Grid deployment consists in a single relay per cluster, and every node of this cluster use this relay as a preferred relay. For load-balancing usage, multiple relays may be used in a single cluster.

To communicate with another node, the emitter sends the messages to the preferred relay of the destination. This means that every node has to communicate potentially with every relay. Since relays are located in different clusters, we use ssh tunnels to emulate each relay into each cluster. The ssh tunnels are located on the same node as the relays (see figure 3).

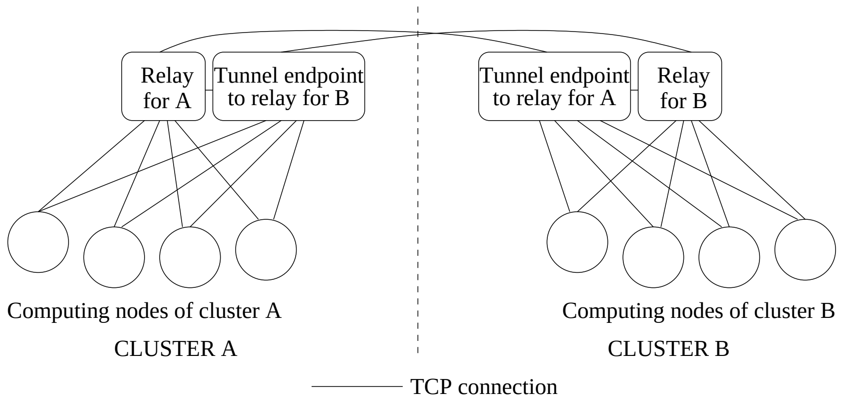


Fig. 3. TCP connections between nodes, relays and tunnels

Each node is connected to the preferred relay of the other nodes. When a process A has to send a message to a process B, it calls the blocking send function of the driver. This function sends the message to the preferred relay of B. When A needs to receive a message, it calls the blocking reception function of the driver. This function emits a request to the relay of A, which does the matching (finding the corresponding available message). If the matching is successful, the message is sent back to A. If the matching fails (there is no message with the expected type or source), the relay waits for a match-message and send it back to A. To ensure progress of the communication with these two blocking routines, a MPICH driver has to implement a third function: probe. When the MPI level can choose between multiple messages, or between computation and communication, it issues a probe. The probe function of the driver forwards the request to the preferred relay, which determines if a matching is possible at this time. If it is, an acknowledge is sent back, if it is not possible, a negative acknowledge is sent back. The two last functions implemented in the driver are called at initialization and finalization.

3.2 Enabling fault-tolerance using relays

This mechanism provides the ability to implement pessimistic message logging under the assumption that the relays will not be subject to failure. Pessimistic message logging technique is a rollback-based fault tolerant technique which does not enforce synchronization of the checkpoints. Each process can checkpoint at any time, and if a process crashes, it is rolled-back to its last checkpoint. Then, the system has to ensure that the rolled-back process will reach a state that is coherent with the current state of the other processes (a rolled-back process may be in a previous state far from the current state of the other processes in terms of nondeterministic actions). In particular, according its algorithm, the rolled-back process may send messages that were already received, and expect messages that were already sent. The system has to enforce two goals: 1) the rolled-back process must not be able to influence other processes while it didn't recover the state just before the crash, and 2) it must execute exactly the same internal actions and receptions it did in the first execution, to ensure that it will reach this state.

In order to do this, we assume that an execution is an alternate sequence of deterministic phases and nondeterministic actions. If a process executes a non-deterministic action, it then follows its algorithm and will always execute the corresponding deterministic phase. To implement a pessimistic message logging algorithm, one has to log every nondeterministic event in a reliable media before they impact the process and to ensure that when replaying, the same nondeterministic actions will be executed at the end of every deterministic phases. Applied to MPI applications and libraries, this mainly means to log messages receptions and probes, which are the only nondeterministic actions. A blocking reception is nondeterministic because a process may receive from two sources at a given time, and may deliver from different sources on two executions. A probe is non deterministic since according to the asynchrony of the network, a message may not be available at the same time.

