Principal features of the voltan family of reliable node architectures for distributed systems

S.K. Shrivastava, P.D. Ezhichelvan, N.A. Speirs, S. Tao and A. Tully

Abstract

A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance. The architecture of a family of such nodes with differing functionalities is presented. These include failure masking, fail-signal and fail-silent nodes. The software architectures of a three processor failure masking and a two processor fail-silent nodes are discussed in detail. The paper also discusses the suitability of VOLTAN nodes as building blocks of reliable distributed systems.
Bibliographical details

PRINCIPAL FEATURES OF THE VOLTAN FAMILY OF RELIABLE NODE ARCHITECTURES FOR DISTRIBUTED SYSTEMS

Principal features of the VOLTAN family of reliable node architectures for distributed systems
(By) S.K. Shrivastava [and others].

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1991.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 348)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 348
SHRIVASTAVA, Santosh Kumar

Abstract

A VOLTAN node is composed of a number of conventional processors on which application level processes are replicated to achieve fault tolerance. The architecture of a family of such nodes with differing functionalities is presented. These include failure masking, fail-signal and fail-silent nodes. The software architectures of a three processor failure masking and a two processor fail-silent nodes are discussed in detail. The paper also discusses the suitability of VOLTAN nodes as building blocks of reliable distributed systems.

About the author

Professor Shrivastava joined the Computing Laboratory in August 1975, where he is a Professor.

Suggested keywords

ATOMIC BROADCAST
DISTRIBUTED PROCESSING
FAULT TOLERANCE
RELIABILITY
REPLICATED PROCESSING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 621.381968
U.D.C. 519.687 681.322.022
PRINCIPAL FEATURES OF THE VOLTAN FAMILY OF RELIABLE NODE ARCHITECTURES FOR DISTRIBUTED SYSTEMS

S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao and A. Tully
Computing Laboratory
University of Newcastle upon Tyne, 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 architecture of a family of such nodes with differing functionalities is presented. These include failure masking, fail-signal and fail-silent nodes. The software architectures of a three processor failure masking and a two processor fail-silent nodes are discussed in detail. The paper also discusses the suitability of VOLTAN nodes as building blocks of reliable distributed systems.

KEYWORDS
Atomic broadcast, distributed processing, fault tolerance, reliability, replicated processing.
1. INTRODUCTION

Fault tolerant systems are often designed and implemented under a rather restricted fault assumption, which is that the underlying processors fail "cleanly" by just stopping. Such an assumption is hard to justify in computer systems intended for critical applications where failure probabilities in the range of $10^{-6}$ to $10^{-10}$ per hour are often specified [eg., 1]. It is then prudent to design and implement such systems under a highly unrestricted fault assumption, namely, that a failed processor, in principle, can behave in a fail-uncontrolled manner (in the literature this failure mode is often referred to as the Byzantine failure mode [2]). While certainly not common, experience has shown that Byzantine failures cannot be ruled out in the design of fault tolerant systems [3,4]. 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 Byzantine processor failures.

In a previous paper we have addressed the issues of building fail-controlled processors, called nodes, using ordinary fail-uncontrolled processors, and presented several architectures with differing functionalities and complexities [5]. The basic idea is conceptually simple: a node is built out of several processors which execute special protocols to carry out replicated processing of computations to achieve fault tolerance. Such fail-controlled nodes constitute the building blocks for constructing highly reliable distributed computing platforms. In this paper we present the detailed architecture of a family of fail-controlled nodes (named VOLTAN). At one end of the 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 n internal processor failures (where $N = 2^n + 1$ and $n \geq 1$) i.e. the failures are not apparent from outside the node. At the other end of the spectrum we consider a node which either works correctly, or stops functioning (becomes silent) as soon as an internal failure is detected. Such a node will be termed a fail-silent node. We will 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 VOLTAN family presented here should be of interests to system builders for several reasons. Because of the software approach adopted, the architectures presented here can be realised entirely by standard 'off the shelf' components (processors), without recourse to any specialised hardware. In this sense, these architectures represent the limits of what can be achieved using standard components; as such, system designers can use these architectures in making hardware-software tradeoff decisions in practical systems (for example, hardware assistance for clock synchronization, as suggested in [6] could be added for performance reasons, without altering any of the architectural principles). We also show how potential sources of non-determinism, such as timeouts, can be dealt with in a uniform manner by VOLTAN.
2. BASIC PRINCIPLES

