The Voltan Application Programming Environment for Fail-silent Processes

D. Black[†], C. Low[‡] and S.K. Shrivastava[†]

[†]*Department of Computing Science*
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

[‡] *Hewlett-Packard Research laboratories*
Bristol, BS12 6QZ, UK

Abstract

The Voltan software library for building distributed applications provides the support for (i) a process-pair to act as single Voltan self-checking 'fail-silent' process; and (ii) connection management for Voltan process communication. A Voltan fail-silent process is written by the application developer as a single threaded program. The Voltan system replicates this program transparently. The active replication of applications engenders problems when dealing with non-deterministic calculations. This paper outlines the mechanisms deployed by Voltan to deal with non-determinism. The current implementation can achieve a level of performance that is suitable for many real-time applications. The work described in the paper provides a way of solving the challenging problem of constructing fault tolerant distributed computing systems capable of tolerating Byzantine failures, using general-purpose, low cost components. The present practice is to employ hardware based approaches to construct a 'fail-silent' node using a self-checking processor pair working in lock-step. However this approach is very costly in terms of the engineering effort required, and further, as processor speeds increase, keeping a pair in lock-step execution may prove difficult.

Key words: distributed systems, fault-tolerance, Byzantine failures, replication, non-determinism.

(paper published in Distributed Systems Engineering, 5, June 1998, pp. 66-77).

1. Introduction

A dominant assumption made in software implemented fault-tolerant mechanisms, such as message logging, checkpointing, process replication for availability, (e.g., [1, 2]) is that the processing elements will suffer only crash failures, i.e., a processing element will either perform correct state transitions or will crash by ceasing to function. To meet this assumption in a realistic manner, we assert that some form of self-checking facility will be required within an element to detect a faulty state transition and stop the element from producing any further outputs. It is nevertheless true that in most non-mission critical applications, it is common to assume that conventional processors, without any self-checking capabilities, will suffer only crash failures. We are interested in applications where self-checking is deemed necessary. There are many applications that are safety-critical and/or demand very high degree of availability (e.g., telephone switching systems that are characterised by a maximum downtime requirement of 3 minutes per year). It is prudent to design and implement such systems under a highly unrestricted fault assumption, namely, that a failed processor and its processes, in principle, can behave in a *fail-uncontrolled* manner (in the literature this failure mode is often referred to as the Byzantine failure mode). While certainly not common, experience has shown that Byzantine failures cannot be ruled out in the design of fault-tolerant systems.

One particular approach is to replace each ordinary processor by an equivalent *fail-silent node,* built out of a collection of ordinary processors that check each other, that either works correctly, or stops functioning (becomes silent) soon after an internal failure is detected. This behaviour of a node, made out of π+1 processors, is guaranteed so long as no more than π processors in the node fail. A two processor fail-silent node (π=1) offers a practical and economical solution to the problem of constructing fail-silent nodes. Hardware implementations of two processor fail-silent nodes have been in use widely, for example in commercial transaction processing systems (e.g. [3]) and in telephone switching. Such nodes have been designed with the assistance of specialised comparator hardware, bus interface and clock circuits. A common (reliable) clock source is used for driving a pair of processors which execute in lock-step, with the outputs compared by a (reliable) comparator; no output is produced once a disagreement is detected by the comparator. A problem with this approach is that every new microprocessor architecture requires substantial design overheads; furthermore, lock-step synchronisation at very high clock speeds (100 MHz and above) may well turn out be difficult to achieve.

An alternative approach that seeks to reduce (or eliminate altogether) the hardware level complexity associated with the approach discussed above is to use standard *off the shelf* processors but maintain replica synchronism at a higher level, for instance at the process level by making use of appropriate software implemented protocols. We have been investigating such implementations of fail-silent nodes that rely purely on software implemented protocols for the management of redundancy [5]. We have designed and implemented a family of failure-masking and fail-silent nodes called Voltan [5, 6]. We have followed the approach, pioneered by the designers of the SIFT system [7] for supporting replicated processing. Unlike SIFT, our nodes are capable of supporting quite general purpose message passing programs.

The software approach also makes it possible to apply the fail-silence measures *selectively,* only to those processes that are deemed critical in a given application. The Voltan system software developed by us is sufficiently lean, making it practical to use it as a software library at process level, giving rise to self-checking process-pairs (*Voltan processes*). Each member of a process-pair contains a number of threads that implement the Voltan self-checking mechanisms. The Voltan system software permits a collection of distributed processes to be replicated transparently giving an equivalent collection of self-checking processes (Voltan processes). Practical distributed programs employ a variety of non-deterministic mechanisms for controlling interactions between processes (e.g., non-deterministic message selection using time-outs). Unchecked, non-determinism within replicas could lead to divergence of states. The Voltan system software has simple but effective measures for controlling non-determinism. The current UNIX-based implementation deals with a large subset of UNIX system calls.

This paper presents the design and implementation of the Voltan software for fail-silent processes; it also presents performance figures. The current implementation can achieve a level of performance that is suitable for many real-time applications. Our implementation is sufficiently realistic to enable software implemented fault-tolerance mechanisms, such as those mentioned at the beginning of this paper to be implemented using Voltan processes. The overall conclusion from this work is that our approach has opened up a very flexible way for meeting the fail-silent assumption using general-purpose, low cost components, such as commodity UNIXä or Windows NTä servers and LANs.

2. The Voltan Algorithm

2.1. Rationale

The approach used in Voltan for replica synchronisation, called the leader-follower approach, has been arrived at after extensive design and implementation effort [6]. We note that the performance of a fail-silent node (process) will depend on how quickly messages can be ordered and compared. Ordering can be achieved in several ways. We first performed a reference implementation of a fail-silent node; this implementation made use of a well-known synchronised clock based message order protocol. The reference implementation was then modified to yield a logical clock based implementation, and later to yield the leader-follower implementation described here. All these implementations were tested extensively for performance. The results obtained indicated that adopting the leader-follower mechanism within a fail-silent node represents the best design choice [6].

