Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes

To cite this article: S Tao et al 1995 Distrib. Syst. Engng. 2 39

View the article online for updates and enhancements.

Related content
- The Voltan application programming environment for fail-silent processes
  D Black, C Low and S K Shrivastava
- The design of a fail-silent processing node for the predictable hard real-time system MARS
  J Reisinger and A Steininger
- Towards an integrated approach to fault tolerance in Delta-4
  P A Barrett and N A Speirs

Recent citations
- F.V. Brasileiro et al
Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes

S Tao†, P D Ezhichelvan and S K Shrivastava

Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK

Abstract. One way of gaining confidence in the adequacy of fault tolerance mechanisms of a system is to test the system by injecting faults and see how the system performs under faulty conditions. This paper presents an application of the focused fault injection method that has been developed for testing software implemented fault tolerance mechanisms of distributed systems. The method exploits the object oriented approach of software implementation to support the injection of specific classes of faults. With the focused fault injection method, the system tester is able to inject specific classes of faults (including malicious ones) such that the fault tolerance mechanisms of a target system can be tested adequately. The method has been applied to test the design and implementation of voting, clock synchronization, and ordering modules of the Voltan TMR (triple modular redundant) node. The tests performed uncovered three flaws in the system software.

1. Introduction

Fault injection techniques have long been recognised as a useful way of testing the adequacy of fault tolerance mechanisms [3, 11, 12], examining coverage of error detection schemes [1, 10, 15, 27], and studying system behaviour under faulty conditions [4, 6, 7]. The actual injection methods differ according to the requirements and the nature of the target system. In this paper we report on our experience of applying a fault injection method which we have developed to uncover any deficiencies and flaws in the design and implementation of the system software for Voltan TMR (triple modular redundant) nodes [30, 32].

Our method is basically intended for testing software implemented fault tolerance mechanisms of distributed systems. It requires that the target software be structured in a modular fashion of objects interacting via messages so that messages can be manipulated to emulate incorrect behaviour of faulty processors. The system software for Voltan TMR nodes has the required structure. In our fault injection experiments, one processor is subject to fault injection while the software running on the other two correct processors is tested. Faults were introduced by changing the contents of a message produced by the injected processor, changing the arrival time of the message, or deleting the message itself. It will be shown that a wide range of faulty behaviour can be emulated in this way. Although our fault injection method has so far been applied only to test the Voltan software, we believe the principles behind the method can easily be applied to other object oriented distributed systems.

A review of related work is presented in section 2. Section 3 describes the focused fault injection method. Section 4 gives a brief description of the Voltan TMR node on which the fault injection experiments were conducted. Section 5 describes the software for focused fault injection as used for the Voltan TMR node. The fault injection experiments and results are presented in section 6. Section 7 draws some conclusions from the work.

2. Review of related work

Fault injection based experiments can be employed to achieve two separate objectives regarding the validation of fault tolerant computing systems: fault forecasting and fault removal [2]. These two different objectives require quite different fault injection techniques.

In fault forecasting, experiments are performed to rate the effectiveness of various dependability mechanisms (such as error detection mechanisms) or to study system behaviour under faulty conditions. In these experiments, faults of random nature, which are believed to represent real faults, are injected into the target system. The results of the experiments are then used in dependability analysis of a given fault tolerant system. Most of the existing fault injection tools and methods, implemented either in hardware [1, 10, 15, 27] or software [19, 25, 28], are essentially intended for fault forecasting.

Achieving the fault removal objective using fault injection involves attempts to eliminate the presence of any...
faults (known as \textit{fault tolerance deficiency faults} [2]) in the design and/or implementation of the target fault tolerant system. This requires the injection of specific classes of faults so that the target system can be adequately tested. There are two different approaches to testing: \textit{structural testing} and \textit{functional testing}. In structural testing, faults are injected in a manner that enables selected structural parts of the target system to be exercised. While in functional testing, specific fault scenarios (such as a ‘two-faced general’ [20]) are created to ascertain that the target system can indeed tolerate such faulty conditions. These two testing approaches are complementary to each other.

In distributed systems where processors communicate with one another through message exchanges, messages provide a natural and convenient way of injecting faults into the system. Using message manipulation for fault injection testing for fault removal has been examined in [11, 12]. Echtle \textit{et al.} [12] have investigated structural testing of fault tolerant protocols/algorithms by injecting specific classes of faults (also called \textit{deterministic fault injection} in [12]). A testbed system [13] was used to support the manipulation of messages, so various fault scenarios can be emulated. Work on structural testing is also reported in [3]. An application specific approach was taken in fault tolerance testing of the Advanced Automation System (AAS) [11], where fault injection provisions were integrated into the AAS architecture by the developers of the system to allow testers to conduct fault tolerance testing. The functional testing approach was adopted in the experiments.

The focused fault injection method that we have used to test the Voltan software supports the injection of specific classes of faults at specially selected points within the target software system. It allows the testing of implemented target systems, not just the testing of protocols/algorithms implemented in a testbed environment. The only requirement for the application of the method is that the target system be structured as objects communicating with one another through message exchanges. No other provisions for fault injection are required in the target system. It thus separates the task of system testing from system development.

3. \textbf{Focused fault injection method}

The Voltan software implements fault tolerant strategies for masking processor failures. Therefore, our aim is to find a means of emulating the behaviour of faulty processors. In a distributed system where processors interact by message passing, the failure of a processor will be exhibited by its external behaviour which is entirely represented by the messages the processor sends (or fails to send). Thus the failure of a processor can be emulated by having the processor send erroneous messages. There is no need to be concerned about the internal conditions of the failed processor. In this section, we first present a classification of faults and then describe how the corresponding faulty behaviours can be emulated using our fault injection method.