2.1. System model and assumptions

We will assume the simple architecture shown in fig.1: a number of nodes, C_i, are connected by a perfect communications bus B (some specific examples of approximating B by redundant communication links and busses will be examined in section 5). Each node C_i maintains the abstraction of a fail-controlled node by replicated processing of computations over fail-uncontrolled processors P_{i1}..., P_{in}, n>1, which make up C_i. The protocols necessary for supporting this form of active replication will be discussed subsequently. Such protocols should strive to meet the replication transparency property: they should be local to a given node and should not require the services of any underlying system wide protocol. In particular this means that inter-node message passing protocols (between say C_i and C_j) should not be unduly affected by the replicated processing activities internal to the nodes. Thus, if a redundant system required the facility of a global time base, in the form of keeping the clocks of all the non-faulty processors of all the nodes synchronised, solely for the maintenance of fail-controlled abstraction, then we would claim that that system does not meet the replication transparency property.

![Fig.1: Basic architecture](image)

We will assume, as stated before, that a failed processor (and therefore the processes running on that processor) can exhibit fail-uncontrolled behaviour. We shall assume that the bus always allows correctly functioning processors within a node to communicate in a known and bounded time. 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 repeatedly 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 C: /* a typical server */
cycle
  receiveany(m)
  process m
  send (result_msg)
end C
```

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 [7]. It is also necessary to assume that the computation performed by a process on a selected message is deterministic. This assumption is fundamental to active replication. Given such a model of computation, active replication of a process, such as C (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 then identical output messages will be produced by them. This is the underlying principle of active replication [7].

Practical distributed programs often require some additional functionality such as using timeouts when waiting for messages. Timeouts (and other asynchronous events), prioritized messages etc. are potential sources of non-determinism during input message selection, making such programs difficult to replicate. In a subsequent section we will describe how VOLTAN provides the necessary functionality for dealing with such cases, and assume, for the time being, 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 [8,9] 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.

### 2.2. Failure-masking and Fail-signal Nodes

NMR replicated processing on N distinct processors with majority voting is a well known method of masking \( \pi \) processor failures (\( N=2\pi+1, \pi\geq1 \)). The failure-masking node to be described here has the following properties: (i) it functions correctly as long as the total number of processor failures within the node does not exceed \( \pi \); and, (ii) any spurious messages emitted by the failed processors of a correctly functioning node can be detected and rejected by all the correctly functioning receiver nodes. Thus a correctly functioning node emits correct messages and, possibly, spurious messages, which can be recognised as such by all correctly functioning receivers.
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. We convert the problem of identical message selection to that of identical message ordering by serialising the inputs to a process. This can be done by presenting a single input message queue, referred to as a valid message queue, VMQ, to a process and ensuring that a process picks up the message at the head of its VMQ for processing. An atomic broadcast protocol (see below) 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. Several mechanisms are necessary to implement this facility.

The clocks of all the non-faulty processors of a node are assumed to 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 10]. Each non-faulty processor of a node is assumed to be capable of atomically broadcasting a message to all other non-faulty processors of that node. We require the properties of atomicity, validity, termination and order from the atomic broadcast mechanism. When a sender broadcasts a message at its local clock time T, then:

(ab1): the message is delivered either to all non-faulty receivers or to none of them (atomicity);
(ab2): if the sender is non-faulty then the message is delivered to all the non-faulty receivers (validity);
(ab3): if the message is delivered then it is delivered to every non-faulty receiver (including the sender, if it is non-faulty) by the sender’s clock time T+Δ where Δ is some known bounded quantity (termination);
(ab4): the ordering of the delivered messages is identical at all the non-faulty receivers (order).

Algorithms for achieving atomic broadcasts (using authenticated messages) in the presence of Byzantine failures exist [11].

Each non-faulty processor of a node runs the following four ‘system’ processes:

(i) Sender 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 π or less number of distinct signatures are sent to the local voter process; messages with π+1 distinct signatures 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 signatures on m as part of the message). If the counter signed message m now contains a total of π+1 distinct signatures, then m is regarded as a valid (voted) message and it is sent over the network to its destination. On the other hand, if m contains a total of π or less number of distinct signatures, then it is sent to all those processors of the node who have not