2.2. Basic Assumptions

We assume that a failed process can exhibit fail-uncontrolled Byzantine behaviour. We however assume that each non-faulty process is able to *sign* a message it sends by affixing the message with a message dependent unforgeable signature; a non-faulty process is also assumed to be able to *authenticate* any signed message it receives. Digital signature based techniques provide a very comprehensive way of meeting this functionality. Thus tolerance against (authentication detectable) Byzantine failures is being considered.

We assume that non-replicated distributed computations are composed of a number of processes that interact only via messages. As an example, the function of a typical 'server' process is to cycle by selecting an input message from any one of its input ports, process it and, if necessary, output one or more messages on its output ports. We assume (unless stated explicitly) that the computation performed by a process on a selected message is *deterministic.* This is the well known *state machine* model (where a state machine is a process) for which the precise requirements for supporting replicated processing are known [8]. Basically, in the replicated version of a process, multiple input ports of the non-replicated process are merged into a single port and the replica selects the message at the head of its port queue for processing. So, if all the non-faulty replicas have identical initial states then identical output messages will be produced by them, provided the queues of all correct replicas can be guaranteed to contain identical messages in an identical order. Thus, replication of a process requires 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.

Practical distributed programs do often perform non-deterministic processing such as using 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. Voltan has efficient mechanisms for dealing with non-determinism using the techniques presented in [9,10,11].

2.3. System structure

Figure 1 shows the logical view of a Voltan process. A Voltan process receives and sends messages to other Voltan processes via its input/output ports.

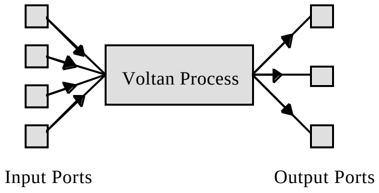

Figure 1: A Voltan process

To meet the property of fail-silence, a single logical process is formed from two replicas. As each replica forms an output message, it signs that message and passes a copy over to its partner. When a replica receives a signed output message it compares it with its locally generated result. If the comparison is successful the replica signs that message (which now has two signatures) and outputs it via the specified port. If a comparison fails then that signals a state divergence and hence a failure. At this point the replica process terminates.

From the above description it can be seen that a fail-silent Voltan process will either output correct messages or, detectably incorrect messages. Detectably incorrect messages (which can be at most singly signed) may originate from a replica which has failed (for example its comparator may be ill-functioning). Note that a fail-silent process only *detects* a failure and does not *mask* it. Higher level mechanisms, such as replicated fail-silent processes would be required for this purpose.

Several schemes for achieving agreement and order requirements have been investigated for Voltan [6]. In the most efficient scheme, the two replicas which form a logical process are assigned roles. One is termed the leader, the other the follower. The leader is responsible for setting the order in which the messages are to be processed and signalling that order to the follower. The two replicas of an application could be running either on the same processor or on distinct processors (this would be the preferred configuration providing tolerance against common mode failures).

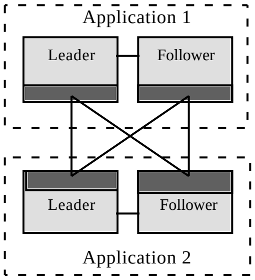

Figure 2: Interconnection of applications.

Figure 2 above shows the logical connection pattern between two process-pairs; such a connection arrangement offers maximum tolerance against communication failures. A typical hardware configuration supporting such a communication structure is shown below that uses dual-redundant network (Figure 3.)

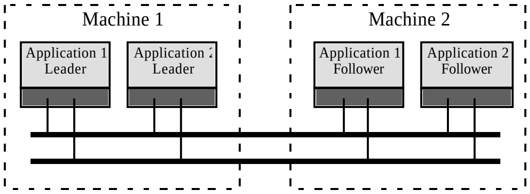

Figure 3: Interconnection of Voltan applications using a redundant network

The method of operation for a fail-silent process pair is shown in Figure 4.

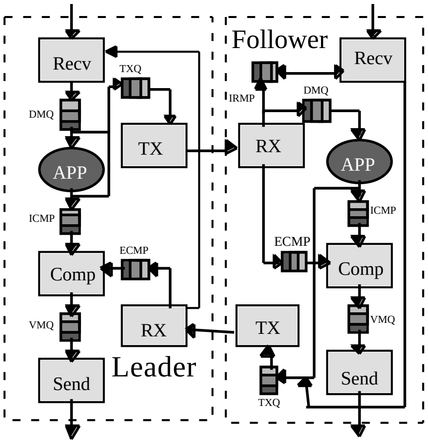

Figure 4: The structure of a Fail-Silent process

The system is structured as a number of co-operating threads of control. Each thread operates independently communicating via message queues. In all there are six threads each within the leader and the follower (this includes the application thread). We will use the term ‘Voltan system’ to refer to the all the software within a process minus the application thread software. The system operates as follows.

The receiver thread of the leader (Recv) accepts only new and authentic double signed messages for processing (discarding the rest including any duplicates). Accepted messages are placed in the application’s Delivered Message Queue (DMQ). The application thread selects a message from this queue for processing, depositing at the same time a copy of this message into the Transmit Message Queue (TXQ) for forwarding to the follower via the Transmission thread (TX).This way of directing selected messages to the follower permits the leader to use complex message reception criteria (such as using message priority) ensuring at the same time that the follower will select identically (this aspect will be discussed further later). The communication protocol between a TX and the corresponding Reception thread (RX) ensures reliable FIFO delivery.

When the Reception thread (RX) on the follower receives a doubly signed message it places it into the DMQ of the application running under its control; a copy of the message is also deposited in Internal Received Message Pool (IRMP) for detecting lost messages as discussed later.

The application threads on both replicas proceed to calculate the next state. When that state is formed an output message is created; a copy of this message is signed and is placed into the TXQ for transmission over to the other replica. The unsigned message is stored locally in the Internal Candidate Message Pool (ICMP) for later comparison. We assume that application threads assign monotonically increasing sequence numbers to new messages they produce; this property enables correctly functioning destination processes to discard replicas of any previously received messages.

When the Reception thread receives a singly signed message it places it within the External Candidate Message Pool (ECMP) for the purpose of comparison.

