A Channel Memory based fault tolerance for MPI applications *

A.Selikhov[a], C.Germain[b]

[a]Supercomputer Software Dep., ICMMG SB RAS, pr.Lavrentieva, 6, Novosibirsk, 630090, Russia

[b] Université Paris-Sud, LAL (CNRS), Bâtiment 200, F-91898, and LRI (PCRI), Bâtiment 490, F-91405, Orsay Cedex, France

Fault tolerant message passing environments protect parallel applications against node failures. Very large scale computing systems, ranging from large clusters to worldwide Global Computing systems, require a high level of fault tolerance in order to efficiently run parallel applications. The Channel Memory approach provides the infrastructure for scalable tolerance to simultaneous faults. Along with a specially designed checkpointing system and recovery protocol, this approach has resulted in the MPICH-V architecture. In this paper, we describe CMDE – a stand-alone distributed program system based on MPICH-V architecture and implementing an approach to tolerate faults of Channel Memories.

Keywords: Channel Memory, Message Passing Interface, Fault Tolerance, Global Computing, Grid

1. INTRODUCTION

Parallel execution models have been designed with a traditional machine model in mind, which can be summarized as strongly coupled: machines are reliable, and information flows reliably across computing entities (processes or threads). This assumption becomes less realistic with present computing infrastructures. The first case is very large clusters, which become increasingly represented in the most powerful installed machines (e.g., those from Top500 list). Even for high-end clusters, the Mean Time Between Failures (MTBF) is typically less than one day. The second case is cycle-stealing systems, either local [1], or worldwide as Global Computing and P2P systems, which gather the idle time of low-end desktops and have MTBF measured in few hours or less. While large clusters target parallel or distributed applications, cycle-stealing systems cannot target communication-bound applications by lack of low-latency communication infrastructure; however, the scope of these systems could extend to loosely coupled parallel applications. The challenge is then to design an execution environment that provides fault tolerance for a wide range of parallel applications.

Previous research in the MPICH-V project [2,3] has defined a protocol for transparent

*This work has been partially funded by the French ministry of research under the ACI GRID grant CGP2P.

fault tolerance for message-passing programs. The core of MPICH-V is a rollback recovery protocol which provides the semantics of a non-faulty execution while only the faulty processes need to be restarted. This feature ensures that the computation may progress even in presence of frequent faults. The protocol is based on uncoordinated checkpointing and distributed pessimistic message logging on dedicated architecture elements – the Channel Memory Servers.

This paper describes CMDE, a standalone parallel execution environment targeted at fault tolerance of MPI applications. The Environment is based on the MPICH-V architecture and manages a dynamic set of computing resources, which can appear and disappear unexpectedly, implementing this transparently for the user application code. CMDE tolerates faults of the nodes running the parallel application up to simultaneous faults of all nodes. It further enhances its flexibility and performance by tolerating faults of Channel Memories through the Limited Replication of Channel Memories algorithm (LRCM). Finally, the design allows for interoperability with other systems providing their own resource management, such as Condor [1] or XtremWeb [4].

2. ARCHITECTURE

CMDE consists of a number of servers and clients providing computing and management services, running on a number of interconnected hosts. The computing services are allowed to be volatile, that is to disappear without notice, while some management services are required to be reliable. These services implement a dedicated MPICH library and utilize the Condor Stand-Alone Checkpointing Library and the TCP/IP protocol. In the following, an MPI program is modelled as a set of abstract *MPI nodes*, which must be implemented on top of volatile physical machines though failure-prone *tasks*.

2.1. Overview of the Services

A CMDE system is composed of one *Dispatcher*, a pool of *Workers* and *Channel Memory Servers* (CM Servers) and some *Checkpoint Servers* (CP Servers) (Fig. 1).

The Dispatcher is the CMDE front-end. It is responsible for storing and queueing MPI applications, and for collecting the other components of the Environment. The Dispatcher monitors resource availability and schedules tasks to Workers, including reallocation of Workers for failed tasks.

The pool of services providing the execution of an MPI application contains Workers, which run the tasks, and the Channel Memory Servers, which decouple the tasks communications and buffer MPI messages. The Worker represents a computing resource in CMDE. Its launches an application task locally and supports task checkpointing facilities by running an additional checkpoint alarm process. A detailed description of the features and principles of the CM Server is presented in [5]. The main purpose of the CM Server is to handle the Channel Memories. The Channel Memories store the MPI messages being in transit between a source and a destination MPI node. Each CM Server is responsible for delivering the messages to a subset of the MPI nodes, which we call its *own* MPI nodes, on their request and for buffering them and storing between two consecutive checkpoints.