yet signed m. 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 \( \pi+1 \) signed.

(iv) Order Process: this process atomically broadcasts 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 for processing.

Note that both clock synchronization and atomic broadcasts are performed locally within a node. Thus the main "visible" difference between an NMR system and its non-replicated counterpart is that the nodes are required to produce \( \pi+1 \) signed messages and use authentication. Further, under normal conditions, inter-node message traffic is increased by \( 2\pi+1 \) times. Nevertheless, this architecture largely satisfies the replication transparency property stated previously.

Elsewhere we have shown how such an NMR node can be enhanced easily to behave like a fail-signal node [5]. A \( \pi \)-fail-signal node is composed of \( N=2\pi+1 \) processors, where, as before, \( \pi \geq 1 \). It generates a 'fail-signal' (a failure exception) if one or more of its processors fail, provided the total number of failures does not exceed \( \pi \). A non-faulty processor can detect failures while executing the agreement and order protocols, and during voting [5]. A fail-signal node can continue to function correctly even after signalling a failure, but subsequent processor failures within the node may cause the node to fail. Therefore, such a signal can be used for initiating the migration of (critical) computations from the signalling node to other nodes.

2.3. Fail-silent nodes

Intuitively, the fail-silent property ought to mean that a node fails just by becoming 'silent', implying that the node can only generate either correct outputs or upon an occurrence of an internal failure, becomes silent for ever. However, this behaviour is impossible to enforce, due to the fact that a failed processor, in principle can emit arbitrary messages (for example, a processor can fail while transmitting a message). A \( \pi+1 \) processor fail-silent node, where, \( \pi \geq 1 \), will be said to exhibit 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, in which case the receivers can detect any messages it does produce as invalid. This behaviour is guaranteed so long as no more than \( \pi \) processors in the node fail.

The NMR node architecture discussed previously can be modified easily to construct a \( \pi+1 \) processor fail-silent node. Each non-faulty processor of a node runs the four system processes, with the sender, receiver and order process performing the same functions as their NMR counterparts. The voter process is replaced by a comparator process which acts like the voter process, except a disagreement during a comparison is treated as an internal failure. Once a failure is indicated, the comparator process stops, which results in no further \( \pi+1 \) signed message being produced from that node. A correctly functioning node will generate \( \pi+1 \) identical copies of its output messages. When the comparator process of a non-faulty processor in a node detects a failure and therefore stops, no new \( \pi+1 \) signed messages can be emitted by that node; any messages coming from this node that are not \( \pi+1 \) signed will be found to be invalid at the receiving nodes.
Having discussed the principles of operations of the VOLTAN family of nodes, we will next discuss their detailed system architecture.

3. NODE ARCHITECTURE

We will concentrate on the two of the most practical node architectures: a three processor fail-masking TMR (Triple Modular Redundant, where, N=3 and π=1) node, and a two processor fail-silent node. Enhancing the TMR node to incorporate the fail-signal property is fairly straightforward [5], so will not be discussed here. 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 (initialized 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 message for matching. We will refer to double signed messages that have been authenticated as valid messages.

The overall software architecture of a VOLTAN node is depicted in fig. 2, where the major components of the system within a processor of a node and their interactions are summarized. 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 atomic broadcast and other message communications facilities.

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

(i) **Valid Message Pool (VMP)**: Contains valid received messages intended for processing.

(ii) **Processed Message Pool (PMP)**: Contains unsigned output messages produced by computational processes. These messages must be validated: checked by the voter (comparator) before transmission to the final destination.

(iii) **Received Message Pool (RMP)**: Contains authenticated messages that have been received for validation.

(iv) **Candidate message Pool (CMP)**: Contains unsigned messages, each waiting for a signed message with identical sequence number to arrive in RMP.

Two operations are defined on a pool:

- `remove(pool,p):` a message is removed from pool `pool` and assigned to `p`.
- `deposit(pool,p):` message `m` is deposited in `pool`, if `m` is not already present in `pool`. 
A message receive operation is defined on the VMQ, which returns the message at the head of the VMQ:

\[ \text{receivefrom}(m) : \text{returns the message at the front of the VMQ of the calling process.} \]

The calling process blocks if the VMQ is empty. The algorithm for the replicated version of a server process, C, discussed previously is shown next:
process \( C_i \): /* a replicated server */

\begin{align*}
\text{cycle} \\
\text{receivefrom}(m) \\
\text{process the message} \\
\text{deposit the result message in PMP} /* the message contains a sequence number */ \\
\end{align*}

end \( C_i \)

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