The Comparison thread (Comp) compares messages with identical sequence numbers one each from ICMP and ECMP (stripping the message signatures before comparison). If the comparison succeeds then the message from the ECMP is signed again and the doubly signed message is placed into the Voted Message Queue (VMQ). A failed comparison will cause the replica to terminate itself and hence the process pair to stop producing doubly signed output messages.

The final thread which operates within the system is the sender (Send). This thread picks up messages from the VMQ and dispatches them to their destinations.

The receive thread (Recv) of the follower also accepts only new and authentic double signed messages for processing but performs a different task to that executed by the receive thread of the leader. As each message is accepted, the follower checks the contents of its Internal Received Message Pool (IRMP). If the message under consideration is already within the pool, then the pair of messages is deleted. If the message is not within the pool then the receiver stores the message in the IRMP and associates a time-out t_1 with it. If the message is not received from the leader before the time-out expires, the follower passes that message over to the leader for ordering by placing it in TXQ (the RX thread of the leader, upon encountering a double signed message, passes it to the Recv thread, see figure 4). The message within the IRMP is then given a new time-out t_2. If this second time-out expires and the message has not be received from the leader as an ordered message, then the follower assumes that the leader has failed and shuts down its comparator thread and terminates. This arrangement ensures that even if a correctly functioning leader misses receiving a valid message for processing but the follower does receive that message then the message nevertheless gets ordered and processed by the pair.

Ideally, the value of t_1 should be set to the sum of the estimated maximum message reception skew between a process pair and the estimated maximum communication delay time between a process pair. In the current implementation, a very simple solution has been adopted: the values has been set to zero. In this manner, upon receipt of a message the follower immediately feeds it back to the leader for ordering. The value of t_2 should be set to be greater than twice the estimated maximum communication delay time between a process pair.

2.4. Message overheads

To examine the message passing required to perform a calculation, the example of a null RPC call is used. The configuration shown in Figure 5, has a fail silent client process pair communicating via RPC with a fail silent server process pair. Here a single byte message is passed from the client to the server, which echoes this value back as its result. In the case of an RPC a total of two messages are exchanged. The enumeration of the messages passed as a result of the replicated Ping-Pong operation is done with reference to Figure 5 below.

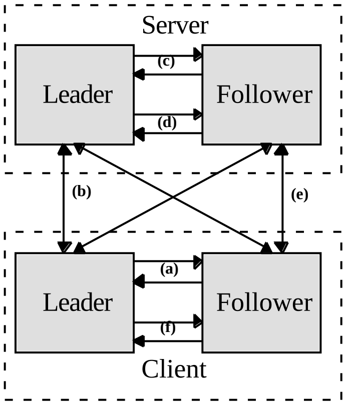

Figure 5: Message overheads

In phase (*a*) of the process the two client replicas form output messages and exchange them for the purpose of comparison. In phase (*b*) the doubly-signed output messages are then transmitted to the server pair, each replica passing one message on each of its two redundant connections (a total of four messages). In phase (*c*) the server leader orders the message and passes it to the server follower and the follower exchanges a feedback copy of its input message (two message transmissions). In phase (*d*) the server replicas form a reply message and exchange them for comparison, these two messages are then returned to the client pair on the replicated links in phase (*e*). The final phase of the Ping-Pong operation (*f*) is the order/feedback exchange of the received message by the client pair.

This gives a total message count of 16 messages for the phases (*a*) to (*f*), which is eight times the message count of an unreplicated RPC. This message count is large, but careful exploitation of parallelism within the Voltan system leads to latency and throughput results which are far better than a crude estimate might suggest.

3. Treatment of Non-determinism

A Voltan process is written by the user as a single threaded program; replication is achieved in a transparent manner by the system. The use of active replication within any system engenders problems caused by non-determinism. Unchecked, non-determinism within replicas could lead to divergence of states. To achieve transparent replication the Voltan Applications Programming Interface (API) provides a series of system calls giving both general purpose and specific mechanisms for avoiding non-deterministic calculations. The following cases illustrates the two types of non-determinism commonly encountered:

Operating system non-determinism: The use of the UNIX `gettimeofday()` system call within a self checking process pair could lead to state divergence (hence stoppage) since replicas cannot be guaranteed to get the same clock reading. Therefore, as we describe below, the Voltan API provides such known non-deterministic system calls as wrapped functions which perform the same calculation, but in a deterministic manner.

Application level non-determinism: The Voltan API provides mechanisms for the user to handle application level non-determinism in a general manner. For example, a user level calculation of a value based on a sample of analogue real-world input could be handled in this manner ensuring that both the replicas process the same value.

Non-deterministic operations can be either synchronous or asynchronous:

Synchronous result: These are the value-result operations typified by the `gettimeofday()` system call mentioned previously.

Asynchronous result: Such phenomena as timer firings and signal reception fall into this category. Voltan ensures that application registered handlers for asynchronous events are processed at the same execution point in both replicas.

The main idea behind preventing state divergence for non-deterministic calculations is to stop the follower from independently calculating a result value and using it for comparison; rather, the follower relies on the value supplied by the leader. Unfortunately this opens up the possibility of a faulty leader's output remaining undetected. A practical way out of this difficulty is for the follower to perform application specific *reasonableness* check to detect any faulty behaviour of the leader.

3.1. Synchronous result non-determinism

Of the two classes of non-deterministic calculations, the synchronous result variant is the easier to solve.

For the previously mentioned synchronous result non-deterministic calculation, the only requirement is that the application replicas both receive the same value. The synchronous return nature of the call ensures that both replicas will receive that value at the same point within the execution of the application. The outline algorithm for the solution of such a calculation is shown in Code fragment 1. In this example the application has made a call to a wrapped system call `gettimeofday()`this call is provided by Voltan as a member function of the base Voltan application class `VApp`.

```
VApp::gettimeofday(parms)  {
if (role == leader) {
    r = ::get_timeofday(parms);
    Message *m = new(Message);
    *m.pack(parms, r);
    diffuser.followerSend(m);
    return r;
  } else {
    Message *m;
    DMQ.pop(m);
    m.unpack(params, r);
    f = ::get_timeofday(parms);
    if (!rangeCheck(f,r,delta))
       exit(-1);
    delete m;
    return r;
  }
}
```

Code fragment 1: Wrapping a non-deterministic system call

In such cases the calculation is performed only by the leader and the result of the calculation is then forwarded to the follower, which unpacks the result, performs a reasonableness check and if successful returns the leader's result, else terminates itself. The check in the above case consists of the follower also calling `get_timeofday(parms)`and then invoking the `rangeCheck()` function, this function compares the two time values with respect to the maximum calculation and messaging delays.

