A Rollback-Recovery Protocol on Peer to Peer Systems

Thomas Hérault and Pierre Lemarinier

Université Paris sud Orsay, Laboratoire de Recherche en Informatique
Thomas Hérault, PhD, herault@lri.fr
Pierre Lemarinier, DEA trainee, lemarini@lri.fr

SUPERVISORS Joffroy Beauquier and Brigitte Rozoy, LRI
KEYWORDS: System modelizing, Pessimistic log protocol, fault tolerance

Abstract. We are intererested in the fault tolerance aspect of a peer to peer computation protocol. We first define a model for a peer to peer system. We then establish a protocol to provide fault tolerance. This protocol is based on pessimistic log protocol. Its main principles relies on using the communication servers, that are necesssary due to firewall problem, as a stable storage for logging messages. Finally we introduce the key points of a theoretical proof of the fault tolerance property of the protocol.

1 Research Area - Main Themes

1.1 X-Trem Web project : from global computing to peer to peer

X-trem Web is a project developed in the LRI through five different research groups. It provides a protocol to compute over Internet. Its basic principles relies on *peer to peer* systems. A member can ask for a computation on X-trem Web system. In exchange he has to offer his waste cycles time of computer's inactivity to X-Trem Web. A member is called "worker" for the rest of the document.

This paper is about our part on this project, the fault-tolerance aspect. We had to model the X-Trem Web system and find a fault-tolerant protocol.

1.2 A need for a fault tolerance system

First of all we consider crash of workers as the only type of fault in the system. The main idea of X-Trem Web project relies on cycle stealing. People lets their computer make calculation for X-Trem Web while they haven't need for them. When someone want to re-use his computer he should be able to get it instantaneously. SETI@home[Ber02] is an example of system relying on cycle stealing. Such system are called massively parallel due to the great number of computer working on one computation.

All the resources used by X-Trem Web have to be freed. This makes a great volatility of resources. Moreover, the only solution to free instantaneously resources is to make the application exit. That's why we can consider the user stop of the computation as a crash.

1.3 Related works

A distributed system is composed of processes and communication channels. The only interaction between processes is messages passing. An event occurs on a process when it computes, sends or receives a message. The event occurrence makes the process change its last state.

The fault tolerance is often achieved with the back-up of some elements during execution. Then when a fault occurs, the faulty processes are rollbacked or re-execute with the knowledge of the saved elements.

There is two main ways of working in the domain of re-execution, the checkpoint based technique and the log based technique.

Checkpoint based Checkpoints protocol consists in defining a restart point on each process and saving their context ("checkpointing" their state). If a failure occurs, all processes are rollbacked to their last checkpoint. In order to preserve global coherence of the system there is a need of synchronization of all checkpoints, with by example global snapshot.

An example of checkpoint based protocol is CoCheck[Ste96]. It's an environment that synchronize checkpoint by Chandy and Lamport snapshot[CL85]. It uses a specific message in the application communication to inform all processes of a snapshot need.

In our case, restarting all processes is not an acceptable solution as it get hundreds of computer to restart their calculation, hundreds more communication on a low network as Internet. In fact, faults are not rare enough to let them affect the entire system with a global effect.

Log based Log based protocols assumes the re-execution of crashed processes without the use of global checkpoint.

The main idea is to save the events of an execution on a stable storage. If a process crashes, it is rollbacked to its initial state. The stable storage give it the ordered events it have to replay to reach back its last state before crash.

Each local execution of process of the distributed system is seen as an alternative sequence of non-deterministic events and their resulting following deterministic events. If we only save the non-deterministic ones on the stable support, it is enough to get the entire local execution. That's the *piecewise deterministic* (PWD) assumption [SY85].

Checkpoint of the process can be added to restart from a later state than the initial one.

So in a distributed system, events logging provides a rollback recovery of the crashed processes to their state preceding the crash while others may continue their computation.