\( m.\text{dest} \): destination node of message \( m \).

\( m.\text{destproc} \): destination process of message \( m \).

\( m.\text{sequencenumber} \): sequence number of message \( m \).

Also, two message sending operations will be assumed:

\( \text{diffuse}(m) \): message \( m \) is signed and sent to the neighbour processors of the node for voting.

\( \text{send}(m) \): message \( m \) is signed and sent to \( m.\text{dest} \) node.

\begin{align*}
\text{process sender :} \\
\text{var m:message} \\
\text{cycle} \\
\text{remove(PMP}, m) \\
\text{deposit(CMP}, m) \\
\text{diffuse}(m) /* signed } m \text{ is sent to neighbours */} \\
\end{align*}

end sender

\begin{align*}
\text{process receiver :} \\
\text{var m:message; me:host_node_identifier} \\
\text{cycle} \\
\text{receive}(m) \\
\text{if } m \text{ is not authentic } \rightarrow \text{discard} /* \text{corrupted message */} \\
\text{m is signed once } \rightarrow \text{deposit(RMP}, m) \\
\text{m is signed twice } \rightarrow \text{deposit(VMP}, m) \\
\text{fi} \\
\end{align*}

end receiver

\begin{align*}
\text{process voter :} \\
\text{var m:message; me:host_node_identifier} \\
\text{cycle} \\
\text{m := a message from RMP identical to a message from CMP} \\
/* such a pair is removed from these pools */ \\
\text{if } m.\text{dest}=\text{me} \rightarrow \text{send}(m) /* double signed voted message} \text{ is sent to its destination */} \\
\text{m.\text{dest}=me } \rightarrow \text{countersign and deposit(VMP}, m) /* local message */ \\
\text{fi} \\
\end{align*}

end voter

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

(i) \text{Acast}(m): m \text{ is broadcast to member processors of the node, including itself.}

(ii) \text{Areceive}(m): receive a broadcast message.
The order process itself is composed of three cyclic processes: *broadcaster*, *receiver* and *feeder*. 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 VMP and broadcasting it using *Acast*. The cycle of the *receiver* consists of receiving a message by calling *Areceive*, and enqueuing the received message to the BMQ (the messages in the BMQs of the non-faulty processors of a node will be ordered identically). Finally, the *feeder* process cycles by dequeuing a message from the BMQ and enqueuing it to the VMQ of the appropriate computational process, taking care not to enqueue a message that has been already enqueued (this is necessary as the BMQ can contain replicas).

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. Since there are only two processors in a node, the clock synchronization and the atomic broadcast protocols can be made particularly simple, as they are expected to work only in the absence of failures.

### 4. Enhancing the Computational Model

In the state machine model assumed so far, a process selects any one of the input messages from its input ports for processing (using *receiveany(...)* primitive). As a consequence of this assumption, in the replicated version, where a process uses *receivefrom(...)* primitive, it is acceptable to return messages in the order specified by the order process. However, it is often desirable for a process to exercise some control over the selection of messages (for example, a process may wish to wait for a message from a particular input port with a time-out). This section examines a range of possible selection criteria and shows how they can be incorporated in the VOLTAN architecture. The approach presented here is based on our work reported in [12]; here we present only the essential aspects. Message selection can be based on either blocking or non-blocking inputs:

(i) **Selective blocking inputs**: a process may specify a subset of its input ports from which it is prepared to accept a message at any given point, but must accept one before continuing (note that the *receiveany()* primitive is a special case when no such subset is specified). This could happen when a server is acting as a resource allocator and under some condition is prepared to listen to only some subset of its clients. If the message supplied first by the order process is not in the subset then we have deadlock.

(ii) **Non-blocking inputs**: a process may specify a subset of its input ports from which it is prepared to accept a message but may continue execution instead of accepting one if no message is available. When we consider replicating such a process we must ensure that either all the replicas accept the same message or they do not and continue execution. Since messages can arrive at replicas at different times, just preserving the order is not sufficient to cope with non-blocking inputs.

Selective blocking inputs can be handled correctly in the replicated version, and deadlocks prevented, if processes are allowed to search the VMQs, starting from their heads, for the first acceptable message. Since messages in the queues of replicas are ordered identically, this solution will work. More complex mechanisms are required to solve the second case of non-blocking inputs. Our approach is to
modify the `receivefrom(m)` primitive which a replicated process uses to get a message from its VMQ to a `generic input primitive`:

\[
\text{receivefrom}(S: \text{set of ports, } m: \text{message, } t: \text{timeval}): \text{ returns a message } m, \text{ within time } t
\]

\[\text{from any of the input ports } S; \text{ if no message is available (the timeout } t \text{ expires), then a null message is returned.}\]

We call this primitive generic, since by setting the parameters appropriately one can get the functionality of either the blocking or non-blocking inputs (for example, if \( S \) is set to all and \( t \) to infinity, then the functionality of `receiveany` (...) is obtained), thus it is sufficient to solve the non-determinism problem for this generic primitive.