The main purpose of the CP Server is to store checkpoint files that record the images of the running tasks, and to send them back on request at the restart point after a fault.

After a task failure, the Dispatcher allocates a new Worker to the MPI node; the task is

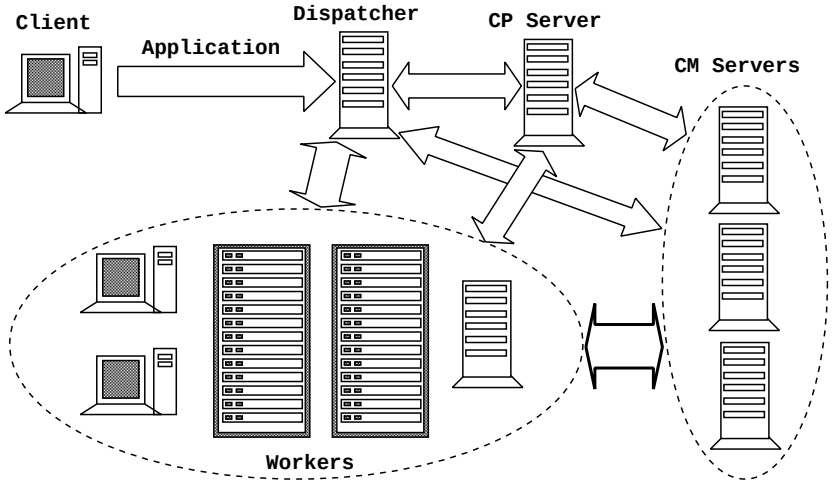

Figure 1. An illustration of the CMDE architecture. Normal arrows – application support communications, bold arrow – communications between tasks and CM Servers during execution of a parallel application

rollbacked to the last successful checkpoint and CM Servers manage its message exchanges since this checkpoint.

2.2. Building a CMDE system

Building a CMDE systems starts from launching the Dispatcher on a host, which should have enough disk space to store the application codes and input/output files, and a few ports opened for the connections of the other services. All the other services may connect to the Dispatcher in any order.

CMDE is able to accumulate resources of different types, *e.g.* nodes of clusters, stand-alone servers and workstations. The only limitation for the current implementation of CMDE is the requirement for the existence of a direct connections between the Workers and the other services. This limitation makes impossible to join two clusters with nodes having no direct connection to the outside world.

3. APPLICATION MANAGEMENT

The application management in CMDE includes linking an application code with the Channel Memory-enabled MPICH-CM library [5], submitting the application to CMDE, launching the application and providing its completion using the fault tolerance mechanisms. In this section, the first three stages will be described in more details, while the support for fault tolerance is presented in the next section.

3.1. The MPICH-CM library design

CMDE utilizes the MPICH [6] implementation of the MPI standard. MPICH-CM provides an MPICH implementation which decouples the communications through CM Servers. In MPICH, user-level MPI functions are implemented by a few functions of

a low-level communication *device*, e.g. *ch_p4*. We defined and implemented a *ch_cm* device, which targets the Channel Memories protocol, for reliable (non faulty) Channel Memories [5]. The *ch_cm* device implements three basic blocking communication functions on top of TCP/IP protocol. Each higher-level MPI message is sent to a destination CM Server together with an information message, containing, in particular, its real destination. Receiving an MPI message involves a request to the own CM Server and blocking receiving the required message. In addition, the functions call checkpoint creation mechanisms.

In order to take into account failures of CM Servers, a new version of MPICH-CM library device, *ch_cmde*, has been designed and partially implemented, providing task-side implementation of LRCM.

3.2. Submission of an application

To prepare a parallel application for submission to CMDE, one has only to link the application with the MPICH-CM library and the Condor checkpointing library. A special Client program manages the application submission process using the IP address of the Dispatcher and a special application configuration file.

When submitting an application to the Dispatcher, the Client checks the configuration file, uploads the application files to the Dispatcher and notifies about the result of the submission. The Client is considered as a textual interface to the Dispatcher for submission of applications and may be redesigned using other user interfaces, e.g. special graphic or web interfaces.

