How to Design Reliable Servers using Fault Tolerant Micro-Kernel Mechanisms

Michel Banâtre[*] *Pack Heng*[◊] *Gilles Muller*[*] *Bruno Rochat*[◊]

[*] *IRISA/INRIA*

[◊] *BULL Research*

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

e-mail : ftm@irisa.fr

Abstract

The purpose of the Fault Tolerant Multiprocessor (FTM) project is to design a fault tolerant machine based on Stable Transactional Memories (STM). The FTM operating system is built from a MACH/OSF kernel which is extended to provide reliable services.

The purpose of this paper is to describe the basic mechanisms that we have added to the MACH micro-kernel in order to achieve fault tolerance.

1 Introduction

The Fault Tolerant Multiprocessor FTM [3] is a general purpose fault tolerant machine based on the association of Stable Transactional Memory (STM) boards with standard open multiprocessor machines. The STM is a fast stable storage device providing atomic memory accesses with low time overhead compared to normal RAMs. The FTM architecture can tolerate any *single hardware fault.*

Unlike STRATUS [8] and TANDEM S2 [9] architectures which mask hardware faults using static redundancy, the FTM architecture is based on dynamic redundancy, e.g. all the processors are performing different jobs in normal operating mode. In the event of a failure, the task of the faulty processor is handled by a backup one in addition to its own jobs. Thus the operating system has to deal with hardware faults and mask them from the users.

Our goal in the FTM operating system is to help the design of *reliable service*s which mask hardware faults from its clients. A reliable service is composed of one or more reliable servers; each reliable server is implemented by storing its variables in STM and by other fault tolerant mechanisms that we propose to add to the MACH 3.0 micro-kernel [1].

The following is structured as follows. In section 2, we present the FTM architecture along with the STM functionalities. In section 3, we describe the basic mechanisms that we have added to the MACH kernel to support reliable servers. In section 4, we give an example of a reliable server running on the extended micro-kernel. We conclude in section 5.

2 The FTM Architecture

The FTM architecture can be seen as a virtual loosely-coupled multiprocessor, in which the processing element is the *stable node*. The FTM architecture ensures the following three properties:

This research was supported in part by the DRET under the grant n° 90346.

1. The architecture tolerates any single hardware fault.
2. A stable node can restore a safe state of data after an internal failure and is able to resume computations.
3. The interconnection medium ensures that communication is always possible between two stable nodes and that no partition even occurs.

2.1 The Stable Node

A stable node S_{ab} (see Figure 1) is built from a primary processor P_a, a backup processor P_b and a Stable Transactional Memory STM_{ab}. The processor P_a can manipulate locally the stable variables in STM_{ab} with a fast access time. In normal operating mode, the STM_{ab} is not shared between P_a and P_b.

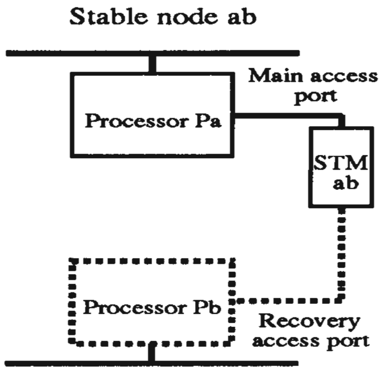


Figure 1. The stable node

The STM_{ab} is able to detect P_a processor failures using a watch-dog mechanism. When the STM_{ab} detects the failure of P_a, it restores a consistent state of the stable data and warns P_b. P_b then resumes aborted computations from the stable data stored in STM_{ab}. After a manual maintenance on the faulty processor P_a, it is restarted in the stable node S_{ab}. The STM_{ab} disconnects the backup processor P_b and reconnects the primary processor P_a. The computations resume on P_a. If both processors P_a and P_b fail, the computations running on the stable node S_{ab} are stopped but can be restarted as soon as one the processor P_a or P_b restarts.