There exists three types of log based protocol :

1. the pessimistic one ensures the events are saved on a stable support before continuing the execution. This authorize a frequent fault tolerance (example of protocol : [LNLE00]),

2. the optimistic one saves the events on a volatile memory and continue the execution. Asynchronously it saves all preceding events in the stable storage (examples of protocols : [SY85,JZ87]),
3. the causal one increases the optimistic log by adding the causal graph of the event to the messages (example of protocol : [EZ92]).

1.4 Model and definitions

A process is seen as a communicating state machine. The system is described as a set of processes and their communication links. A configuration of the system is a vector of each process states and the in-transit messages, taken at a given time. A transition is a configuration operator that leads from a configuration to another configuration, through a single action of some process. The occurrence of the action is called an event.

A computation is an alternative sequence of configurations and transitions.

Two computations are considered to be equivalent if they both leads to the same configuration regardless of the order of transitions.

An execution of a distributed system is a partial order on a set of events.

A canonical computation is a fault-free computation.

We modelize a fault by a specific transition. This transition begins in the last configuration before a crash occurs and leads to a special configuration in which a process is considered to have crashed. That means that its state is some particular past state of its own initial execution.

2 Directions of the Works

As explained before,global checkpoint is not an acceptable solution since it gets all processes restarted. Moreover we can not presume whether a computer will be ready to compute again. All checkpoints of processes must be logged on distant servers if we want them to be re-execute on an other computer. This distant back-up cost several minutes of inactivity and a great network traffic.

We decided to use a pessimistic algorithm due to the considerable number of crash expected, and because all the messages have to be relayed. It assumes a stable storage to log events. The X-Trem Web system is basically composed of a task server and a group of communication servers. This group of servers is necessary to provide communication between workers due to Internet firewalls. The idea for our fault tolerance protocol is to use this necessary reliable media as a stable storage location for a pessimistic protocol. So the use of communication servers is one solution for two problems : firewalls and stable storages.

2.1 MPI Program and non-deterministic events

We have to determined what sort of event are non deterministic. A reception event is a non deterministic event because the computation can depend on the message's contents. There is other type of non deterministic event such as the

processor-time dependent event. However program that run on X-Trem Web are parallel computations. Most of their codes have only receptions as non deterministic events so we will consider receptions as the only non deterministic events.

The X-Trem Web programs use MPI or *Message Passing Interface*. It's a well-known library of communication functions. It's purpose is to provide a protocol of communication for parallel program. It's commonly used among parallel community. We will transform a MPI implementation on X-trem Web into a fault tolerant one.

2.2 Extending and modelizing X-Trem Web System

The X-Trem Web system is composed of a task server and a group of communication servers. These communication servers are used as a storage media of events.

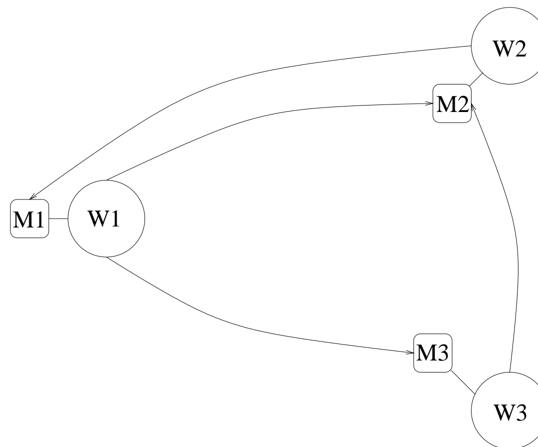

Fig. 1. three worker and their memories

To modelize the X-Trem Web system, we introduce a **process** type called *memory* that is assumed to be reliable. Here is the model of the system : each worker has a memory attached to it. A bidirectional link binds a process and its associated memory. An unidirectional link joins a sender process to the memory of the receiver process.

2.3 Protocol of memory process

A memory saves all messages sent to its process, and logs any of the non-deterministic event such as the checks of the presence of messages.

It uses a few list of messages : a queue for each sender and a ***given*** queue for the messages that have already been transmitted to the process.

To define the protocol, we have to define the behavior of the memories on communication. MPI is built upon a few basic communication functions :

1. **send :** to send a message to a specific process,
2. **receive :** to attempt to get a message from a particular process and wait until the message has been completely receive,
3. **receiveany :** as a receive but from any process, it just waits the first ready,
4. **probe :** to test if a message is ready to be receive.