Since all these actions (reception and probe) pass through a relay, we may use relays to replay the execution, that is to log every message-related nondeterministic action during the initial execution, then emulate the same nondeterministic choices in a rollback phase. During the initial execution, the preferred relay of a process logs all its receptions and probes. It keeps a copy of the message payload, and remembers the deliveries and probes order in a fully ordered list. Each event is named uniquely by its type (probe or reception) and a few other integers. We use a specificity of the MPICH implementation to reduce the number of information to store. After an unsuccessful probe, the driver will never issue a blocking receive, and after a successful probe, the driver will always issue a blocking receive. Thus, we do not maintain a full event for probes, but count the number of unsuccessful probes between deliveries. Thus, an event is defined by its source, destination, sequence number of delivery (implicit in the linked list) and number of unsuccessful probes before this reception. During this execution, the driver also maintains a vector of counters to know how many message it sent to each peer, how many messages it received, and how many probes it has done since the last reception. These counters are saved with every checkpoint image.

When a process crashes, the runtime environment detects this failure and launches the crashed process again on a spare resource. We use a slightly modified version of the fault-tolerant mpirun developed for clusters in this case. The process rolls-back using its checkpoint image stored on a checkpoint server, and connects to its preferred relay. It sends its counters to the relay, which compare them to its known state for this rank. It moves the pointers of next message to send to the crashed process back in its history to the next missing message, then answers back with a vector of counters, which holds the number of outgoing messages per rank that the driver must emulate and not execute (that is, the driver makes a fake blocking send for these messages). This ensures that the process will not interfere with the system during its rollback. Then, the relay waits for probes and blocking receives from the driver. Each time it receives a probe request it sends a negative acknowledge until reaching the number of probes logged in the next message to deliver (although the next message to send is most probably already available). Experiment demonstrated that this is mandatory since the MPI library may expect another message if the probes are not unsuccessful. Then, when the number of probe failures is zero, the relay may accept a reception request or a probe request. In case of probe, it answers yes and expect a reception request. In case of reception, it sends the logged message again, and the process continues its execution up to the state just before the crash. Then, the relay continues the execution according the algorithm in the absence of failures.

4 Implementations

The relay is a key point of the architecture. Its implementation must be well designed and suited to two tasks: 1) multiplex many communication channels with the most efficiency, 2) log messages.

For addressing the first issue, one can find four general approaches in the literature:

1. This first approach is to implement a single-threaded single-process application multiplexing all the communication streams with select-like system calls;
2. at the other extreme, java-like approach is to create a thread per connection and let the operating system scheduler handle the multiplexing of all the communications;
3. the first approach can be derived with an initial pool of threads which handle a subset of the communications, ensuring that at each time one thread at least observes the idle communication links;
4. the last approach is to design a pipeline of specialized tasks (reception, logging, emission) and to implement a multi-threaded application where pools of threads handle each a specialized action.

We implemented the four approaches and evaluated them in the performance evaluation section. We describe here with more details the implementation of each of the approaches.

In every implementation, logging is done by keeping a copy of every messages in the virtual memory of the process and ordering them into linked lists. We experimented logging using ad-hoc storing backups and files, but the performance gain is not significant. Indeed, the caching policy of the linux kernel prevents from having a strong control on the amount of free memory, and even when using files and flushing them to log the messages, the overhead of page replacement has to be paid when the physical memory is exhausted. However, the page replacement algorithm of the linux kernel is efficient enough to use the virtual memory system to store the logs.

4.1 Single process multiplexing streams

In this method, a single process with a single thread uses a multiplexing system call to observe events on all the file descriptors. All sockets are made non blocking at creation time, and every input/output operation is done only when some data will be transmitted. The MPICH driver implements only blocking communications. This introduces a lot of very small messages (50 bytes), the control messages. In order to ensure that these messages are processed as soon as possible, every TCP socket has its Nagle disabled using the unix socket option `TCP_NODELAY`. A performance hit of 1000 times slower was measured if this is not done.

A producer/consumer algorithm is implemented between source and destination. A socket is observed for output only if the producer algorithm has received enough data for the destination and if the destination sent a request for reception. The `poll` system call is used to implement the multiplexing. Since the relay reacts only on communications, no other action is done and the application sleeps until an input/output operation at least is feasible.