Known non-deterministic system calls may be wrapped in this manner. However there is still a requirement for a solution to application-generated non-determinism. A replicated Voltan application can have no knowledge of its role as either a leader or a follower. To allow the application access to such information would create more causes of non-deterministic operation and break the concept of replication transparency.

Voltan provides a mechanism whereby the application can hand over the calculation of a non-deterministic value to the Voltan system which can safely determine the role of the process and proceed as before. The mechanism is provided via a base class `NDC` (non-deterministic calculation). This class is shown below (minus constructors etc.) in Code fragment 2:

```
class NDC
{
 void doAndPack(Message &m)=0;
void rangeCheck(Message &m)=0;
Message calculate(void);
};

Message NDC::calculate(void) {
  if (role == leader) {
    Message *m = new Message;
    doAndPack(*m);
    Message *n= new Message(m);
    diffuser.followerSend(n);
    return m;
  } else {
    Message *m;
    DMQ.pop(m);
    if (!rangeCheck(*m))
      exit(-1);
    return m;
  }
}
```

Code fragment 2: A base class for application level non-determinism

The class definition contains three member functions:

`doAndPack(..)`: A pure virtual function which is supplied by the derived class. It contains the code to perform the calculation and pack the result.

`rangeCheck(..)`:A pure virtual function which is supplied by the derived class. It contains the code used on the follower side to apply a reasonableness test upon the supplied data.

`calculate(..)`: This function co-ordinates the actions of the replica dependent on its assigned role, in a manner transparent to the application.

As an example of the use of this class, consider Code fragment 3. Here a Voltan process is required to access a real-time counter through an input port. The mechanisms involved allow the transparent replication of such calculations.

```
class Sensor: public NDC
{
 void doAndPack(Message &m);
 void rangeCheck(Message &m)=0;
public:
 long int read(void);
};
 void Sensor::doAndPack(Message
&m)
{
  long int res = inVal(port);
  m.pack(r);
}

void Sensor::rangeCheck(Message
&m)
{
  long int res = m.unpack();
  return ((res > counterMin) &&
         (res < counterMax));

}

long int Sensor::read(void)
{
  Message m;
  long int res;

  m = calculate();
  m.unpack(res);
  return res;
}
```

Code fragment 3: An example class implementation

3.2. Asynchronous result non-determinism

The requirement here is that both the replicas must receive any asynchronous signal (e.g., a time-out exception) at exactly the same point within the execution of their respective computation.

The approach taken for dealing with time-based non-determinism is to convert any asynchronous event into a message and to ensure that the replicas both select the same message at the same point during their execution. An asynchronous event would more likely than not be generated by an operating system entity which is not part of the 'Voltan world'. We make use of special 'fan-out' and 'fan-in' objects for interfacing with non-Voltan entities (see the next sub-section).

Consider a specific example. Assume that application programs have the facility of setting a timer object (a non-Voltan entity) to receive a time-out exception after the specified interval. In the Voltan-version, a call on the function 'set-timer' will produce an output message which after comparison is dispatched to a fan-out object. This object will thus receive double signed message for setting the timer and not be susceptible to a faulty leader arbitrarily setting time-outs. Logically the fan-out object communicates with a timer process which exists outside the Voltan world. When the timer process detects the timer expiry it dispatches two messages to a fan-in object which vote the message and pass the result back to the Voltan application, this again guarantees that no replica will receive a time-out message unless both do. This arrangement is shown in Figure 6.

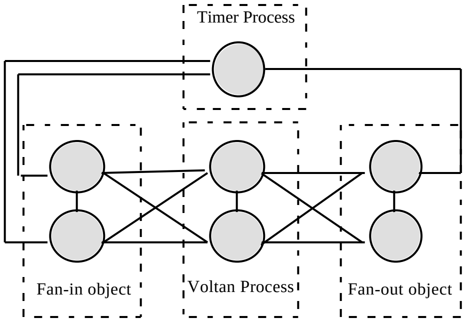

Figure 6: Treatment of asynchronous signals.

It is to be noted that the above is a logical representation of the system. In practice the timer process would be executed by the leader of the fan-out object.

Synchronisation of event message receptions between the leader and the follower is straightforward. Since it is the leader which orders all messages and the follower accepts that order, then if the leader detects an event message it merely passes it over to the follower. This guarantees message ordering.

It is considered that an outstanding event is of a higher priority than any incoming message from another service or application. To this extent event messages are placed at the head of the queue (DMQ) for processing. When accessing its input queue for the next application message, the leader loops through all outstanding event messages, dispatching them to the follower, then performing the event action, before returning any message which matches its input criteria. The follower keeps accepting messages from the leader, executing event handlers if necessary and returning a message only on receipt of a non-event message.

The above scheme ensures that both replicas select the same event message at the same point in their execution. However, for this scheme to be effective, it is necessary that application threads frequently execute message reception/dispatch calls. There are two cases where this condition is difficult to meet:

Computationally intensive applications: An application which spends a large amount of time calculating results in relation to the number of send or receive calls it makes will not be responsive to event messages.

Idle servers: When a server has had no connections made to it, or has had all of its connections removed, there will be no messaging calls made. However Voltan uses events to trigger the up-calls indicating a new connection has occurred. This would mean that in the absence of messaging calls no up-calls would be triggered and so no indication of a new connection could be made.

To solve these two problems there are two API calls which deal with events, shown below.

API calls for event handling
void await(void);
void yield(void);

await(): This primitive waits until an asynchronous event occurs. In the case of an idle server it can be used to suspend operation until a new connection is established.

yield(): A call to yield can be inserted in the application code to enable frequent inspection of input message queue and trigger all outstanding events. If no events are outstanding the leader has to synchronise with the follower by dispatching a no event message. This extra message overhead means that the granularity of calls to `yield()` must be considered carefully.

