The Design and Implementation of Voltan Fault-Tolerant Nodes for Distributed Systems

N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan and S.K. Shrivastava

TECHNICAL REPORT SERIES

No. 454  October, 1993
THE TECHNICAL REPORT SERIES

No. 454 October, 1993

The Design and Implementation of Voltan Fault-Tolerant Nodes for Distributed Systems

N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan and S.K. Shrivastava

Abstract

A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance. The processors of a node execute agreement and order protocols to 'keep in step'. The design and the implementation of three processor failure masking and a two processor fail-silent nodes using transputers are described in detail.

(This paper is to appear in Transputer Communications)

© 1993 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Science, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

DESIGN AND IMPLEMENTATION OF VOLTAN FAULT-TOLERANT NODES FOR DISTRIBUTED SYSTEMS

The Design and Implementation of Voltan Fault-Tolerant Nodes for Distributed Systems
[By] N.A. Speirs [and others]

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 454)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
SPEIRS, Neil Alexander

Abstract

A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance. The processors of a node execute agreement and order protocols to 'keep in step'. The design and the implementation of three processor failure masking and a two processor fail-silent nodes using transputers are described in detail.

About the author

N.A. Speirs is a Lecturer in the Department of Computing Science at the University of Newcastle upon Tyne.

S. Tao is a Research Associate in the Department of Computing Science at the University of Newcastle upon Tyne.

F.V. Brasileiro is a PhD student in the Department of Computing Science at the University of Newcastle upon Tyne.

P.D. Ezhilchelvan is Lecturer in the Department of Computing Science at the University of Newcastle upon Tyne.

S.K. Shrivastava joined the Department of Computing Science in August 1975, where he is a Professor.

Suggested keywords

ATOMIC BROADCAST DISTRIBUTED PROCESSING FAULT TOLERANCE
ORDERING PROTOCOLS REPLICATED PROCESSING RELIABILITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 621.381958
U.D.C. 519.687 681.322.02
THE DESIGN AND IMPLEMENTATION OF VOLTAN FAULT-TOLERANT NODES FOR DISTRIBUTED SYSTEMS

N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan and S.K. Shrivastava
Department of Computing Science,
University of Newcastle,
Newcastle upon Tyne, NE1 7RU, UK.

ABSTRACT
A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance. The processors of a node execute agreement and order protocols to 'keep in step'. The design and the implementation of three processor failure masking and a two processor fail-silent nodes using transputers are described in detail.

KEYWORDS
Atomic broadcast, distributed processing, fault tolerance, ordering protocols, reliability, replicated processing.

(to appear in Transputer Communications)
1. INTRODUCTION

Replicated processing on distinct processors whereby outputs from faulty processors can be prevented from appearing at the application level (by employing means such as voting the outputs produced by the processors), provides a practical means of constructing systems capable of tolerating \textit{fail-uncontrolled} (Byzantine) processor failures. A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance in a manner suggested above. The basic idea is conceptually simple; a node is built out of several processors which execute special agreement and order protocols to carry out replicated processing of computations to achieve fault tolerance. Such fault-tolerant nodes constitute the building blocks for constructing highly reliable distributed computing platforms. In [1] the detailed architecture of a \textit{family} of such nodes (named VOLTAN) is described, where we also discuss how distributed systems can be composed out of VOLTAN nodes. In this paper we will therefore confine our attention to a single node, concentrating on its internal implementation details. One member of the VOLTAN family is a perfect (failure free) node; such a node can be approximated by an architecture supporting N Modular Redundant (NMR) processing. An NMR node is capable of masking \(\pi\) internal processor failures (where \(N = 2\pi + 1\) and \(\pi \geq 1\)) i.e. the failures are not apparent from outside the node. Another member is a node which either works correctly, or stops functioning (becomes "silent") as soon as an internal failure is detected. Such a node has been termed a \textit{fail-silent} node. In this paper we describe transputer based implementations of a triple modular redundant, TMR, node (with three internal transputers, \(N=3\) and \(\pi=1\)) and a fail-silent node comprising two internal transputers. We assume a purely software approach to the management of redundancy and, further, that the nodes are to be used in a distributed system where processors as well as nodes communicate only by message passing.