This implementation is straightforward and was stable first. This is contradictory with the general belief that multi-threaded applications (one thread per connection) are the most easily implemented by developers.

4.2 One system thread per connection

The second implementation of the relay takes an opposite point of view. Here, the application uses a multi-thread design, with one system thread per stream. Posix threads were used for this implementation.

Sockets are left blocking, and most of the time a thread sleeps in the `read` system call, awaiting for requests from its connected driver. We used the fact that the driver uses only blocking communication routines. A process may receive a message only after it requested it with a blocking receive which produces an emission of request from the driver to the relay. Thus, even if a driver A sends a message to the relay with destination a driver B, the relay may pass the message to the driver B only when it requested it.

Thus, the producer/consumer algorithm between emitters and receivers does not have to interrupt the blocking communications. As long as a driver sends a message, this message is stored. If the destination requested for the reception, conditionals are used to wake up the thread of the destination and begins the emission. When a thread finished sending all the data of a message, it begins its request reception loop.

4.3 Pool of threads multiplexing streams

The first implementation does not use multiple CPUs when they are available. Moreover, meanwhile the processing of an input/output more messages can come on other sockets. To ensure that the application receives every request or data and sends them as soon as possible, one can use a pool of threads.

Each thread implements the same algorithm as the monothreaded solution. However, only one thread at a time is left entering the poll system call using semaphores. When a thread leaves the poll system call, it computes which sockets are active for the corresponding observed events. It removes these sockets from the observed set and let another thread enter the `poll` system call. This ensures that the minimum time is spent with no process observing network events.

The producer/consumer algorithm which implements the pipeline of the communication has to be a modified version of the monothreaded application. Computing the observed set for writing is a more complex tasks and depends on the operations pursued by many threads.

At the time we write this article, the implementation of this method is done, but we did not have time to evaluate experimentally its performance.

4.4 Specialized threads multiplexing streams

The last implementation we compare with the others is fully oriented producer/consumer. We distinguish three roles in a relay: A) receiving requests and data, B) logging of messages and computing the answers and C) sending answers. This fourth method introduces three pool of threads, one for each role.

The pool B which consists in logging messages and computing the answers is reduced to zero, because of the early experiments on logging and because almost no computation has to be done between reception and emission. Its role is taken over by the reception thread which log messages at reception in the virtual memory system and computes the answer, which is a negative acknowledge for probes, and an order of emission for the pool C of threads for receptions.

Each pool is of a given size, which is an argument of the relay application. The set of sockets is distributed among the different threads of each pool, and a multiplexing system call (`poll`) is used as the waiting point of each thread. Writing threads also wait for conditionals implementing the producer/consumer algorithm.

Since a thread may have to send messages when another of its sockets also needs to send a message, we used a signal mechanism between the threads of pool A and the threads of pool C to interrupt the polling or emission of one writing thread when a new message is to be sent in a new socket.

Like in the monothreaded implementation, all sockets are made non blocking at creation and the `poll` are the only blocking points of the execution of each thread.

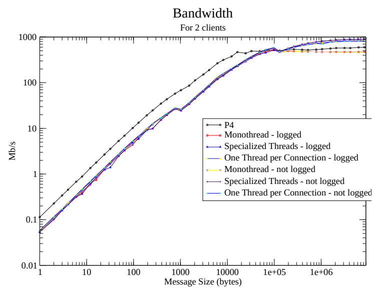
5 Performance evaluation

In this section, we present the performance evaluation and comparison of the four implementations of relays. The experiments come in two sets: a set of experimental measurement of the raw performances of the implementations in a single cluster, and a set of experimental measurement of the global performances of the different implementations in a grid.

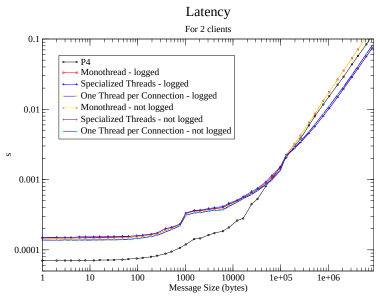
5.1 Raw performances in a cluster

For these experiments, we used the cluster Grid Explorer. Grid Explorer is a high performance cluster dedicated to experiments in computer science and Grids founded by the French ministry through the "Masse de donnees" action of the CNRS. It consists in 216 bi-opteron 2.0GHz with 2GB of RAM and IDE hard drive. Each node is connected to the others through 2 Gigabit Ethernet cards (only one of them was used in all the following experiments). Experiments were run on the Linux operating system version 2.6.8-11 and the applications were compiled using gcc version 4.0.2 (libc 2.3.5). Classical optimization options -O3 was used.