3.3. Interfacing with the outside world

Fail-silent processes cannot be expected to deal solely with other fail-silent processes. In the light of the performance penalties expected for the deployment of fail-silence, it can only be assumed that fail-silence will be deployed within a system only where it is necessary. A system of Voltan processes (Voltan world) interacts with the outside world through two objects, the fan-in object and the fan-out object mentioned earlier. The fan-in and fan-out objects themselves have been designed not to require asynchronous event processing.

A fan-in object accepts messages from the outside world votes on them to create a doubly signed message and passes them on into the Voltan world. In essence a fan-in object operates a special form of the Voltan input ordering protocol. Instead of using leader ordering with follower feedback, the input algorithm requires both leader and follower to receive an input message before proceeding to vote and output a message into the Voltan world.

A fan-out object strips a Voltan message of all of its signatures etc., and passes the output message on into the outside world. This is shown in Figure 7.

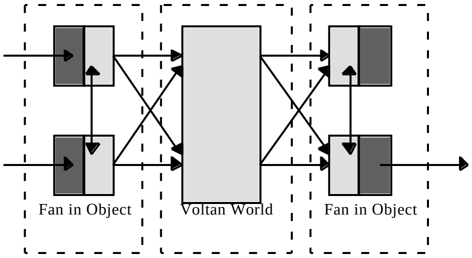

Figure 7: Interacting with the outside world

4. System Support for Distributed Applications

A Voltan application is comprised of a number of named Voltan processes termed *services,* which communicate through message passing. A given application is constructed by using services in an application specific manner. It is necessary therefore for an application to be able to connect (bind) to named services. In distributed systems, it is common to provide a binding agent - a connection manager - for this purpose.

4.1. Nizam Connection Manager

To aid connection between and to services, a third party agent is used. The Voltan system server *Nizam*, facilitates such inter-service connections[1].

[1] Nizam al-Mulk, was the vizier of the Seljuk sultan, Voltan Alp Arslan, responsible for the introduction of an organised system of

Part of the Voltan API consists of primitives for the creation of services and the establishment of connections between them. As an overview consider the client-server system shown in Figure 8.

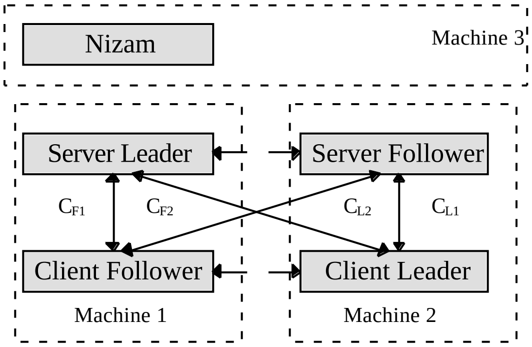

Figure 8: A typical Voltan system

The system consists of two application services named ‘Client’ and ‘Server’ and the Voltan system server Nizam. Logically there is a configuration manager which places the replica parts onto specific machines, however currently that operation is performed manually.

The Voltan system server Nizam provides the following services:

- assigns roles to process replicas (leader or follower.)
- assigns Unique Identifiers to both process pairs and to individual replicas.
- allows service replicas to synchronise at start up and negotiate the intra-service connection.
- holds a registry of active services and their locations. Allowing connecting services to negotiate inter-service linkages.

When a new service is created the two replicas are executed. The initialisation code within Voltan contacts Nizam and awaits an indication of the role which that replica is to perform. A designated leader will publish a port for intra-nodal communications by passing TCP port address and host details across to its partner process (via Nizam). A designated follower process will await the published port information from Nizam and subscribe via a TCP `connect`() call. Once both service parts are established and initialised they indicate this fact to Nizam which places the finalised connection information into its registry of active services.

Once a service is registered with Nizam it can begin to process information, form new connections and react to services connecting to it.

When a connection is established from a source service (the client in Figure 8) to a destination service (the server in Figure 8), the source service is considered to be performing in the *active mode*. In such cases the Voltan call to create the link is fully synchronous, that is it returns only after a connection is established.

For the Voltan process which is being attached to (the server in Figure 8), the mechanisms involved are different. In such a case the connection occurs asynchronously. The application is informed of the presence of a new connection by the triggering of an application registered up-call. This method of operation is categorised as *upcall* mode connection processing.

In the client server model shown in Figure 8 the client is in the active mode and the server the upcall. Modes can of course be mixed with a given application both responding to and initiating numerous connections.

4.2. Connection management

API calls are provided for initiating and reacting to new connections. For the sake of exposition these calls are divided into groups by their nature (active or upcall). Upcall calls are handled by user supplied call-back functions. Active calls are normally synchronous invocations which pass into the Voltan kernel and return upon a successful result or failure.

When a connection is established the unique process ID for the communications end-point generated by Nizam is passed to the Voltan process by the Voltan kernel. All message dispatch and reception primitives accept these Single Process ID values (SpidType) as source or destination values for messages. For reception primitives which can receive from multiple services the parameter is a set of SPID’s (SpidSet).

Upcall connection management calls
`void newLink(SpidType from);`

administration.

```
void disconnected(SpidType by);
```

The primitive `newLink` above indicates to the application that another service has established a connection with it. When the user supplied call-back function is triggered the parameter from indicated the service which initiated the connection.

The up-call `disconnected` indicates to the application that the connection designated by the given SPID has been closed.

Active connection management calls

```
SpidType linkTo(String to,
TimeOut time=∞);
void disconnectFrom(SpidType
from);
```

The primitive `linkTo` establishes a connection with the named service `to`, in the time-out period `time` which defaults to an infinite value. The return value is the SPID for the destination service.

The primitive `disconnectFrom` severs the connection with the service specified by the given SPID.

Connection establishment protocol

When an active service establishes a new connection with an upcall service, there are two requirements:

(i) The establishment of the communications links
(ii) The triggering of the newLink up-call for the passive service.