After successful submission of the application files, the Dispatcher registers the application in the local queue and checks if there are enough services to assign them to the application and to launch it. At least two CM Servers and one CP Server may be assigned, whereupon a free Worker, if any, is assigned to the application and the Dispatcher sends it a task for execution. Re-launching a failed task involves some additional information to be sent to the Worker to let the task code be started and connected to CM Servers.

4. FAULT TOLERANCE MECHANISMS

Providing fault tolerance for parallel applications in CMDE is based on uncoordinated (asynchronous) checkpointing of application tasks and on pessimistic logging of messages being transferred between the tasks. It consists of task-side support included in the MPICH library device and based on Condor Stand-Alone Checkpointing library (CSCL) [7], and of a special task checkpointing and recovery protocol [2], making possible to perform checkpointing and restart asynchronously. Making Channel Memories be also fault tolerant involves an additional algorithm, replicating the Channel Memory to another CM Server. Failures of CNDE components are considered as fatal: if a component reconnects later, it is considered as a newcomer. The set of fault recovery mechanisms determines different recovery possibilities for different components of CMDE.

4.1. Task-side fault tolerance support

Task-side fault tolerance support includes creation of checkpoint images of the task and an implementation of Channel Memory fault detection and handling. Checkpointing is implemented at the beginning of each of the three basic communication functions of the MPICH library device. When a checkpoint is scheduled, a new process is forked and

utilizes the CSCL API to store the process state in a file.

When the CM Servers themselves become an unreliable communication medium (see details in 4.3), new features are has to be added to the task-side fault tolerance support to ensure a consistent view of the messages between the Workers and the CM Servers. Each write issued by a Worker to a Channel Memory has to be acknowledged, to ensure message integrity. Each of three basic communication functions in *ch_cmde* begins with a request to the destination CM Server. If any of the CM Servers managing the application has failed, the job suspends its execution until the reconfiguration of the CM servers completes. After successful reconfiguration, the task reconnects to a new own CM Server if the old one has been disconnected and restarts the communication from the beginning. All this work is performed by the communication device of a task independently from other tasks and asynchronously.

4.2. Task checkpointing and recovery protocol

The MPICH-V task checkpointing and recovery protocol is presented in [3,2] and is shortly summarized here.

After successful launching of a task, the Worker starts a checkpointing alarm process, which sends checkpointing signals to the task process. The communication protocol between CMDE services in order to create a checkpoint for the task has the following steps. First, after a fixed checkpoint timeout, the checkpointing alarm process creates a new thread which establishes a connection to CP Server, sends the checkpointing signal to the application task and waits until the task has created its checkpoint image. The task sends a notification to its CM Server about starting creation of a checkpoint, forks a new process to make the checkpoint image and continues its execution. The new process writes the checkpoint image to the host hard disk and exits. The thread waiting for the file sends it to the CP Server and disconnects it. The CP Server stores the checkpoint image and notifies the Dispatcher and the CM Server about the success. This allows the CM Server to remove all channel memory messages being buffered before the checkpoint.

An essential property of the current checkpointing implementation is that a task process may detect the signal from the checkpointing alarm process only at the start of an MPI communication. Being simple to implement, this creates some limitations and is intended to be redesigned in the future.

4.3. Fault tolerance for Channel Memories

The possibility to tolerate faults of CM servers is based on the fact that the CM Servers have a very simple and predictable functionality, in contrast to user jobs. Restoring the service of a CM Server after a fault means only restoring all the Channel Memories (message logs and structurecontrol information) handled by this server. Besides, the MPICH-CM library monitors the faults and the reconfiguration process.

The LRCM algorithm [8] is aimed to manage both disappearing and appearing of CM Servers during execution of parallel applications. The general idea of LRCM algorithm is to replicate the Channel Memories handled by a CM Server onto *one mirroring* CM Server being chosen appropriately. One-by-one mirroring used in LRCM algorithm allows tolerate only one CM Server fault at a moment of time, until reconfiguration of CM Servers has been finished.

4.4. Fault recovery of the Environment

To summarize all fault tolerance mechanisms implemented in CMDE, one can consider recovery from faults of various services.