The first results we presents were done using the NetPIPE [19] tool version 4. NetPIPE is a well known tool for bandwidth and latency measurement over many kind of networks and communication paradigm, including native TCP and MPI. This version features the possibility to run simultaneously communication between many pairs, which is mandatory for our experiments.



(a) Bandwidth comparison, 2 clients

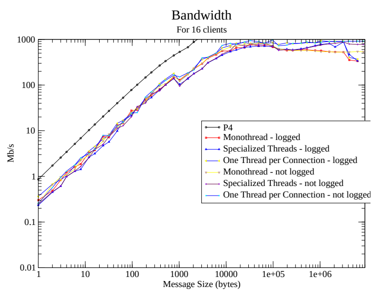


(b) Latency comparison, 2 clients

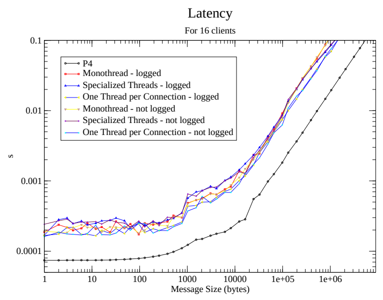
Fig. 4. Bandwidth and latency comparison for all implementations, and two clients on the same relay

Figure 4, 5 and 6 presents bandwidth and latency measurements for respectively 2, 16 and 64 nodes using only one relay. Contrary to P4, when using a single relay every communication pass through a single card. For 2 nodes, P4 and our implementations have the same theoretical maximal bandwidth, but when dealing with 16 or 64 nodes, P4 has a maximal bandwidth of around respectively 8 and 32 times the maximal theoretical bandwidth of a single card. THe latency for P4 for different number of nodes does not change since NetPIPE exchange messages only between fixed pair of nodes (between rank 0 and rank N, 1 and N-1 and so on). When a relay has to deal with only 2 nodes, every implementation has quite the same performance: due to blocking communications the relay has to treat only one communication at a time.

Figure 7 and 8 presents the bandwidth for three sizes of messages and the latency for 1 byte messages of the different relay implementations when respectively enabling and disabling the logging and for different number of nodes (from 2 to 64). The bandwidth presented here is the bandwidth as measured by NetPIPE, which is the overall bandwidth of the application. So, for 64 nodes, which represents 32 pairs (32 nodes send messages of increasing size and 32 receive them), the theoretical maximal bandwidth should be at most 32 Gb/s. However, since all communications pass through a single relay, the theoretical maximal bandwidth is limited to 1 Gb/s (limitation of the Gigabit Ethernet card of the relay). For small and medium message sizes, the monothreaded implementation has around the same performance as multithreaded implementations: the

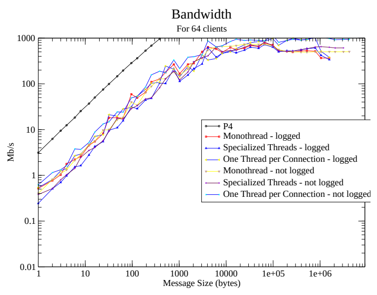


(a) Bandwidth comparison, 16 clients

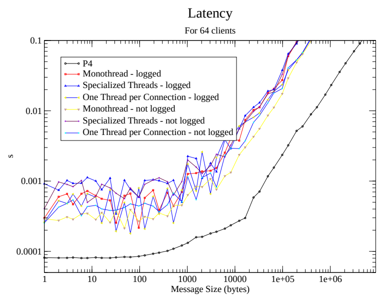


(b) Latency comparison, 16 clients

Fig. 5. Bandwidth and latency comparison for all implementations, and sixteen clients on the same relay