```plaintext
procedure receivefrom(S: setof ports, m: message, t: timeval):
begin
within t do
\{m := RECEIVEFROM(S)\}
ontimeout:
\{deposit a self directed null message in PMP \}
m := RECEIVEFROM([S, self])
end receivefrom
```

The algorithm for the generic input function can be implemented as shown. We assume a blocking primitive `RECEIVEFROM(S)` which scans the VMQ of the process to return the first message from a port in \( S \). The process executing the generic input function waits for a maximum duration \( t \) to receive a message. Assuming the timeout does not occur at any of the replicas, the blocking `RECEIVEFROM(S)` primitive at each replica will return identical messages. If no message is received by a process within the duration \( t \), then the timeout clause will be executed by that process. The process then sends a null message to itself and waits for either this self-directed message or a message from a member of \( S \). Consider now the specific case of an NMR node: if a `majority` of replicas timeout, then a self-directed message will be majority voted and will arrive at the VMQs of all the non-faulty replicas. The order processes will ensure that messages are identically ordered at all the replicas. Therefore either all replicas will receive the same message from a member of \( S \) first or all will receive the self-directed message first. The case for a \( \pi+1 \) fail-silent node is similar, where a comparison between \( \pi+1 \) messages must take place. So, if one of the replicas receives the message \( m \) before the timeout, the rest will receive \( m \) as well, since \( m \) will eventually be available at all the VMQs (this will happen even if some other replica times out locally: no processor will get \( \pi+1 \) null messages, so no comparison will take place). A timeout will only succeed if the following happens: all the replicas timeout, and the null message gets ordered before \( m \).

The power of our approach derives from the fact that blocking and non-blocking message input operations (with and without timeouts) have been converted to message selection operations on identically ordered queues. Since identical ordering can be maintained in the presence of Byzantine faults, our approach is capable of tolerating such failures. Of course, the generic input function is much more complex now, but this price has to be paid if application requirements dictate the enhancements
to the computational model assumed in section 2. The generic input function can be further enhanced to process messages with priorities; these and other aspects are discussed in [12].

Care is needed for the generation of sequence numbers, since it is possible for a non-faulty process to produce a null message which is not necessarily produced by any of the other non-faulty replicas: the counter values can therefore diverge. A simple remedy exists. If a process, after producing a null message finds that a normal message is returned by the input function, then it decrements its counter by one. Similarly, if a process receives a null message from the input function, but it itself did not produce a null message, then it increments its counter by one. Note that null messages are sent locally within a node, and valid messages produced by a node will always contain monotonically increasing sequence numbers.

5. SPECIFIC SYSTEM ARCHITECTURES

We present two examples of systems architectures to illustrate how VOLTAN nodes can be used in distributed systems. We will also show how redundancy in communication paths can be incorporated. 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 should take place over fast directly connected links between the processors of a node. First we discuss a replicated (TMR) systems architecture using point to point links.

5.1. A TMR system using point to point links

We develop a specific communications architecture necessary for supporting replicated TMR processing [13]. This architecture exploits the following property that we assume for all of the processors: each processor has a fixed number of communication links through which processes executing on that processor may send or receive messages from processes of other connected processors. Some current microprocessors such as the Inmos Transputers provide just the kind of communication facilities assumed here (a Transputer has 4 bi-directional 10Mbit/sec serial communication links). Using such links, large multi-Transputer networks can be built for specific applications. A number of these applications make use of Transputer farms (either single or two dimensional Transputer arrays) for parallel processing. The architecture developed here can be used for such applications.
We will assume that a processor has four bidirectional links. Four links are enough to construct a pipeline (or a ring) of nodes as shown in Figure 3. The pipelined nature of interconnection means that a message intended from one node to some other node may have to be relayed by intermediate nodes. From a TMR system we require the capability of masking at most one processor failure per node. Under such a failure assumption, the pipelined system possesses the following two properties:

(i) Any non-faulty processor of a node is directly connected to all other non-faulty processors of the same node.

(ii) From any non-faulty processor in a node \( C_i \), there is a non-faulty path connecting that processor to any non-faulty processor of node \( C_k \). A path between any two non-faulty processors is non-faulty if all the intermediate processors in the path are non-faulty (for example, the path drawn in bold lines in the Figure is non-faulty).

We will refer to a link connecting processors of a node as internal and a link connecting processors of two adjacent nodes as external. Then a primitive operation for relaying a message over an external link is required:

\[ \text{diffuse}_{\text{external}}(message) : \] This operation is invoked by a relaying processor if it receives a double signed message on one of its links. If a message is received on an external link, then the processor sends the received message on all internal links as well as the external link which is towards the direction of the destination. If a message is received on an internal link, then it is only forwarded outwards via the appropriate external link.

Given this primitive, VOLTAN nodes can be pipelined as shown in fig.3 [13]. If more than four links per processor are available, then structures other than a pipeline can be formed, for example, a triplicated grid structure which requires six links per processor (if necessary, a six link "processor" can be built using two four link processors).
5.2. Using two-processor fail-silent nodes

A reliable distributed systems architecture composed of two processor fail-silent nodes connected by a dual redundant busses (e.g., LANs) is shown in Fig. 4. As before, we assume that the processors of a node are directly connected by a link for intra-node communications, so the LANs are used for inter-node communications only. Each processor has two network interfaces for connection to the network, enabling the comparator of a processor to send valid messages over both the LANs. On such an architecture, 'node level' processes can be replicated on distinct nodes for increased availability (a node level process itself is composed of two processes, one each on the underlying processors, and behaves like a fail-silent process). In particular, such a system architecture can be used for building highly available services by constructing K-resilient node processes: a K+1 replicated node level process (K>0) can tolerate a maximum of K replica failures before a subsequent failure makes the services it is providing becoming unavailable. In a separate paper we have shown how protocols for group communication between node level processes, necessary for supporting such services, can be implemented to run on two processor fail-silent nodes [14]. Such an architecture can form the basis of building responsive and reliable distributed applications.

![Fig. 4: A distributed systems architecture employing fail-silent nodes](image)

6. IMPLEMENTATION DETAILS AND FUTURE PLANS

An instance of the VOLTAN architecture is currently in the final stages of implementation. It is being implemented on a network of T800 transputers to produce a TMR failure masking node (the three members of a processor triad are connected by transputer links as indicated in Fig. 3). This system can easily be adapted to function like a two processor fail-silent node. The software has been implemented to run on the Helios operating system [15]. The software implementing the replication and communication layers (Fig. 2) have been written in C++ and is structured as a set of active objects which communicate by passing message blocks via queues [16].

The required message pools and queues are declared as instances of a C++ base class Queue. The base class Queue provides (concurrency controlled) operations to push and pop message blocks onto queues.
However, it also contains private member functions to find a message on a queue, insert a message onto a queue and match a message against one in the queue. Where necessary, these operations are used by classes derived from Queue; for example in the implementation of the voter process (matching message in RMP and CMP) and in the feeder process (inserting messages in the VMQ).

A message block is defined as a class and 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, sequence numbers etc.). Message blocks also store message data in the form of a sequence of bytes.

The class Active-Object allows any class derived from it to contain an active thread running at a specified priority. Any system class requiring a thread is derived from active object and contains a member function 'main()' which contains the thread of control. Hence the structure of the Voltan system consists of several active objects communicating asynchronously with each other via queues using message blocks to transfer information. Active objects are also used for performing basic link level communications.