The implementation of VOLTAN nodes presented here suggests that it is possible to construct nodes capable of tolerating a wide class of failures (Byzantine in the limit) by utilising standard 'off the shelf' components (processors), without recourse to any specialised hardware. In this respect, we have followed the approach, pioneered by the designers of the SIFT system [2], of using software implemented agreement and order protocols for supporting replicated processing. Unlike SIFT, VOLTAN nodes are capable of supporting quite general purpose message passing application programs. The main advantages of our approach over the traditional 'tightly-synchronised' approach of employing specialised voter/comparator hardware and a fault-tolerant hardware clock for driving the processors of a node are that: (i) technology upgrades appear to be easy; since the principles behind the protocols do not change, the protocol
software can be ported relatively easily to any type of processor (including the ones expected to be available in future); (ii) we note that by employing different types of processors within a node, there is a possibility that a measure of tolerance against design faults in processors can be obtained, again without recourse to any specialised hardware assistance; and (iii) since replicated computations are loosely synchronised, a node is likely to be more robust against transient failures [3] (this is because transients are less likely to affect loosely synchronised computations on the processors in an identical fashion).

The paper describes the design and implementation of TMR and fail-silent nodes. We have carefully designed our software in a modular fashion that makes it relatively easy to modify or replace modules. Thus our fail-silent node implementation has been obtained from the TMR one essentially by replacing the clock synchronisation and message ordering algorithms by simpler ones required for the operation of fail-silent nodes. We are also investigating ways of improving the performance of our nodes by employing new types of message ordering algorithms. Implementations based on these algorithms are also easy to incorporate within our design. Towards the end of this paper we describe how we have replaced the basic message ordering algorithms described in this paper by different ones that have significantly improved the performance of our nodes.

2. VOLTAN ARCHITECTURE

2.1. System model and assumptions

We will assume that a failed processor (and therefore the processes running on that processor) can exhibit fail-uncontrolled behaviour. We assume that (non-replicated) distributed computations have been composed out of a number of processes that interact only via messages. As an example, the function of a typical 'server' process is to pick up an input message from one of its input ports, process it and, if necessary, output one or more messages on its output ports. We also assume that if a process with multiple input ports has input messages on those ports then any one of these messages is chosen non-deterministically for processing. Message selection is however assumed to be fair, that is, the process will eventually select a message present on a port. A primitive operation, receiveany(m) is assumed for receiving a message from any of the input ports:

```plaintext
process S: /* a typical server */
  cycle
    receiveany(m)
  process m
    send (result_msg)
  end
end S
```
The model presented here is based on the well known *state machine* model (where a state machine is a process) for which the precise requirements for supporting replicated processing are known [4]. Given such a model of computation, active replication of a process, such as S (with a replica, one each running on the underlying processors of a node) will require the following two conditions to be met:

*Agreement:* All the non-faulty replicas of a process receive identical input messages;

*Order:* all the non-faulty replicas process the messages in an identical order.

So, if all the non-faulty replicas of a process of a node have identical initial states, and the computations performed on selected messages are *deterministic*, then identical output messages will be produced by them. This is the underlying principle of active replication.

Practical distributed programs often require functionality such as use of time-outs when waiting for messages. Time-outs (and other asynchronous events), high priority messages etc. are potential sources of non-determinism, making such programs difficult to replicate. In previous papers [1, 5] we have described how VOLTAN provides the necessary functionality for dealing with such cases. In this paper, we will assume the simple model discussed above.

It will be assumed throughout that the originator of a message can be *authenticated* by a non-faulty receiver. Digital signatures [6] implement authentication (with high probability). We therefore assume that each processor has a mechanism to generate a unique unforgeable signature for a given message and further that each processor has an authentication function for verifying the authenticity of a message signature. Thus if a non-faulty processor sends a message with its signature to some other non-faulty processor, any corruption of this message during the transmission can be detected by the receiver by authenticating the signature associated with the message.

All the nodes in the system will be assumed to possess unique identifiers (numbers). Similarly, each and every group of triplicated (duplicated) computational processes of a node will also be assumed to possess unique group identifiers (numbers). Each copy of a computational process will be assumed to maintain a local counter variable (initialised to zero) which is used for the generation of sequence numbers for the messages produced by that process. The sequence number for a message is produced as follows: whenever a process produces a new message, it assigns it a sequence number composed by concatenating the host node number, group identifier of the process and the counter value; the counter is then incremented by one. Correctly functioning replicas of a process will produce messages with identical sequence numbers, so voters (comparators) can use sequence numbers for selecting messages for matching. Sequence numbers are also used for duplicate message detection and removal throughout the system.
2.2. TMR Nodes

The failure-masking, TMR node to be described here has the following properties: (i) it functions correctly as long as no more than a single processor within a node fails; and, (ii) any spurious messages emitted by the failed processor of a correctly functioning node can be detected and rejected by all the correctly functioning receiver nodes.