(a) Bandwidth comparison, 64 clients



(b) Latency comparison, 64 clients

Fig. 6. Latency comparison for all implementations, and sixty-four clients on the same relay

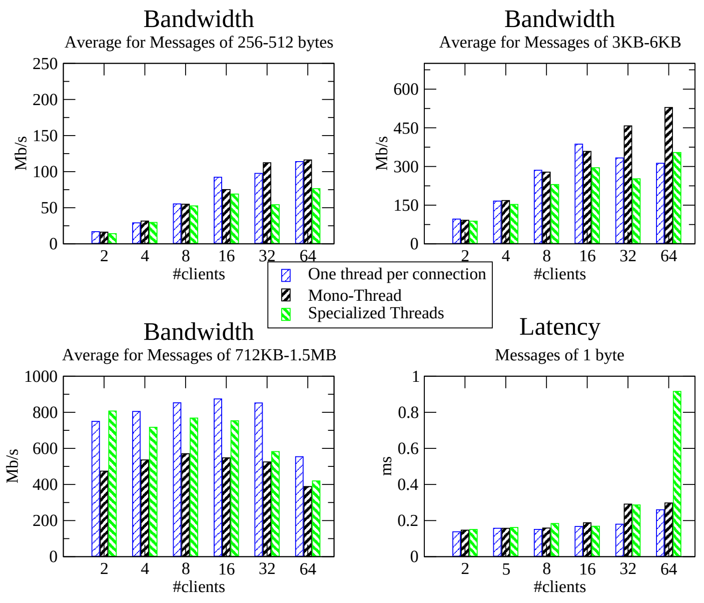


Fig. 7. Comparison of bandwidth and latency for all the implementations with logging enabled

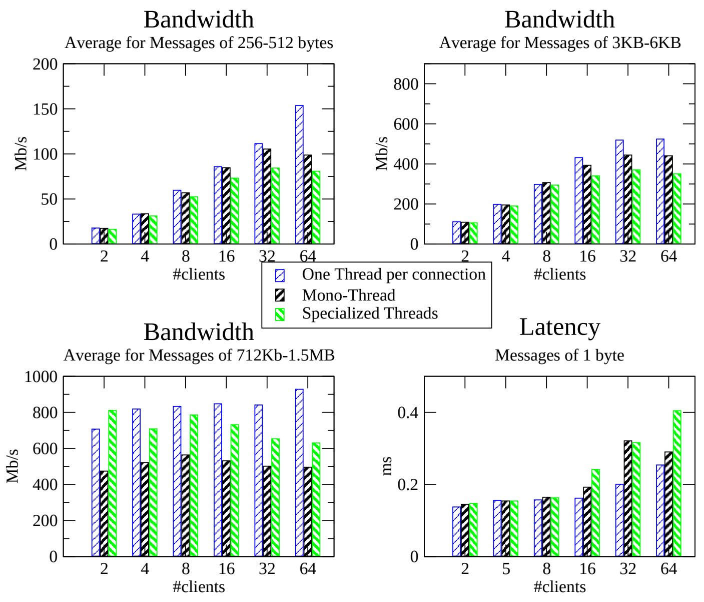


Fig. 8. Comparison of bandwidth and latency for all the implementations without logging

kernel has internal buffer of sufficient size to begin reception of messages before the relay application post the request. But for bigger messages, the application have to post the request to the kernel to allocate memory space before the reception can effectively begin. Multithreaded implementations have always a thread listening to socket and ready to receive any message.