Stable nodes are physically paired in such a way that the primary processor of one stable node is the backup processor of the other stable node (see Figure 2). Thus, there is no backup processor dedicated to fault tolerance in the FTM architecture (dynamic redundancy).

The physical FTM architecture is based on open multiprocessor machines. Its complete and detailed description, particularly of the interconnection medium, can be found in [3].

2.2 The Stable Transactional Memory

The STM provides two notions: the *stable object* and the *transaction*. A stable object is a contiguous set of memory words and a transaction is an atomic set of basic operations performed on the stable objects. All STM operations *can only be performed* within transactions. Objects of any size from kernel lists of few words elements to virtual memory pages of several Kilobytes can be managed atomically.

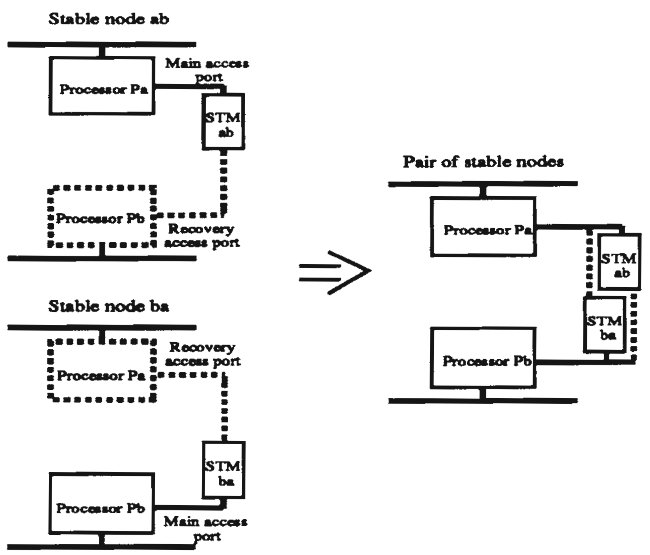


Figure 2. The pair of stable nodes

The STM is internally implemented using two banks of 32 Megabytes of battery-backup RAM memory and an intelligent controller which implements transactions using a two phase commit protocol.

2.2.1 Stable Objects and Protection

For performance purposes, the STM is directly accessed in the physical address space of a processor as a local memory. Generally, processors are not *fail-stop*: a faulty processor may thus corrupt accessible memory. To protect the STM, stable objects have to be made visible using an explicit *open* operation, before being modified by the processor. Similarly, after modification the object have to be closed using a *close* operation. Any access to a non opened or non existent object is detected and rejected by the STM.

The processing of an STM access violation by the micro-kernel depends on the level of the faulty software. If the fault comes from the kernel or a well tested server, we assume that there is some hardware malfunction and the processor is halted. If the fault comes from an application software or a server currently being tested, the faulty program is aborted.

2.2.2 The Transaction Facility

The transaction is the most important functionality of the STM. It allows the programming of complex atomic functions. The operations on transactions are *Begin, Commit, Abort.* When a transaction is committed (resp. aborted), all effects on stable objects are validated (resp. invalidated).

Several transactions can be used concurrently at any time. Thus a transaction is identified by a descriptor stored itself in the STM. The creation of a new transaction descriptor is performed atomically within another transaction and is distinct from its activation (*Begin*). When the STM detects a processor failure, it automatically aborts active transactions.

3 Basic Fault-Tolerant Mechanisms of the Micro-Kernel

3.1 Overview of the MACH Kernel

Recent evolution in the design of operating systems has advanced the notion of *micro-kernel.* A micro-kernel offers a minimal set of concepts allowing the implementation of complex distributed systems and applications in a modular way. MACH [1] and CHORUS [12] are examples of well known micro-kernels.

There are five basic entities in MACH: task, thread, port, message and memory object. A task is an allocation context of resources: it possesses a logical address space and access rights to ports or memory objects.

A thread is the execution unit and is attached to some task which defines its execution environment. Several threads can be created within the same task and share the same resources. In particular, they share the same address space and can communicate directly by shared memory.