A fault of a Worker and, as a consequence, of a task is handled by allocation of a new free Worker for the task if such a Worker is available and do not stop the application. If there is no free Worker, all other tasks of the application are suspended at the point where they start waiting for the failed task messages.

A fault of a CM Server leads to suspending all tasks of all applications handled by this CM Server beginning from the next communication of each this task. In the case of two or more CM Servers registered in CMDE before the fault, tasks of all the applications continue their communications right after reconfiguration of the remaining CM Server(s). Otherwise, all the applications will be restarted from the beginning as soon as the first two CM Servers will be registered in CMDE.

The CP Server is expected to be reliable in the current CMDE architecture. Thus, its fault leads to the cancellation of all applications it handled. Allowing applications to continue without checkpointing will give them a chance to finish their work if no faults occur on Worker hosts. Otherwise, these applications will be restarted from the beginning using another CP Server if any or as soon as the first CP Server will be registered.

The Dispatcher is considered to be unique in the current CMDE architecture and to be reliable, therefore a fault of the Dispatcher leads to cancellation of all applications being queued and to the disconnection of all CMDE services. However, because the Dispatcher does not participate in the application communications, all running applications will finish if no faults occur in Workers and CM Servers. Afterwards, all the services have to reconnect to a new Dispatcher.

5. PERFORMANCE

Obviously, there is a price to pay for the advantage of fault-tolerant communications. Mediating the message-passing through the unreliable CM Servers brings an overhead, just as TCP/IP brings an overhead with respect to UDP by implementing reliable low-level communications. In the following, we discuss the performance of CMDE only in the sense of its ability to limit all the overheads induced in the course of a normal execution of an MPI applications, when no fault occurs. All other performance characteristics of CMDE like the application submission cost, or the cost for registration of new services, or the cost of fault recovery process itself are out of consideration here. However, from what has been sketched before, the fault tolerance overheads are limited to the exchange of some control messages, and do not include any massive data transfer, and are rather negligible if the faults are not extremely frequent.

As usual in performance evaluation for communication software, the methodology is to compare the performance of some test applications run on top of CMDE with the same applications run on top of a reference communication software. The reference in the following is MPICH-1.2.4 over TCP/IP (with the *ch_p4* communication device).

The main contribution to the execution overhead is, of course, the utilization of the Channel Memories. Detailed graphs are presented in [3,5]. The overall result is that the time of CMDE blocking communications is at most twice the *ch_p4* implementation. This

is explained by the fact of two actual communications through Channel Memory based communication *ch_cm* devices instead of one through *ch_p4* device.

The implementation of the LRCM algorithm includes two more phases for each task communication: probing the availability of a Channel Memory and receiving the acknowledgement about the communication. The first phase involves request-response messages passing, the last one is implemented as receiving a short message. All these additional transactions use fixed-length messages and therefore bring a constant latency overhead.

The last significant performance factor is the implementation of the checkpointing mechanism. The checkpointing process does not block the task, but consumes a fraction of the processor time and some network bandwidth. Thus, the computation is slowed down, and the messages may be delayed.

The testbed to obtain preliminary performance test results is a cluster of dual Pentium III nodes running Linux. The nodes communicate through one switch and 100 Mbits network. According to the performance results for a round-trip test with the *ch_cm* device, the overhead is very low at message sizes less than 220 kB and becomes nearly 75% of *ch_p4* performance for larger messages. The throughput of the Channel Memory based communications is 1.66 MB/sec on 1-kB messages (1.8 MB/sec for *ch_p4*), 5.4 MB/sec for 100-kB messages (5.45 Mb/sec for *ch_p4*) and 4.3 Mb/sec for 500-kB – 1-Mb messages (5.5 Mb/sec for *ch_p4*).

The current implementation is characterized by the latency (time for a an empty message) around 1 ms for the communication device supporting LRCM (*ch_cmde*), 0.4 - 0.6 ms – without LRCM support (*ch_cm*), in comparison with 0.16 ms for MPICH with *ch_p4*.