The protocol define the behavior of the memories on these communications.

- First case : a non faulty execution :
  - **send :** When a process p_1 wants to send a message to process p_2, it sends the message to the memory attached to p_2. The memory saves it on the queue corresponding to messages from p_1 so as to be able to perform the transmission later,
  - **receive :** When p_2 asks for the next message from p_1, it sends a request to its memory. This one puts the next message in the ***given*** queue and sends it to p_2,
  - **receiveany :** If p_2 asks for the next message, it sends a request to its memory which chooses a message among all sender queues, put it in the ***given*** queue and finally sends it to p_2,
  - **probe :** When p_2 probes for a message, it asks its memory. This one tests whether a message is ready to be transmitted, saves the answer of this test in the ***given*** queue and sends this answer to p_2.
- Second case : a faulty execution : When a process crashes, the memory is assumed to be instantaneously informed of this situation. The process is rollbacked to its last local checkpoint.
  - **send:** When the process sends a message to a receiver memory, this later can test whether the message has already been sent. In this case the memory discards it, else it considers the process to be not on re-execution and acts as before.
  - **receive :** When the process probes or requests a message to its memory, this one uses his ***given*** queue to send back in the same order the initial sent messages. When all the ***given*** queue have been re-sent then the re-execution ends and normal execution continues.

3 Theorem and proof

We want to prove that the protocol we described ensure fault tolerance. We state that fault-tolerance is achieved by proving this next theorem :

Theorem *for any canonical configuration, if there is a finite number of fault-transitions, leading to a configuration I, the canonical execution and the execution starting from I are equivalent.*

Our protocol is based on the main principles that ① replaying the same events on the crashed processes is enough to reach back their last state before crash. ② If crashed processes have reached back their last state and have not interacted with non-faulty process then we are in a possible configuration of the computation.

First we prove the part ①. We prove that rollbacking the entire distributed system to the initial configuration and then replaying the same events in the same order is enough to reach an execution equivalent to the initial one. Particularly, a single process produces the same execution, just giving its initial events.

We prove the part ② with a projection of events. We consider the events of a re-execution from a configuration I after a fault transition. We pursue some kind of execution from I to a configuration in which all crashed processes have reached their last state before crash.

We project this execution on the set of events in which no replayed events appear. We pursue the canonical execution with this projected events. Finally we compare the configuration reached by the re-execution with the configuration reached by the canonical execution.

The proof is done by recursion on the number of faults: we exhibit for one set of simultaneous faults a computation which is equivalent to the canonical computation. We starts from the configuration obtained by applying the fault-transitions, then we prove (also by exhibiting equivalent computations) that an execution with $n+1$ set of simultaneous faults is equivalent to an execution with n set of simultaneous faults.

4 Conclusion, Results and Perspective

We have designed a model for peer to peer computation systems. A protocol have been designed to make the application fault tolerant based on the *PWD* assumption and the pessimistic protocol style. Moreover a proof has been made of the fault tolerance aspect of this protocol.

A local checkpoint system has been added to the message log protocol so that the process could be rollback to a later state. This has been realized by adding some checkpoint servers in the model to collects the checkpoints and few messages type to the protocol to warn concerned process of a checkpoint held.

Finally the complete system and the MPICH-V has been implemented and tested by the parallel architectures research groups.

References

[Ber02] Berkeley. Seti@home, 2002. http://setiathome.ssl.berkeley.edu/.

[CL85] K. M. Chandy and L.Lamport. Distributed snapshots : Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, volume 3(1), pages 63–75, february 1985.

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in manetho. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 18–27, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[JZ87] D B. Johnson and W Zwaenepoel. Sender-based machine logging. Technical report, Department of Computer Science, Rice University, Houston, Texas, 1987.

[LNLE00] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. In *Parallel Processing Letters, Vol. 10, No. 4, 371-382 , World Scientific Publishing Company.*, 2000.

[Ste96] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[SY85] E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, Aug 1985.