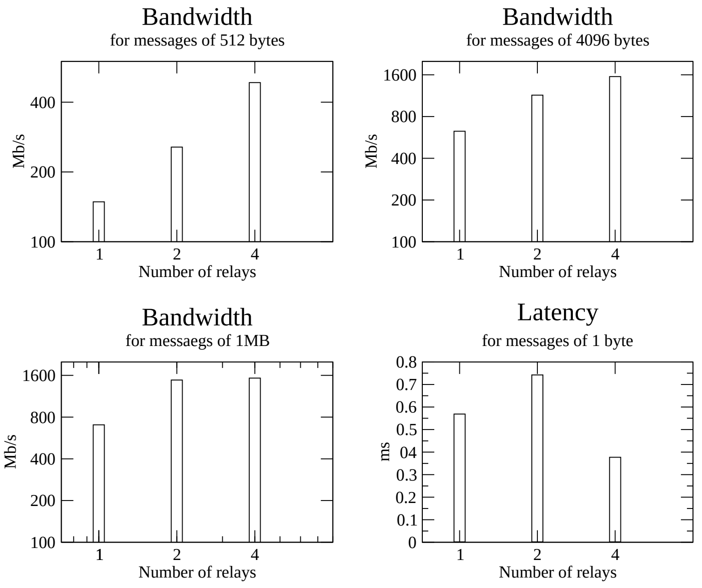


Fig. 9. Scalability of relays for NetPipe with 128 clients and 1 to 4 relays in a cluster

Figure 9 presents the scalability of relays inside a cluster. We ran NetPIPE-4 with 128 nodes (64 pairs), connected using one, two or four relays with the 1 thread per connection implementation. The figure demonstrate that multiple relays may be used to implement load balancing, up to network saturation. Indeed, bandwidth doubles each time the number of relays double, up to the network saturation. As expected, we can obtain performances over the NIC saturation only when there are more than one relay. NetPIPE computes the overall bandwidth as the multiplication of the smallest bandwidth pair by the number of pairs, thus since nodes communicate by pairs only, the observed bandwidth saturation should be 64 times the NIC saturation. However, since all communication pass through a single, or two or four relays, saturation is reached much sooner.

The latency observed is more variable. It remains acceptable and close to the expected latency (twice the TCP latency since messages has to pass 2 times through NICs), but we cannot explain why it increases for 2 relays, then decreases for four. It should remain constant.

5.2 Overall performances of the relays

The second set of experiments aims at validating the approach for real-life experiments in a Grid. We used for this Grid5000, an experimental Computational Grid founded by the French ministry of research and dedicated to computer science and Grid experiments.

Grid5000 includes 8 clusters in France, whose size will go from 100 node to 1000. In its current state, we used 4 cluster located in Orsay, Rennes, Bordeaux and Sophia-Antipolis. All these clusters are composed of bi-opteron 2.0GHz with 2Gb RAM and IDE hard drives. Grid5000 provides the ability to configure every computer of each experiment, and we configured our experimental setup with homogeneous operating systems (Linux 2.6.8.11), and standard library.

Figures 10 and 11 presents the bandwidth and latency for communication between two pairs of nodes according their location. These measurement were done with the NetPIPE-4 tool with two nodes and the TCP communication medium.

bandwidth (Mb/s)	Bordeaux	Orsay	Rennes	Sophia
Sophia	68.054155	42.425423	40.150269	940.459517
Rennes	110.025962	95.376749	940.354765	
Orsay	108.072588	930.419220		
Bordeaux	940.180969			

Fig. 10. Bandwidth measurement of the medium of the Grid

latency (ms)	Bordeaux	Orsay	Rennes	Sophia
Sophia	5.14608	8.61752	9.06626	0.03695
Rennes	4.00316	4.70542	0.04223	
Orsay	4.10875	0.06037		
Bordeaux	0.04204			

Fig. 11. Latency measurement of the medium of the Grid

One can see in these figures that the available bandwidth between two clusters is much smaller (10 to 20 times smaller) than the available bandwidth between two nodes of the same cluster. Moreover, the latency of figure 11 is two order of magnitude slower between two clusters than between two nodes. Thus, we consider that this setup represents a typical computational grid as expected in this study.

Although Grid5000 uses a security and authentication model that isolate completely the nodes of each cluster from Internet, thus authorize direct IP communication between two pairs of nodes anywhere in the grid, we used ssh tunnels to establish communications between each cluster, as described in the architecture section. Indeed, this security model is not usual and there are more chances that a non-experimental grid consisting in the merge of clusters from different administrations will not provide this ability. Thus, the application would have to rely on tunnels established on-demand to authorize communications between two clusters.

So, for the validating experiment, we used the NAS parallel benchmark suite [20]. Note that since we do not address the issue of workflows or dataflows but the issue of running an MPI application over a computational grid, we did not use Grid NAS benchmark, but older parallel NAS benchmarks which were designed for homogeneous clusters.