The non-deterministic mechanisms for synchronous event triggering needed in the second case are described in section 3. This section will deal only with the connection establishment phase (case 1).

Fig. 8 shows a client operating in an active manner is connecting to a server which will be performing in the upcall mode. There are two connections to be established for each replica (C_{F1} and C_{F2} in the case for the follower.)

The algorithm is executed by all replicas of the two Voltan processes involved in the connection, and proceeds as follows

1. The active role connector requests Nizam for connection to the named service with a specified time-out (possibly infinite).
2. If the requested service is not already registered with Nizam and fails to register within the required time period, Nizam returns a time-out value to the active service which attempted to create the connection. The connection primitive at the active end then returns a time-out value as a result of the `linkTo`() primitive.
3. If steps one and two above have been successfully completed Nizam passes host information for each side of link to the corresponding replica. The active initiator of the connection publishes details of a TCP port for the upcall service process, via Nizam. The upcall replicant subscribes to this port. An ‘ack’ message is sent to Nizam by both parties indicating the success of the connection.
4. Stage 3 is repeated for the second connection to the other replica.
5. When all of the connections are established (C_{L1}, C_{L2}, C_{F1} and C_{F2},) Nizam will have received four acknowledgements from stages 3 and 4. At this point Nizam sends a message to all four replicas indicating that the whole connection scheme is complete and that communications may begin.
6. The connection handling threads register the file descriptors for the new connections with the Voltan kernel so message dispatch and reception may occur.
7. The active connector indicates the SPID of the new service to the Voltan application via the return value from the synchronous `linkTo()` command. The passive connector triggers the up-call `newLink()`.

In the connection scheme described above it is the active role connector which publishes the ports and the upcall role conectee which performs the subscription. This method of publishing and subscribing is necessitated by the provision of time-outs. Such a mechanism allows the active role process to abort the connections at any point when a time-out occurs. The upcall role process in the presence of dropped connection will perform a clean up operation, and the `newLink` up-call will never be triggered.

The current system as described here can lead to deadlocks due to the multi-threaded nature of the Voltan kernel. If a single system thread where to be used to handle both upcall and active connection requests, then deadlock may occur, this is shown in Figure 8.

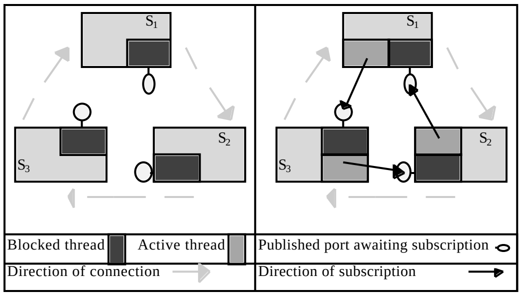

Figure 9: Connection management Agents

Figure 9(1) shows a situation where three services have initiated connections as follows

$$S_1 \rightarrow S_2$$
$$S_2 \rightarrow S_3$$
$$S_3 \rightarrow S_1$$

at the point of deadlock S_1 has published a port and is awaiting subscription by S_2. Similarly S_2 is awaiting on S_3 and S_3 upon S_1. No possible progress may now be made and the system deadlocks.

To combat this problem two threads are used in connection establishment. These are the `OriginateAgent`, which handles active connections and the `ResponceAgent` which deals with upcall connection details. This is shown in Figure 9(2).

In this system the OriginateAgent awaits subscription and the ResponceAgent initiates it. This arrangement plus the synchronous nature of the active connection mechanism allows a connection model where: A Voltan application may only initiate one connection at a time, but may respond to many simultaneous requests and will not deadlock.

4.3. Message dispatch and reception

The basic unit of information inter-change within Voltan is the `Message`. A `Message` is a packed buffer. The Voltan API contains one primitive for the dispatch of a message and a number of calls to handle the reception of a message. However the message reception calls may be generalised to a single function presented here which, with suitably overloaded parameters can present many different manners of behaviour [9,10]. These two primitives are:

Message dispatch and reception primitives

```
void
sendTo(SpidType to, Message m);
```

```
SpidType
generalRecv(SpidSet from,
            Message m,
            TimeOut t = ∞,
            Bool prio = false);
```

The API call `sendTo` accepts as its parameters a SPID to dispatch the message to, and a message to dispatch.

The sendTo primitive simply marks the message with the destination and enqueues it into the application’s output message queue (ICMP, see fig. 4). The Voltan system then takes care of the dispatch of that message to the destination upon successful conclusion of the comparison process.

The primitive `generalRecv` returns the first message from one of the services held in the set of SPID’s `from,` within the parameter `m`. An optional time-out `t` may be associated with the call, if no message arrives within the given time frame a null message is returned as the result. The final parameter `prio` determines if the system accepts messages in priority order. If priority is enabled the selected message will be the highest (user-supplied) priority message from any of the specified sources. In cases where multiple suitable messages are present in the input queue, the algorithm operates a fair-share policy so no source service should be starved of treatment. In all cases the function returns the SPID of the source service, or a time-out indicator.

The primitive `generalRecv` works as follows: the leader selects the next message from the input queue (DMQ) according to the specified criteria. The selected message is then passed over to the follower as the next ordered message. `generalRecv` on the follower simply becomes a matter of popping the first message off the input queue and returning that. In the case of a time-out the leader sends a null message across to the follower.

5. System Performance

5.1. Implementation

We have produced implementations of Voltan failure masking as well as fail-silent *nodes* using T800 transputers [4,5,6]. The C++ class library developed for these experiments has been reused and developed further to create the system software for Voltan fail-silent *processes*. The current platform consists of two HP-747 workstations with VME backplanes, each containing two HP-742 processors, each running the real time UNIX operating system HP-RT (see fig. 10). The Voltan system software runs on HP-742 processors.

The processors are configured to use the backplane as a second channel for communications; this is used for intra-nodal traffic. The Ethernet is used for inter-nodal messaging. A client-server system was configured (see fig. 10) for measuring the performance of the system.