As stated earlier, it is necessary that the replicas of computational processes on non-faulty processors within a node select identical messages for processing, to ensure that they produce identical outputs. This can be done by presenting a single input message queue, referred to as a delivered valid message queue, DMQ, to a process and ensuring that a process picks up the message at the head of its DMQ for processing. An atomic broadcast protocol, capable of tolerating Byzantine failures (the authenticated message algorithm presented in [7]) meeting both the agreement and order property is then employed to ensure that identical messages are enqueued in an identical order at all the non-faulty replicas of a node. The broadcast mechanism itself requires that the clocks of all the non-faulty processors of a node be synchronised such that the measurable difference between readings of clocks at any instant is bounded by a known constant.

Algorithms for achieving this abstraction exist which require all the non-faulty processors of a node to exchange authenticated messages amongst themselves (see for example [8]). The ability of a VOLTAN node to tolerate Byzantine failures arises because both the clock synchronisation and atomic broadcast protocols used are capable of tolerating Byzantine failures.

Each non-faulty processor of a node runs the following five 'system' processes:

(i) Diffuser Process: this process takes the messages produced by the computational processes of that processor, signs them and sends them to all the other processors of the node for voting.

(ii) Receiver Process: this process accepts only authentic messages from the network for processing. Messages with single signatures (these are from local processors of a node intended for voting) are sent to the local voter process; messages with two distinct signatures (these are from other nodes within the system) are sent to the local order process for distribution to the local destination processes.

(iii) Voter Process: the voter processes the messages coming from the receiver as follows. If the contents of such a message, say m, are identical to its locally produced counterpart, then m is countersigned (by considering the existing signature on m as part of the message). This doubly signed message is regarded as a valid (voted) message and it is sent to the transmitter process for transmission over the network to its destination. Messages that cannot be matched at a non-faulty voter are never countersigned and sent out. It follows that all correct (valid) messages issuing from a node will be double signed.
(iv) **Transmitter Process**: this process transmits valid messages coming from the voter process to their destinations.

(v) **Order Process**: this process orders messages by atomically broadcasting valid messages coming from the local receiver to all the order processes of that node (including itself). This permits order processes to construct identical queues of valid messages (DMQs) or processing.

### 2.3. Fail-silent nodes

A VOLTAN two processor fail-silent node implements the abstraction of fail-silent behaviour in the following sense: it produces either correct messages which can be verified as such by the receivers, or it ceases to produce new correct messages. This behaviour is guaranteed so long as no more than a single processor in the node fails. Any spurious messages produced by the failed processor of a node can be detected as such by all correctly functioning receiver nodes.

The TMR node architecture discussed previously can be modified easily to construct a two processor fail-silent node. The voter process needs to be replaced by a **comparator process** with the following functionality: a message that cannot be compared because its counterpart does not arrive within a time-out period or a comparison which detects a disagreement indicates a failure; once a failure is indicated, the comparator process stops, which results in no further double signed message being produced from that node. The message ordering and clock synchronisation protocols can be made particularly simple, since they are required to work only in the absence of any failures. The order protocol works as follows. The order process of a processor stamps a (valid) message to be ordered with its local clock reading; a copy of the timestamped message is signed and sent to the order process of the other processor in the node. If $T$ is the timestamp of the message received from or sent to the order process of the other processor, then the message becomes stable at local clock time $T + d + e$, where $d$ is the maximum transmission time taken to travel from one order process to another order process, and $e$ is the clock synchronisation bound (the maximum measurable difference between the readings of the two clocks at any instant of time). The comparable delay for a TMR node is $T + 2(d + e)$. A message with timestamp $T$ will be said to be stable, if no message with timestamp $T_1 < T$ can be received by an order process. Stable messages are queued at the relevant DMQs in increasing timestamp order (with care being taken to remove duplicate messages).