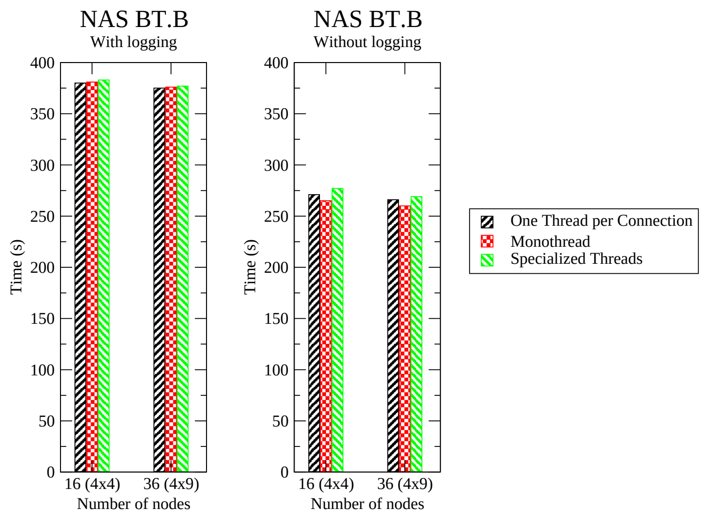


Fig. 12. Overall performances on the NAS BT.B benchmark, on a Grid deployment, with 4 clusters and 16 to 36 nodes, with and without logging

Figure 12 presents the completion time of the NAS BT benchmark, class B for two sizes (16 and 36 nodes), with and without logging for the three implementations. For each experiment, we used the four clusters, and distributed equitably the nodes on each cluster. Each cluster also holds a supplementary node, which runs the relay application for this cluster and the endpoints of three ssh tunnels to permit communication between nodes of this cluster and relays of the other clusters.

This figure demonstrate that the differences of bandwidth and latency observed between the three implementations are too small to be significant in this kind of grid. The monothreaded implementation seems to provide slightly better performance without logging compared to one thread per connection and the specialized thread implementations. However, this difference is not very significant and may be due to favorable network conditions at the time of the experiment.

Figure 12 also shows the impact of logging when dealing with a real-life application. Without logging, relays pass messages from one socket to another establishing a pipeline and the performance loss is mainly due to the major difference of bandwidth between intra-cluster and inter-cluster communications. With logging however, relays also have to store a copy of the messages. As long as the physical RAM can hold it, this does not impact the performances. But when the physical RAM is fully used (BT.B.16 uses up to 4Gb of virtual address space in each relay due to message logging), the system begins to swap pages at every message reception, and induces a significant slowdown of the distributed application. We tried to modify this behavior by implementing a ad-hoc logging mechanism which store in files the message payload. However, the caching policy of the Linux kernel introduces the same page faults and performance degradation. The current implementation tries to trigger disk synchronization when low network input/output is observed, but this has to be improved and the logging still introduces an important overhead in the application.

Last, this figure shows that with or without logging, the NAS BT benchmark loses its scalability when distributed over 4 clusters. The benchmark introduces big communications a small number of times, but these communication are mostly all-to-all. Considering the high heterogeneity of the bandwidth between clusters and inside clusters, this is easily explainable. Although running any kind of MPI application is desirable over a grid, the application must be fitted with the grid network heterogeneity to obtain the maximum performances. Activity traces of the experiment demonstrated that CPUs where used at 75% in BT.B.16, and at 32% in BT.B.36. This demonstrates that the communication cost is too high to provide scalability.

6 Conclusion

In this work, we present four methods for implementing message relay for MPI applications running on a computational grid. We propose a fault-tolerant implementation of the relay which provides remote pessimistic message logging. We evaluate three of the implementation methods on a single cluster for bandwidth and latency measurement, then on a French computational grid for real-life benchmark. Every evaluation was done with and without logging in order to estimate the overhead of the message logging.

Micro benchmark demonstrated that the implementation can have a significant impact on the performance of the relay. Surprisingly, for long messages, only the most thread-consuming implementation obtained the medium saturation. This is surprising since this approach is generally believed as quite inefficient. However, the study demonstrated that the generic approach of the monothreaded method may provide the same performances for long messages and also provide the best performances for short messages. This method is implemented and is being evaluated as we write this article.