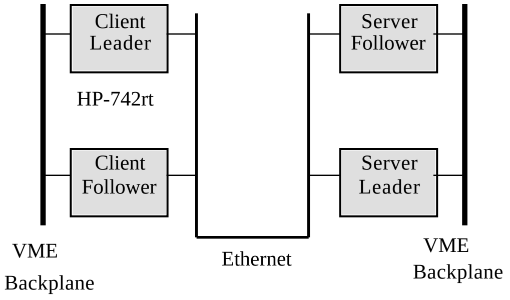

Figure 10: Hardware configuration

5.2. Performance Figures

The results presented in this section detail the system performance in three major areas:

1. Connection establishment
2. Overheads for message passing
3. Overheads for controlling non-determinism.

For connection management, the metric taken was the time between the initiation of a connection with a server and the time to the reception of the first reply. Two tests were performed to assess the overhead of message passing. In the first case a latency measure was taken. Here the round trip times for a Ping-Pong operation was measured (using messages of varying sizes). In the second case, throughput was quantified; the test consisted of sending a large number of messages before beginning to accept the replies. This way we measured the maximum throughput: the maximum rate a server process can order and compare messages. The final test category was used to gauge the penalty involved in the mechanisms for dealing with non-determinism. We measured the maximum number of yields per second that the follower could perform, as well as average delay times in the triggering of asynchronous events at the follower.

When describing the throughput and latency of the system the message size indicated is the payload. The actual message sent has a 38 byte overhead added to it, which contains such information as signatures and process identifiers.

Connection establishment

The delay experienced between a request for a connection to a service and the point at which the `connectTo()` primitive completes averages at 1.5 seconds. This figure reflects the complexities of binding two replicas in the presence of the required multiplexed connection model. In many fault tolerant applications it is realistic to assume that communication between fail-silent process pairs will be long lived. In the light of this observation, connection times are not deemed slow.

Message passing overheads

The latency figures for a null RPC are shown below in Figure 11.

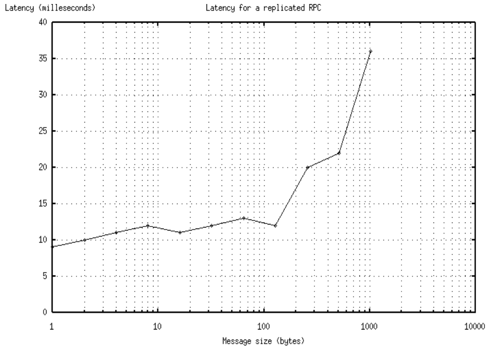

Figure 11: Latency for a replicated RPC

For messages of 100 bytes, the average send/receive latency is approximately 15ms. This is an order of magnitude slower than a non fail-silent RPC, but it compares favourably with RPC latencies achieved a few years ago.

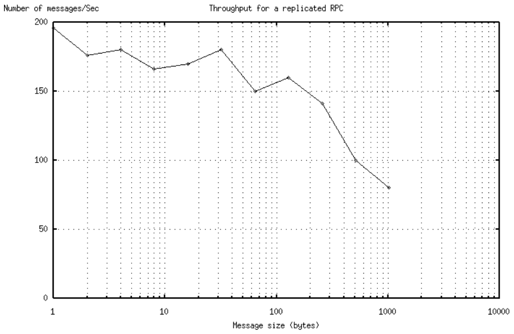

Figure 12: Throughput for a replicated RPC

The throughput figures for a server is shown in Figure 12. A Voltan fail-silent process has a throughput of 138 messages per second for 100 byte messages.

The throughput and latency figures are certainly encouraging, if it is recalled that fail-silence behaviour despite (authentication detectable) Byzantine failures is being achieved, which requires sixteen messages per RPC (see fig. 5), and the entire system is composed of ‘off the shelf hardware/software components’.

<u>Overheads for controlling non-determinism</u>

Voltan uses messages passed from the leader to the follower to synchronise actions in the event of a non-deterministic choice being required. The size of such messages varies from 1 byte in the case of an indication that a yield was not taken to 16 bytes in the case of a `gettimeofday()` system call (this is in addition to the 38 bytes which are used as a header in all Voltan messages).

Measurements were taken to determine how many yield() calls per second may be performed when the system was under no load. The average was over 1000 yield calls per second. However when application processes are doing useful work, inserting frequent calls to yield is likely to be counter-productive. As such the granularity of calls to yield must be carefully considered, one such call in approximately 50ms is considered a reasonable compromise.

A second measurement taken was that of non-deterministic event latency. This measured the time delay between the triggering of an event on the leader and its activation on the follower. The lower bound on this value was approximately 1ms. The upper bound on this value is dependent upon the frequency of calls to yield or to the messaging functions. In practice as stated earlier this should be of the order of 50ms.

6. Related Work

A fail-silent node that uses replicated processing with comparison/voting must incorporate mechanisms to keep its replicas synchronised, so as to avoid the states of the replicas from diverging. Synchronisation at the level of processor micro-instructions is logically the most straightforward way to achieve replica synchronism. However, as we mentioned in the Introduction, such hardware-based designs are proving increasingly difficult to implement. Hence there has been much interest in developing software-implemented nodes.

The task/process level synchronisation approach used in Voltan was pioneered by the designers of the SIFT failure-masking node [7]. In SIFT, application processes are structured as a set of co-operative cyclic tasks. Each task performs a deterministic computation. The execution of a particular iteration of a task consists of inputting some data (possibly generated by previous iteration of other tasks), processing the data, and outputting some results. Fault-tolerance is achieved by voting on the input data. Thus, task replicas must be synchronised at the beginning of each iteration (start of a *frame*). To achieve this, SIFT maintains a global timebase, and uses a static, priority based scheduling, which schedules tasks at pre-defined time frames. The global timebase is implemented by keeping the clocks of all the correct processors synchronised by a software implementation of a Byzantine resilient clock synchronisation protocol. In normal operation, the system only allows interruptions from clocks, which are handled by all correct processors at the beginning of the same time frame. Because of its application dependent design, the SIFT architecture can only be applied to a restricted range of applications. This is also the case for the VOTRICS system [12] which follows the design principles of SIFT to provide fault-tolerance in a different, but still specific, class of applications (railway signalling systems).