A port is the communication resource and allows different threads to exchange messages. Several tasks can possess the send right to a port, but only one has the receive right. Rights can be transmitted by messages but only under the control of the kernel. Ports also permit to identify any MACH object.

A memory object must be mapped into a task's address space before being accessed by a thread. If a memory access raises a page fault, the kernel sends a message to the memory object port notifying it of the fault. The page fault can be treated by an external server (pager) which owns the page and sends it back to the kernel.

The five MACH basic entities allow to implement servers. A server is a task which performs a cyclic job. It receives requests from clients, treats them and returns results. Requests can be served by one or several threads. Clients are identified by their sending port and the server by its receipt port.

3.2 The FTM Fault Tolerant Micro-Kernel

The FTM micro-kernel extends MACH by adding mechanisms which allow the implementation of fault-tolerance. Normal MACH entities, as tasks, ports, memory objects, can be corrupted by a processor failure. To restore a safe system state, the solution is to create stable equivalent MACH entities. The FTM micro-kernel provides the standard MACH entities and in addition three other ones: *stable task*, *stable port* and *stable memory object.* All these stable entities are created and modified within STM transactions. Transactions are executed by threads and ensure consistency of stable entities in case of a processor failure.

3.2.1 The Stable Task

A stable task defines an execution environment which is associated with a stable node. As with a normal task, a stable task contains access rights and a logical address space. Each time a processor switch occurs, the environment is restored on the active processor by the FTM micro-kernel. Then a thread is restarted from the entry point of the task.

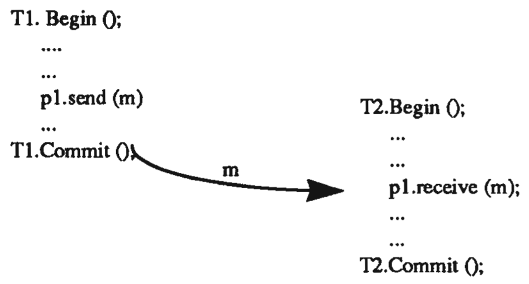
3.2.2 The Stable Memory Object

A stable memory object can be viewed as a logical STM. It has the same functionalities as the physical STM described in 2.2. A stable memory object is mapped into a stable task and is private to this task. Thus, threads can activate transactions, create and modify stable objects.

The FTM micro-kernel manages the sharing of the physical STM between the different stable memory objects. In particular, it performs STM page allocation and deallocation.

3.2.3 The Stable Port

A stable port is used to perform reliable communications between two threads. Like any of the operation defined on stable entities, operations on stable ports are performed within transactions. That means that sending or receiving messages is effective only at the transaction commitment.



```
T1. Begin ();
    ....
    ...
    p1.send (m)
    ...
T1.Commit ()
                                T2.Begin ();
                                    ...
                                    ...
                                    p1.receive (m);
                                    ...
                                    ...
                                T2.Commit ();
```

Figure 3. Message exchange between two tasks

For instance, in Figure 3, the message *m* is only sent when *T1* commits. Similarly, if *T2* aborts, the message is put back in the receipt queue of *p1*.

Stable ports are implemented in the micro-kernel, using lists of messages which are stored in STM. A reliable network message server is used when threads are located on different stable nodes. One of its functions is to locate the active processor of the destination stable node.

3.3 The FTM Reliable Server

A reliable server masks hardware faults from its clients and is implemented using a stable task. In the event of a processor failure, the stable task is restored on the active processor of the stable node and a thread is restarted on the entry point. Each server operation is executed atomically within an STM transaction. If the server fails, the message is put back in the port and the modified stable objects are restored to their original state. Thus, the atomicity property in the treatment of each remote procedure call ensures that the server does not lose any client message and executes *exactly once* each call.