The real-life benchmark demonstrated the feasibility of the relaying technique and its limitations. For maximum performance, the application should be aware of the different networks capacities and node placement, groups and other issues should be addressed automatically by the grid MPI library. Coupled with a checkpointing mechanism and a checkpoint scheduler to alleviate the memory consumption of the relays when logging is enabled, remote pessimistic message logging can be of low cost as it is proposed in this work.

The current implementation is a proof of concept for relays used as fault-tolerance components in the computation grid. For performance, direct communications must be re-enabled between the peers of a same cluster. This introduces two issues: 1) the matching of messages must be distributed between the preferred relay and the driver, which will have to multiplex its communication channels; 2) the event logging can no more happen at the relay-level only. This work, and in particular the section discussing the performances of relays with many channels to multiplex on a same cluster, gives clues on how the first point must be addressed. The second point relates to composition of fault-tolerance protocols. A first step will be to compose a message logging protocol at the cluster level with the pessimistic logging protocol used at the relays level.

Another issue not addressed in this work is the integration of this fault tolerance technique in the classical grid framework. The current implementation uses a script-based, ad-hoc, crude, resource booking mechanism and failure detection. Managing the nodes, deploying the relays, checkpoint servers, detecting the failures and booking new resources to cope with them should be automatic and fitted for any kind of grid. We hope to develop collaborations on this subject with the other CoreGrid members.

Last, to provide fault-tolerance, we still have to assume that the relays will not be subject to failure. This introduces multiple points of failures, one per cluster at least. This also means that the current implementation can not cope with mass failures like the loss of a complete cluster including its relay. This is a legitimate concern since it is likely that a whole cluster may fail simultaneously because of an Internet disconnection or a power failure of the cluster site. To address this issue, one has to introduce replication or distribution of logging. Doing this without introducing a high performance overload is a hard task.

References

1. W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.
2. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference.* The MIT Press, 1996.
3. A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V project: a multiprotocol automatic fault tolerant MPI," *International Journal of High Performance Computing Applications*, 2006, to appear.
4. A. Tanenbaum, *Modern Operating Systems, 2nd edition.* Prentice Hall, 2001.
5. R. Stevens, *UNIX Network Programming, 2nd Edition.* Prentice Hall, 1999.

6. T. Ylnen, "Ssh – secure login connections over the internet," in *Proceedings of the Sixth USENIX Security Symposium*, 1996.
7. "Openssl: The open source toolkit for ssl/tls," http://www.openssl.org.
8. M. Meissner, "Relayd: a relay daemon," http://www.opus1.com/www/telnet/tools/relayd.c.
9. I. Foster and N. T. Karonis, "A grid-enabled mpi: Message passing in heterogeneous distributed computing systems," in *proceedings of the ACM/IEEE SC 1998 Conference (SC'98)*, 1998, p. 46.
10. I. Foster, C. Kesselman, and S. Tuecke, "The nexus approach to integrating multithreading and communication," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 70–82, 1996.
11. I. Foster and C. Kesselman, "The globus toolkit," pp. 259–278, 1999.
12. N. Karonis, B. Toonen, and I. Foster, "Mpich-g2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 63, no. 5, pp. 551–563, May 2003.
13. N. Carriero and D. Gelernter, "Linda in context," in *Communication of the ACM*, vol. 32, no. 4, 1989, pp. 444–458.
14. W. Gropp and E. Lusk, "Fault tolerance in MPI programs," *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.
15. M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375 – 408, september 2002.
16. K. M. Chandy and L.Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75.
17. L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel, "An analysis of communication induced checkpointing," in *29th Symposium on Fault-Tolerant Computing (FTCS'99).* IEEE CS Press, june 1999.
18. L. Alvisi and K. Marzullo, "Message logging : Pessimistic, optimistic, and causal," in *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995).* IEEE CS Press, May-June 1995, pp. 229–236.
19. Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performace evaluator," 1996. [Online]. Available: citeseer.csail.mit.edu/snell96netpipe.html
20. D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Report NAS-95-020, 1995.