### 2.4. Node Architecture

The overall software architecture of a VOLTAN node is depicted in Figure 1, where the major components of the system within a processor of a node and their interactions are summarised. There are two main layers: the replication layer ensures that all message interactions of computational processes are agreed and ordered at all the replicas, while
the communication layer provides intra-node message communications facilities (note that a processor of a TMR node will contain a voter while a processor of a fail-silent node will contain a comparator).

![Diagram of VOLTAN node architecture]

**Fig. 1**: Software architecture of a VOLTAN node

For the purpose of sending and receiving valid messages, a processor maintains several message pools:

(i) *Received Message Pool (RMP)*: Contains valid received messages intended for ordering.
(iii) External Candidate Message Pool (EMP): Contains singly signed messages that have been received for voting (comparison).
(iv) Internal Candidate message Pool (IMP): Contains unsigned messages, each waiting for a signed message with identical sequence number to arrive in EMP.

Two operations are defined on a pool:
remove(pl,m): a message is removed from pool pl and assigned to m.
deposit(pl,m): message m is deposited in pl, if m is not already present in pl.

A message receive operation is defined on the DMQ, which returns the message at the head of the DMQ:
receivefrom(m): returns the message at the front of the DMQ of the calling process.

The calling process blocks if the DMQ is empty. The algorithm for the replicated version of a server process, S, discussed in the previous section is shown next:

```
process S_i : /* a replicated server */
cycle
    receivefrom(m)
    process the message
    deposit the result message in PMP
end
end S_i
```

The algorithms for the diffuser, receiver and the voter for a TMR node are given next, where we are making use of the following notation:

m.dest : destination node of message m.

Also, two message sending operations will be assumed:
diffuse(m): message m is signed and sent to the neighbour processors of the node for voting.
send(m): message m is signed and sent to m.dest node.

The function of the order process is to pick up a message from the RMP, order it using the services of the atomic broadcast protocol before placing it in the appropriate DMQ. Two primitive operations for atomic broadcasts will be assumed:

(i) Acast(m): m is broadcast to member processors of the node, including itself.
(ii) Areceive(m): receive a broadcast message.

The order process itself is composed of four cyclic processes: broadcaster, relayer (not shown in the fig.), transfer and deliver (their precise functions and interactions are discussed in a latter section). A queue of messages named Broadcast Message Queue (BMQ) is maintained by the order process. The cycle of the broadcaster consists of removing a message from the RMP and broadcasting it. The relayer and transfer processes are responsible for enqueuing received messages to the BMQ (the messages
in the BMQs of the non-faulty processors of a node will be ordered identically). Finally, the deliver process cycles by dequeuing a message from the BMQ and enqueuing it to the DMQ of the appropriate computational process, taking care not to enqueue a message that has been already enqueued (this is necessary as BMQ will contain replicas). Sequence numbers are used to detect and remove duplicate messages. The algorithms for the two processor fail-silent node are similar, with the exception that the comparator (which replaces the voter) of a non-faulty processor halts as soon as it detects a disagreement during a message comparison. Also, as mentioned before, the clock synchronisation and the ordering protocols have been simplified.

```
process diffuser :  
  var m: message  
  cycle  
    remove(PMP, m)  
    deposit(IMP, m)  
    diffuse(m) /* signed m is sent to neighbours */
end
end diffuser

process voter :  
  var m: message; me: host_node_identifier  
  cycle  
    m := a message from EMP identical to a message from IMP  
    /* such a pair is removed from these pools */  
    if m.dest ≠ me → send(m) /* double signed voted message */  
      is sent to its destination via  
      the transmitter process */
      m.dest = me → countersign and deposit(RMP, m) /* local message */
    f i
end
end voter

process receiver :  
  var m: message; me: host_node_identifier  
  cycle  
    receive(m)  
    if m is not authentic → discard /* corrupted message */  
    if m is signed once → deposit(EMP, m)  
    if m is signed twice → deposit(RMP, m)
end
end receiver
```

3. IMPLEMENTATION DETAILS
Efficient implementations of the protocols described in the previous section require that the processors within a node be capable of exchanging messages quickly. Ordering and voting/comparison take place within a node, so it is important to provide fast communication paths between the processors of a node. Transputers provide fast point to point communication links with just the kind of functionality we require. For this reason, we have chosen to implement VOLTAN nodes using T800 Inmos Transputers.
For the case of a TMR failure masking node, the three members of a processor triad are
directly connected to each other by transputer links, thereby providing fast
communication paths for the executions of clock synchronisation, broadcast and voting
protocols. Each processor involved in these protocols has direct links to its neighbours.
Elsewhere we have described how VOLTAN nodes can be used as building blocks for
constructing reliable distributed computing platforms [1, 9, 10, 11].