6. RELATED WORK

Rollback-recovery protocols in message-passing systems have undergone extensive study (for an in-depth analysis, see [9]. In the last few years, a lot of implementations of these protocols have been explored. Systems based on the Chandy-Lamport algorithm [10] are exemplified in Cocheck [11] and Starfish [12]. Such systems rollback the execution to a coherent snapshot, requiring the re-execution of all processes, while systems based on message logging can rollback a limited number of processes. The MPICH-V protocol rollbacks only the failed ones. A completely different strategy is to provide the user an API and a toolbox to monitor the faults and recover from them, *e.g.* Clip[13] or FT-MPI[14]. Recent work [15,16] has focused on protocols that log the messages unreliably, typically on the sender (Worker in our context), with various strategies to keep information about the execution dependency graph. This approach can improve performance over the Channel Memory one for long messages, due to lower use of the overall network bandwidth, when memory usage is not an issue. However, in real applications, logging messages consumes a lot of memory; thus, logging messages on Workers can compete with the task execution for memory space, and make the behavior of the MPI task less predictable for user codes that optimize the accesses to the memory hierarchy. As some dependencies must be respected in the unreliable log, logging messages on reliable services improves the latency of short messages, and allows for a clear view of memory requirements.

7. CONCLUSION AND FUTURE WORK

This paper has presented an environment providing transparent fault-tolerance for MPI applications, based on the MPICH-V protocol. The main result is that n–fault tolerance can be implemented in a real, standard-based system, by the Channel Memory approach, with bounded performance degradation with respect to a non fault-tolerant system. Future work will address the performance tradeoffs: infrequent checkpoints free network bandwidth for productive message-passing, but require more memory space to log messages, thus more Channel Memories. Adaptive algorithms would balance the checkpoint scheduling with the memory requirements to optimize message latency or throughput.

REFERENCES

1. R. Raman and M. Livny, High throughput resource management. Chapter 13 in The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, California (1999).
2. T. Hérault and P. Lemarinier, A rollback-recovery protocol on peer to peer systems. Proc. of MOVEP'2002 Summer School (2002) 313–319.
3. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri and A. Selikhov, MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In Proc. IEEE/ACM SC2002 Conf., Baltimore, Maryland (2002) 1–18.
4. G. Fedak, C. Germain, V. Neri and F. Cappello, XtremWeb: a generic global computing platform. IEEE/ACM CCGRID'2001. IEEE Press (2001) 582–587.
5. A. Selikhov, G. Bosilca, C. Germain, G. Fedak and F. Cappello, MPICH-CM: A communication library design for a P2P MPI implementation. Proc. 9th European PVM/MPI User's Group Meeting, Linz, Austria, September/October 2002, LNCS, Vol. 2474. Springer-Verlag, Berlin Heidelberg (2002) 323–330.
6. W. Gropp, E. Lusk, N. Doss and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, 22:6 (1996) 789–828.
7. J. Pruyne and M. Livny, Managing checkpoints for parallel programs. Workshop on Job Scheduling Strategies for Parallel Processing IPPS '96.
8. A. Selikhov and C. Germain, CMDE: a Channel Memory based dynamic environment for fault-tolerant message passing based on MPICH-V architecture, Proc. 7th Int. Conf on Parallel Computing Technologies (PaCT-2003), LNCS, Vol. 2763, Springer-Verlag, Berlin Heidelberg (2003) 528–537.
9. E.N. Elnohazy, L. Alvisi, D.B. Johnson and Y.M Yang, A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys, 34:3 (2002) 375–408.
10. K.M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems. ACM Trans. on Comp. Systems, 3(1) (1985) 63–75.
11. G. Stellner, CoCheck: Checkpointing and proces migration for MPI. Proc. 10th International Parallel Processing Symposium (IPPS'96), Hawaii (1996) 526–531.
12. A. Agbaria and R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. Proc. 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99) (1999) 167–176.

13. Y. Chen, J. S. Plank and K. Li, CLIP: A checkpointing tool for message-passing parallel programs. Int. Conf. on High Performance Networking and Computing (SC'97) ACM Press (1997).
14. G. Fagg and J. Dongarra, FT-MPI: fault-tolerant MPI, supporting dynamic applications in a dynamic world. Proc. 7-th EuroPVM/MPI User's Group Meeting, LNCS, Vol. 1908 Springer-Verlag, Berlin Heidelberg (2000) 346–353.
15. R. Batchu, J.P. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass and M. Apte. MPI/FT: architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. DSM 2001. (2001).
16. A. Bouteiller, F. Cappello, T. Hrault, G. Krawezik, P. Lemarinier, F. Magniette. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. IEEE/ACM SC 2003, (2003)