As a normal RAM memory, the STM is shared between reliable servers. Its 32 Megabyte capacity satisfies all system needs. User applications, which are designed for the FTM, may also have access to STM. However, to support non fault tolerant applications, we have designed a reliable distributed virtual memory server [2] which allows to run transparently users programs. This server integrates in a uniform view all physical memories of the system (RAM, STM and disks).

4 Programming Example

4.1 Programming with the STM

A C++ interface of the STM has been designed to help the programming of applications. Two classes, *Transaction* and *Stable* are defined in this interface and mask the STM hardware to the C++ programmer. Moreover, the interface allows the definition of a C++ stable object simply by inheritance of the *Stable* class [11]. A similar inheritance mechanism is also proposed in Arjuna [6] and Avalon [5].

The C++ STM interface is defined as follows:

```
class Stable {
public:
   operator new (long s);                /* Creation of new stable object of size s */
   operator delete (void *pt);           /* Destruction of the object pointed by pt */
   void Open ();                         /* Open the stable object */
   void Close ();                        /* Close the stable object */
};
class Transaction {
public:
   operator new (long s);                /* Creation of a new transaction descriptor */
   operator delete (void *pt);           /* Destruction of the transaction descriptor */
   void Begin ();                        /* Activation of the transaction */
   void Commit ();                       /* Commit the transaction */
   void Abort ();                        /* Abort the transaction */
};
```

To illustrate programming using the STM, we treat the example of stable segment management in a virtual memory. We first define a *Segment* class independently of the STM mechanisms. To allow all stable segments to be allocated in the STM, the *Segment* class inherits from the *Stable* class. We present in the following example a simple segment class with two operations which read and write a page (the page has a fixed size of 1 Kilo bytes).

```
const int page_size = 1024;
typedef char[page_size] Page;                          /* page type */
class Segment : public Stable {
   const int segment_size = 10*page_size;              /* 10 K bytes */
   char S[segment_size];                 /* the segment is implemented by an array of characters */
public:
   void read_page (int no_page, Page& page);           /* read a page */
   {  int offset = (no_page*page_size)%segment_size;   /* offset in the segment */
      for (i=0; i<page_size; i++) page[i] = S[i+offset];
   }
   void write_page(int no_page, Page page)             /* write a page */
   {  int offset = (no_page*page_size)%segment_size;   /* offset in the segment */
      for (i=0; i<page_size; i++) S[i+offset]=page[i];
   }
};
```

As shown on the previous example, the programmer of a stable class does not need to know the STM mechanisms. On the contrary, the user of a stable segment has to manage explicitly transaction and object visibility. This is shown in the following example:

```
Transaction T;                           /* transaction which manipulates a stable segment */
Segment *s;                              /* pointer to a stable segment */
Page page;                               /* a volatile page */
```

```
/* atomic creation of the stable segment */
T.Begin (); s = new Segment (); T.Commit ();
T.Begin ();                                /* atomic move of page 0 to page 1*/
   s->Open ();
      s->read_page (0, page);
      s->write_page (1, page);
   s->Close ();
T.Commit ();
```

4.2 Example of a Reliable Server

We are now extend the previous example by designing a reliable server which offers in its interface, the *read*_page and *write*_page operations on a segment. The clients call these operations by remote procedure call.

A request message sent by a client to the server contains:

- the receiver stable port to which the message is sent,
- the sender stable port from which a result is eventually waited for,
- the type of the request (READ_PAGE, WRITE_PAGE, RESULT),
- the page number of the page which is read or written.

The client which calls a remote operation atomically sends a message to the server stable port and atomically waits for a result. The following example describes the remote procedure call to *write_page (0, page)*:

```
typedef unsigned int StablePort;           /*stable port identifier */
enum Request_t {READ_PAGE, WRITE_PAGE, RESULT};
struct {
   StablePort receiver_port;
   StablePort sender_port;
   Resquest_t request;
   int page_number;
   Page page;
} Message;

Transaction TClient;                       /* transaction of the client */
StablePort client_port;                    /* stable port of the client */
Page page;                                 /* the page to write */
TClient.Begin ();                          /* Atomic send of the write_page request */
 server_port.send (
   Message client_message=(server_port, client_port, WRITE_PAGE, 0, page));
TClient.Commit ();
TClient.Begin ();                          /* atomic wait for a result */
   client_port.receive (Message result_message);
TClient.Commit ();
```

As a message is only sent at the commitment of the transaction. It is necessary that *Tclient* commits between the send and receive operations. Otherwise the client would be indefinitely blocked on the receive operation.