We have chosen to implement the replication and communication layers of Fig. 1 in an
object oriented language, C++ [12], and have used the facilities of the Helios operating
system [13], a Unix like operating system which runs on transputers and supports the
client/server model for structuring programs. These choices are not central to our
design and implementation, and have been taken mainly because we have extensive
Unix/C++ based systems programming experience. VOLTAN nodes can easily be
implemented using OCCAM. Some familiarity with C++ will be assumed in this
section. For the sake of uniformity, we will concentrate on the description of a TMR
node.

The VOLTAN system makes extensive use of classes, inheritance and virtual
operations to implement the software architecture shown in Fig. 1. Base classes exist
for processes (Active_Object), buffers (Queue) and messages (Message_Block). The
functionality of the system is then implemented in classes derived from these base
classes [14]. Our implementation has been performed in a layered fashion, reflecting the
structure shown in Fig. 1. The lowest layer utilises Helios services for providing basic
system services for constructing programs composed of active objects communicating
via message queues. These services are then used by the next layer, the communication
layer (shown in Fig. 1), which provides intra-node communication facilities. Then
comes the replication layer that implements the capability for replicated processing.

3.1. System Services

In Helios, processes are spawned by a call to the Fork() function. In the VOLTAN
system we encapsulate process spawning into a common Active_Object base class from
which individual process classes may then be derived.

In this way, the base class provides the thread of activity while the derived class
provides the algorithm by specifying the behaviour of the virtual operation main(). In a
class hierarchy, C++ constructors are executed in a bottom-up fashion; the
Active_Object constructor will be executed before that of the derived class. It is
conceivable therefore that main() will be called before the derived class has had time to
initialise the data structures used by main(). A semaphore started is used to prevent this
situation from occurring. Thus the derived class controls when main() becomes active.
The code for Active_Object is shown in Fig. 2. A parameter supplied to the derived
class constructor determines the priority of the thread. There are two priorities
available, high and low, corresponding to the priorities handled by the transputer scheduler.

```cpp
class Active_Object {
public:
    Semaphore started, terminated;
    Active_Object (priority p = LOW, word stack_size =
        Object_Stack_Size);
    ~Active_Object();
    virtual void main() = 0;
};

void inter_main(Active_Object *obj_ptr)
{
    Wait(&obj_ptr->started);
    obj_ptr->main();
    Signal(&obj_ptr->terminated);
}

Active_Object::Active_Object(priority p, word stack_size)
{
    InitSemaphore(&started, 0);
    InitSemaphore(&terminated, 0);
    if (p == LOW)
        {Fork(stack_size, (VoidFnPtr) inter_main,
            sizeof(Active_Object *), this);
        }
    else {HiFork(stack_size, (VoidFnPtr) inter_main,
            sizeof(Active_Object *), this);
    }
}

Active_Object::~Active_Object() {};
```

Fig. 2: The Active_Object Class

Two active objects can communicate with each other by accessing a common passive object via a pointer parameter passed to both constructors. Thus, for example, a Producer class object (Fig. 3) may communicate with a Consumer class object (Fig. 4) if the Queue pointer parameter passed to both constructors refers to the same Queue, as shown in Fig. 5.

Note that any implementation of a passive object class must consider issues of concurrency control. The Queue class allows concurrent active objects to enqueue() and dequeue() without interference. The Queue class object used in VOLTAN manages a collection of void pointers which may refer to arbitrary data structure elements, so all users of a queue must implicitly know the type of the elements. The interface to the Queue class is shown in Fig. 6.
class Producer : public Active_Object
{
    Queue *qp;
public:
    Producer(Queue *q_outp);
    ~Producer();
    void main();
};

Producer::Producer(Queue *q_outp): Active_Object(), qp(q_outp)
{ Signal(&started); }

Producer::~Producer() {};

void Producer::main()
{
    String *message;
    while(true)
    {
        message = new String("Hello World\n");
        qp->enqueue(message);
        Delay(OneSec);
    }
}