There has always been a concern over the performance of software-implemented nodes due to the overheads imposed by redundancy management protocols. In SIFT for instance, redundancy management protocols can consume as much as 80% of the processor throughput [13]. Hybrid solutions have been proposed to circumvent this problem. MAFT [14], FTP-AP [15], and Delta-4 [11] are hybrid architectures that share the same basic design. These architectures are structured around a micro-instruction synchronised hard core, on top of which conventional processors are replicated. The micro-instruction synchronised hard core is responsible for executing redundancy management functions (e.g., message voting). This certainly improves the performance; however, the hard core re-introduces the problems associated with the hardware-implemented nodes.

In our work, we have taken the SIFT approach further by investigating the design of a family of failure-masking and fail-silent nodes, Voltan, that are capable of supporting quite general purpose message passing programs. We have implemented several replica synchronisation schemes and evaluated their performance [6]. The leader-follower design described here probably indicates the limits of what can be achieved using standard 'off-the-shelf' processors.

Approaches that do not use processor replication but rely instead on various *application specific* forms of checking mechanisms (e.g., watchdog timers) for detecting the erroneous behaviour of a processor have been considered [e.g., 16]. The error detection coverage of one such node has been estimated to be better than 99% [17]. However, these approaches are application specific (rather than general purpose).

The primary-backup approach used in the ‘hypervisor’ fault-tolerant node [18] is in many ways similar to the leader-follower approach used by us. There is an important difference though. The hypervisor fault-tolerant node design assumes that the underlying processors will suffer only crash failures, so the second processor (backup) is used for taking over the role of the failed primary. We do not make such a failure assumption, but instead use leader-follower approach to *implement* crash failure semantics. The performance overheads reported for a hypervisor node are comparable to what has been reported here.

7. Conclusions

The Voltan system software described in this paper provides a realistic way of constructing distributed applications using self-checking process pairs. One of the major difficulties in using the active replication for the provision of fault tolerance is the dependency on strictly deterministic calculations. In this paper we have presented practical solutions to the problems of treating non-determinism.

Effective parallelism within a Voltan process-pair, achieved by multi-threading, results in request/response latencies for client-server RPC’s which compare favourably with the performance of standard RPC packages of a few years ago. Poor performance is commonly given as the main reason for not deploying software-implemented fail-silent approaches. Indeed, for a given processor type, hardware-implemented nodes will always out perform their software equivalents. Closer scrutiny of this argument however would indicate that this is not a fair comparison.

Hardware-based replication is one of the most difficult technologies to master, especially as processor busses become wider, and clock rates become faster. Further, every new microprocessor architecture requires considerable re-design effort. As lead-times for the deployment of hardware fail-silence increase, such implementations are beginning to suffer performance penalties through the utilisation of processor technology which is of an older generation. The current rapid increase in modern processor speeds means that this situation can only be exacerbated. Thus the software approach of Voltan offers long term solutions for exploiting faster processors and network technologies with minimum lead time for porting the system software.

Acknowledgements

This work has been supported in part by Hewlett-Packard Research Laboratories, Bristol and the UK Engineering and Physical sciences Research Council. Acknowledgement is due to Francisco Brasileiro, Roger Fleming, Paul Harry, David Ingham , Graham Parrington and Andrew Thomas for their help in the work reported here.

References

[1] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit", IEEE Transaction on Computers, Vol. 41(5), pp. 526-531, May 1992.

[2] K. Birman, "The process group approach to reliable computing", CACM , 36, 12, pp. 37-53, December 1993.

[3] P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", IEEE Computer, Vol. 21 (2), pp. 37-45, February 1988.

[4] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, "Principal Features of the Voltan Family of Reliable Node Architectures for Distributed Systems", IEEE Transactions on Computers, (Special Issue on Fault-Tolerant Computing), 41 (5), pp.542-549, May 1992.

[5] N.A. Speirs, S. Tao, F.V. Brasileiro, P.D. Ezhilchelvan and S.K. Shrivastava, "The Design and Implementation of VOLTAN Fault-Tolerant Nodes for Distributed Systems", Transputer Communications, Vol. 1 (2), pp. 1-17, November 1993.

[6] F. Brasileiro, P D Ezhilchelvan, S. K. Shrivastava, N. Speirs and S. Tao, "Implementing fail-silent nodes for distributed systems", I.E.E.E. Transactions on Computers, 45(11), pp. 1226-1238, November 1996.

[7] J.H. Wensley et al, "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control", Proceedings of IEEE, Vol. 66 (10), pp.1240-1255, October 1978.

[8] F. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial", ACM Computing Surveys, 22 (4), pp. 299-319, December 1990.

[9] A. Tully and S.K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, pp. 104-113, October 1990.

[10] S.K. Shrivastava and A,. Tully, "Active replication of non-deterministic programs", Technical report number 418, Depat. of Computing Science, University of Newcastle upon Tyne, March 1993.

[11] D. Powell (ed.), "DELTA-4: A generic archetecture for dependable distributed systems", Springer-Verlag, October 1991.

[12] N. Theuretzbacher, "'VOTRICS': Voting Triple Modular Computing System," Digest of Papers, FTCS-16, Vienna, Austria, pp. 144-150, July 1986.

[13] D.L. Palumbo, and R.W. Butler "Measurements of SIFT Operating System Overhead," NASA Tech. Memo. 86322, 1985.

[14] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," IEEE Transactions on Computers, Vol. 37(4), pp. 398-405, April 1988.

[15] J.H. Lala, and L.S. Alger, "Hardware and Software Fault Tolerance: A Unified Architectural Approach," Digest of Papers, FTCS-18, Tokyo, Japan, June 1988, pp. 240-245.

[16] J. Reisinger, and A. Steininger, "The Design of a Fail-Silent Processing Node for the Predictable Hard Real-Time System MARS," Distributed System Engineering Journal, Vol. 1(2), pp. 104-111, 1993.

[17] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," Digest of Papers, FTCS-20, Newcastle upon Tyne, pp. 466-473, June 1990.

[18] T. C. Bressoud and F. B. Schneider, "Hypervisor-based Fault-tolerance", ACM Trans. on Computer Systems, 14(1), PP. 80-107, 1996.