In the treatment of a client request, the server atomically receives the message sent by the client, performs the operation on the stable segment and sends back the result.

```
Transaction TServer;                       /* transaction of the server*/
StablePort server_port;                    /* stable port of the server */
Segment s;                                 /* stable segment */
```

```
main()
{ /* infinite loop */
 while (TRUE) {
   Message client_message;               /* message received from a client */
   Page result_page = "";                /* result page */
   /* Atomic transaction : receive a message, treats an operation, sends a result */
   TServer.Begin ();
   server_port.receive (client_message);
   switch (client_message.request) {
   case WRITE_PAGE:
      s.Open ();
      s.write_page (client_message.page_number, client_message.page);
      s.Close ();
      break;
   case READ_PAGE:
      s.Open ();
      s.read_page (client_message.page_number, result_page);
      s.Close ();
      break;
   }
   client_message.sender_port.send (Message message_result =
      {client_message.sender_port, server_port, RESULT,
      client_message.page_number, result_page});
   TServer.Commit();
 }
}
```

5 Discussion

Using atomic actions to build robust distributed programs has been extensively investigated in ARGUS [10] and CAMELOT [7]. FTM contributes to this work by providing an efficient implementation of atomic actions on small data objects (arrays, lists). Thus, transactions can be used within the design of kernel and system services. Generally the implementation of remote operations only offers the semantics *at most once*. Using stable ports, we can offer the *exactly once* semantics.

As mentioned in this paper, the failure of a reliable server is transparent to the clients. However, if the client fails, its effects on the server are not undone. The coordination between client and server needs other mechanisms as distributed atomic actions. We are now working on the implementation of atomic action which are dynamically determined using communication between clients and servers.

The STM boards are currently under development. Consequently we are prototyping the FTM micro-kernel using a SUN 3 version of MACH 3.0 and a C++ emulation of the STM. In parallel, we are also porting MACH 3.0 on to the Motorola 68030 based processors of the FTM.

References

[1] M. Accetta and R. Baron and W. Bolosky and D. Golub and R. Rashid. *A New Kernel Foundation for UNIX Development. USENIX 86*, July 1986.

[2] M. Banâtre, G. Muller, B. Rochat, P. Sanchez. A Reliable Distributed Virtual Memory on top of the Mach kernel. *OSF Micro Kernel Applications Workshop*, Grenoble, France, November 1990.

[3] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. Design Decisions for the FTM : A General Purpose Fault Tolerant Machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing*

Systems, pages 71-78, Montréal, Canada, June 1991.

[4] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[5] G.N. Dixon and S.K. Shrivastava. Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems. In *Proc. of the 6th Symposium on Reliability in Distributed Software and Database Systems*, pages 107–114, Williamsburg, March 1987.

[6] J. L. Eppinger and A. Z. Spector. A Camelot Perspective. *UNIX REVIEW*, 7(1):58, 1989.

[7] E. S. Harrison and E. Schmitt. The Structure of SYSTEM/88, a Fault-Tolerant Computer. *IBM Systems Journal*, 26(3):293–318, 1987.

[8] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, Montréal, Canada, June 1991.

[9] B. Liskov and R. Scheifler. Guardians and actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[10] G. Muller, B. Rochat, and P. Sanchez. A Stable Transactional Memory for Building Robust Object Oriented Programs. In *EuroMicro 91*, Viennes, Autriche, September 1991. to appear.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.