Fig. 3: Active Producer Derived Class

class Consumer::public Active_Object
{
    Queue *qp;
public:
    Consumer(Queue *q_inp);
    ~Consumer();
    void main();
};

Consumer::Consumer(Queue *q_inp): Active_Object(), qp(q_inp)
{ Signal(&started); }

Consumer::~Consumer() {};

void Consumer::main()
{
    String *message;
    while(true)
    {
        qp->dequeue((void **) &message);
        printf("%s\n", message->string());
        delete message;
    }
}

Fig. 4: Active Consumer Derived Class

main()
{
    Queue queue;
    Consumer consumer(&queue);
    Producer producer(&queue);
    while(true);
}

Fig. 5: Connecting Active Objects
class Queue
{
    virtual bool match(void *infop, void *q_item) { return true; }
protected:
    void *find(void *d_infop, bool auto_match, bool extract, 
        bool wait, bool abort);
    void insert(void *d_infop, bool auto_match, void
        q_item); 
public:
    Queue();
    ~Queue();
    bool enqueue(void *q_item);
    bool dequeue(void **q_itempp);
};

Fig. 6: The Queue Class Interface
In VOLTAN it has been found desirable to derive a class Message_Block_Queue from Queue to implement additional type checking. The construction of Message_Block_Queue is simplified by containing queue complexity and concurrency control entirely within the Queue base class. The derived class is also the mechanism through which additional queue operations may be performed. The Queue class provides two protected functions find() and insert(), and a virtual function match(). The derived class calls find() and insert(), and provides a real function match(). These services are used in VOLTAN by the Order processes for inserting and extracting messages from BMQ and DMQ. All the message pools and queues required by VOLTAN are declared as instances of Message_Block_Queue which is derived from the base class Queue.

class Message_Block
{
    Message data;
    Control_Block control;
public:
    Message_Block();
    Message_Block(Message_Block& mb);
    ~Message_Block();
    sign();
    bool authenticate();
};

Fig. 7: The Message_Block Class Interface
A VOLTAN message is implemented as a passive object Message_Block and is defined as a class which represents the structure of a message accepted by queues and transmitted across transputer links. A Message_Block contains a control component which handles all the system information relating to a message (for example, signatures, sequences numbers, timestamps etc.). Message blocks also store message data in the form of a sequence of bytes. The interface for class Message_Block is shown in Fig. 7. Hence the structure of the VOLTAN system consists of several processes (active objects) communicating asynchronously with each other via queues (passive objects) using message blocks (passive objects) to transfer information.
3.2. Communication Layer
Helios provides two different communication mechanisms: primitives for client-server processes and direct point-to-point communication over 'raw' links. Intra-node communication of VOLTAN uses this raw link-level services whereas inter-node communication presently uses the client-server facilities provided by Helios. Our basic approach is to distinguish between intra-node and inter-node communication. Intra-node communication is used for agreement and order, so for efficiency reasons, it takes place over the fast directly connected links between the processors of a node.

Using the link level primitives, together with the system services described in the previous sub-section, we have built a Neighbourhood Communication Service (NCS) for communicating messages between processors of a node. Each processor contains two active objects Tx and Rx. Tx dequeues messages from a queue specified in its constructor and sends them over a given link (also specified in the constructor). The Rx object awaits messages on a link specified in its constructor. It receives messages and inspects a control field within a message which specifies the queue on which it should be placed. It then enqueues the message on the appropriate queue. The possible destination queues are specified by passing the Rx constructor a pointer to an array of possible destination queues. In a TMR node, each processor runs two Tx and Rx objects - one for each link connected to a neighbouring processor of the node. Hence, to send a message to a neighbour, a process simply specifies the destination queue in the message control field and enqueues it on the appropriate queue. NCS will then ensure delivery of the message to the correct queue in the destination processor. Both Tx and Rx run at high priority so as to minimise communication delays within a node.