The clock synchronization algorithm implemented is that proposed by Halpern et al. [10]. For a system of three processors with at most one faulty processor, the maximum difference between the clocks of correct processors is of the order of two milliseconds. Currently simple checksums are being used as signatures. The atomic broadcast protocol is the standard signed message algorithm for Byzantine agreement [11]; the constant $\Delta$ is of the order of ten milliseconds. These are preliminary figures, and we expect substantial improvements as the implementation is tuned for performance (see below).

We intend to run a number of sample applications to test that our implementation does handle message selection correctly. Then we plan to extend our work in two directions. First of all, we want to adapt our software to bypass the operating system; this is essential to improve the performance of the clock synchronization and broadcast protocols, which are currently executed as user level processes. Secondly, we want to examine the failure detection coverage obtained from the double signature mechanism, starting from simple checksums to digital signatures. For this purpose we plan to perform extensive fault injection experiments, using tools such as those reported in [17]. Through these experiments we hope to establish the practical utility of the architecture.

7. CONCLUDING REMARKS

The VOLTAN family of node architectures presented here can be regarded as representing the limits of what can be achieved using standard 'off the shelf' components (processors, network interfaces etc.) to construct highly reliable nodes as the building blocks of distributed systems. We have followed the approach, pioneered by the designers of the SIFIT system [1], of using software implemented agreement and order protocols for supporting replicated processing. We have also shown that it is possible to support quite expressive models of message passing computations. The main advantages of this approach are that: (i) technology upgrades appear to be easy; since the principles behind the protocols do not change, the protocol software can be easily ported 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, a measure of tolerance against design faults in processors can be obtained, again without recourse to any specialized hardware assistance; and (iii) since the replicated computations are loosely synchronised, the architecture is likely to be capable of detecting common mode transient failures [18] (this is because transients are unlikely to affect the computations on the processors in an identical fashion). However, as we indicated in section 6, further work remains to be done before the practical utility of this approach can be firmly established.

The work reported here has therefore not yet reached the degree of maturity achieved by system builders employing special purpose hardware. We discuss a few such examples. The approach adopted in [4, 19] relies on a sophisticated hardware implemented fault tolerant clock source for a node: the copies of processors are tightly coupled to execute a given instruction in an identical number of clock cycles. This has the advantage that there is no need for the sophisticated protocols that a VOLTAN node requires for message selection. The Sequoia fault tolerant system [20] is another example of this approach: here a fail-silent node is built using two tightly coupled processors that check each other (the VOLTAN fail-silent node can be regarded as its 'software' equivalent). The DELTA-4 system is a good example of what can be achieved, by judicious hardware support, in building dependable distributed systems [3]. Here, specially designed fail-silent Network Attachment Controllers (NACs) have been used for constructing a communications subsystem capable of providing the atomic broadcast facility over a LAN (the NACs use the tightly coupled approach for achieving fail-silence). This permits a very flexible approach at the application level: if the host nodes are assumed to be fail-uncontrolled, then application level processes need the replication degree of 2π+1 to tolerate π replica failures; on the other hand, the degree required is only π+1 if the hosts are regarded as fail-silent. A high performance version of the DELTA-4 system, XPA, specially intended for fail-silent hosts is also under development [21]. Various integrated hardware approaches to building fail-silent NACS and hosts are being investigated. The architecture presented in section 5.2 can be regarded as providing the same functionality, but avoiding the need for any special hardware support. It is our intention to evaluate these complementary approaches.

ACKNOWLEDGEMENTS

This work has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project 2252 (DELTA-4).

REFERENCES