3.3. Replication Layer
The clock synchronization algorithm implemented is that proposed by Halpern et al. [8]. It consists of two active objects, which respectively monitor the time and await incoming clock synchronization messages, together with a passive object Clk which maintains the time. Consistency, with respect to concurrent accesses to the clock, is maintained by a semaphore. Clock synchronization objects communicate via the NCS and run at high priority so that the synchronization will be as tight as possible and will be relatively independent of the load on each processor.
class Diffuser: public Active_Object
{
    MBQ *PMF, *IMP, *LEFTLINK, *RIGHTLINK;
public:
    Diffuser(MBQ *pp, MBQ *ip, MBQ *ll, MBQ *rl);
    ~Diffuser();
    void main();
};

Diffuser::Diffuser(MBQ *pp, MBQ *ip, MBQ *ll, MBQ *rl):
    Active_Object(), PMF(pp), IMP(ip), LEFTLINK(ll),
    RIGHTLINK(rl)
{ Signal(&started); }

Diffuser::~Diffuser() {}

debug Diffuser::main()
{
    Message_Block *local_message;
    unsigned int seq_no;
    Table seq_table;
    while(true)
    {
        PMF->pop(&local_message);
        seq_no = seq_table.get_and_inc_seq_no();
        Message_Block *internal = new
            Message_Block(*local_message);
        internal->set_seq_no(seq_no);
        Message_Block *external_left = new
            Message_Block(*internal);
        external_left->set_type(RMP);
        external_left->sign();
        Message_Block *external_right = new
            Message_Block(*external_left);
        IMP->enqueue(internal);
        LEFTLINK->enqueue(external_left);
        RIGHTLINK->enqueue(external_right);
        delete(local_message);
    }
}

Fig. 8: The Diffuser Process

The diffuser and voter processes described in section 2 are relatively simple to implement given the infrastructure services described above. To illustrate this, example implementations of the diffuser and voter processes are shown in Figs. 8 and 9.

In the diffuser, messages are collected from the Processed Message Pool, and the next sequence number is provided by a sequence number object. Copies of the message are made so that it can be diffused to the neighbours. The copies of the message to be transmitted are signed after the type field of the message is updated so that the communication system will correctly deliver the message to the correct queue. Copies of the message are then pushed onto the relevant queues. The transmitter object Tx is responsible for deleting the messages sent to the processor's neighbours.
// Simple voter which uses checksums rather than
// byte by byte comparison of messages for voting

class Vote: public Active_Object
{
    MBQ *IMP, *EMP, *VMP;
public:
    Vote(MBQ *ip, MBQ *ep, MBQ *vp);
    ~Vote();
    void main();
};

Vote::Vote(MBQ *ip, MBQ *ep, MBQ *vp):
    Active_Object(), IMP(ip), EMP(ep), VMP(vp)
{ Signal(&started); }

Vote::~Vote() {}
void Vote::main()
{
    Message_Block *local_message, *remote_message;
    unsigned int local_seq_no, remote_seq_no;
    unsigned int local_checksum, remote_checksum;
    while(true)
    {
        IMP->pop(&local_message);
        local_seq_no = local_message->m.get_seq_no();
        local_checksum = local_message->m.get_checksum();
        do{
            EMP->pop(&remote_message);
            remote_seq_no = remote_message->m.get_seq_no();
            remote_checksum = remote_message->m.get_checksum();
            if ((local_seq_no == remote_seq_no) &&
                (local_checksum == remote_checksum))
            {
                remote_message->sign();
                VMP->push(remote_message);
                delete local_message;
            }
            else
            {
                delete remote_message;
            }
        } while ((local_seq_no != remote_seq_no) ||
            (local_checksum != remote_checksum))
    }
}

Fig. 9: A Simple Voter Process

The voter process collects a local message and tries to match it against messages
coming from its neighbours. When a match is found, the remote message is
countersigned and deposited in a message buffer for voted messages (VMP) for
transmission.

The order process consists of four active objects corresponding to the four processes
mentioned earlier. Three of these implement the standard signed message atomic
broadcast protocol [7] as follows. The Broadcast object picks up messages from the
RMP, timestamps them and diffuses them to the other processors in the triad. It also
inserts the message into the BMQ in increasing timestamp order (if two distinct
messages contain identical timestamps, they are ordered by their processor numbers).

The Relay object picks up broadcast messages and performs a timeliness check [7]. It
then inserts timely, valid messages into BMQ and relays them to the third processor of the triad. The Transfer object picks up relayed messages, performs a timeliness check and inserts accepted messages into the BMQ. The fourth active object, the Deliver object, takes messages from the BMQ, removes duplicates and enqueues the messages on the relevant DMQs.

3.4. Performance Issues

In this section we give some preliminary performance figures for a lightly loaded TMR and fail-silent nodes. Of principal interest will be the time taken to order a message. Clearly the performance of the system greatly depends upon the efficiency of the underlying system services. Measurement of queue access times yielded an average queue access time of 0.35ms. The performance of the Neighbourhood Communication Service was found to be 2.56ms. This figure represents the time taken to dequeue a message, disassemble it for communication, send it across a link and reassemble and enqueue it at the destination processor. Currently, messages are sent in two parts, the message content and the control information. It may be more efficient to combine this information into a single byte stream although this would require copying at least one part of the message at both source and destination. The trade-offs of this approach are currently under examination. Furthermore, at present, simple checksums are being used as signatures and so have a minimal impact upon system behaviour. The impact of using more complex signature mechanisms is yet to be assessed.

For a system of three processors with at most one faulty processor, the clock synchronisation algorithm [8] gives the maximum difference between the clocks of correct processors, $\epsilon$, as:

$$\epsilon = d + 2Rp$$

where $d$ is the maximum time required for sending a clock synchronisation message between the processors of a node, $R$ is the interval between resynchronisation and $p$ is the rate at which clocks drift. Terms of $O(p^2)$ and $O(pd)$ have been neglected in the above equation. Experiments under worst case circumstances determined the smallest safe value for the maximum transmission delay $d = 12ms$. In our system we chose $R = 5s$. Hence, assuming $p = 10^{-6}$, the second term in the above equation becomes negligible and we obtain $\epsilon$ to be equal to 12ms.

We take the time taken to order a message to represent the cost of replication. This time is determined largely by the overheads imposed by the underlying order protocol. In the TMR implementation, the termination time of the order protocol is: $2(d+\epsilon) = 48ms$. For a fail-silent node, with only two processors, the value of $d$ can be reduced to 8ms. Further, $\epsilon$ can be set to $\delta/2$, hence we fix $\epsilon$ to be equal to 4ms and the termination time becomes $d+\epsilon = 12ms$. The time taken up by message authentication, queue
manipulation and voting/comparison amounts to approximately 5ms, yielding an overall ordering delay of 53 ms for a TMR node and 17ms for a fail-silent node. We are investigating several ways of improving the above performance figures. We have obtained notable successes using approaches that optimise for the failure-free situation. For TMR nodes, we have devised an optimised ordering protocol based upon logical clocks and time-outs which substantially reduces the ordering overhead [15]. Using this protocol we have been able to reduce the overhead to approximately 28ms for a node running without any failed processor. A simplified version of this protocol can also be used for the case of fail-silent nodes. However, even greater efficiencies are possible if the ordering of messages is dictated by one of the two processors in a node. The processors have the roles of leader and follower with the leader dictating the order in which incoming requests are serviced. Using this leader/follower protocol, we have reduced the total overhead for fail-silent nodes to between 5 and 8ms. For a more detailed discussion of these protocols for fail-silent nodes see [16].

4. CONCLUDING REMARKS
We have described the design and implementation of VOLTAN TMR and fail-silent nodes using software implemented clock synchronisation, ordering and voting (comparison) protocols. Since synchronisation, ordering and voting/comparison take place within a node, it is important to provide fast communication paths between the processors of a node. In this respect, transputers with their point to point communication links have provided us with the right functionality. As this is our first attempt at building such nodes, we have been cautious in our approach, striving for logical simplicity and relying on the use of 'standard' well-known ordering protocols. However, we have taken care to structure our software in a modular fashion that permits easy modifications. We have now begun a critical examination of our design and implementation with the objective of improving the performance of our nodes beyond what we have achieved using standard protocols. To this end we have modified the design described here with new ordering protocols that hold the promise of substantial performance improvements [15, 16]. It is also important to be able to demonstrate experimentally that our nodes can, in practice, tolerate a large class of failures (Byzantine failures in the limit). For this purpose we have performed extensive fault injection experiments, not only to demonstrate the fault tolerance capability of a node, but also for uncovering any flaws in our implementation [17].

ACKNOWLEDGEMENTS
We would like to thank Alan Tully for his assistance in implementing the communication layer. This work has been supported in part by a grants from the UK Science and Engineering Research Council and CNPq/Brazil.
REFERENCES