3.1. \textbf{Modelling faulty behaviour}

Processor faults can be classified into \textit{omission} fault, \textit{value} fault, \textit{timing} fault and \textit{arbitrary} fault [14, 31]. An omission fault causes the expected response not to be produced at all by the processor. A value fault causes the response to be produced within the specified time frame but with its contents corrupted. A timing fault causes the response with correct contents to be produced outside the specified time frame, either early or late. An arbitrary fault causes any violation from the specified behaviour in terms of timing and/or value.

An arbitrary fault subsumes all the other three classes of faults. The relationships among these four fault classes can be expressed by the fault lattice (see figure 1(a)), where an arrow from A to B, A→B, indicates that fault class A is a special case of fault class B. An omission fault can be treated as either a (very) late timing fault or a value fault causing no value to be produced (a special case of corrupted value).

In distributed fault tolerant systems, replicated processing is often employed to increase system dependability. In such systems, a processor is often required to generate replicated responses for a given input. The fault model outlined in figure 1(a) can be extended to deal with the various ways a replicated response may differ from the correct one [14, 31]. A correct replicated response will be the one in which all individual responses have correct (identical) values and are produced in the required time frame. An incorrect replicated response can take many forms. One specific case can be defined as a \textit{consistent} fault which causes the individual responses of a replicated response to violate the specified behaviour in an identical way, such as having identical but wrong values (\textit{consistent value} fault), while with the general case of a value fault, the individual response values could be wrong and need not be identical, or only some values could be wrong and others correct. In a similar manner, a \textit{consistent omission} fault and a \textit{consistent timing} fault can be seen as special cases of an omission fault and a timing fault respectively. We use the fault lattice in figure 1(b) to summarize the relationship among the extended fault classes [14, 31]. Our classification is a generalization of the hierarchies of fault classes proposed by other researchers (e.g. Cristian [9], Powell [23]).

3.2. \textbf{Software structures supporting focused fault injection}

The software within a processor can be structured out of a collection of \textit{active objects} representing processes, which communicate with one another by exchanging messages.
for example, a processor with a number of processes to centralize the coordination work involved. Consider, the collection of processes running on it and generating output behaving like a 'two-faced general' messages. The key is to use a single injection object to inject faults into the faulty behaviour of a processor which has an arbitrary output queue (Q). This structure represents a functional process while L represents a link handling (output) process which actually sends outgoing messages down a physical link of the processor hosting P and L. Thus, faulty behaviour of this processor can be emulated by modifying the messages being output by L. P puts its output messages on FQ. FO picks up messages from FQ, does the fault injection work by modifying the messages, and puts the output messages on Q which is used by L as its input queue. In normal operation mode, P is started with Q as one of its parameters; but in fault injection mode, P is started with FQ instead of Q as one of its parameters. The injection object is started with FQ and Q as its parameters. The object L is unchanged. Various classes of faults can be injected by the fault injection object. The net effect is that the processor hosting P and L produces erroneous messages. Thus we can tamper with messages produced by specific processes within a processor, so as to be able to create the required fault scenarios.

An omission fault in P can be injected by having the injection object delete a message. A value fault in P can be injected by having the injection object change the content of a message. A late timing fault in P can be injected if the injection object holds the message for a period of time before depositing the message in its output queue (Q). Unfortunately there is no equivalent way of injecting a timing fault of early message arrival, though it is possible to achieve this in a target-system-dependent way. For example, an early arrival fault in the Voltan ordering module can be emulated by changing the value of the timestamp of a received message. The injection of an arbitrary fault can be done either by the injection object injecting both timing and value faults, or by having two pipelined injection objects, one injecting a timing fault and the other injecting a value fault.

The approach outlined above can be extended to the case of a processor producing replicated outputs (see figure 3). The injection object FO with multiple input and output queues has full control over faults injected. This structure is particularly important when the effects of a processor behaving like a 'two-faced general' [20] are to be tested.

Fault injection objects can also be used for emulating the faulty behaviour of a processor which has an arbitrary collection of processes running on it and generating output messages. The key is to use a single injection object to centralize the coordination work involved. Consider, for example, a processor with a number of processes

4. Voltan TMR node

A Voltan TMR node is a reliable node constructed out of three interconnected ordinary processors that execute appropriate redundancy management protocols to mask the failure of a single processor. The three processors of a node are self-contained with their own memory and communication links; the boundaries between processors are clearly defined and processors communicate with each other by message exchanges only.

In this section we briefly introduce the architecture and implementation of the Voltan TMR node. More detailed information on the Voltan family of reliable nodes can be found in [30, 32]. Figure 6 shows the node hardware
organization of the present implementation which uses T800 transputers [18]. A transputer has four links, of these two are used for intra-node communication and two for inter-node communication. The three-processor TMR node masks the failure of one component which may be a processor and/or its links. Since a link failure can be seen as the failure of the processor associated with the link, we will only be concerned with processor failures.

Voltan nodes can either be used as independent reliable nodes or within a distributed system. When a node is used as an independent reliable node providing a dependable service to a client, the client machine will be connected to all three processors of the node using one external link of each processor. The client machine will communicate with the node in a replicated way, sending and receiving multiple copies (one copy for each node processor) of messages. This is the set-up adopted in our fault injection experiments. Voltan nodes can also be pipelined to form a distributed system as illustrated in [30]; in this case both external links of a node processor will be used and a node processor is connected to the corresponding processors of the two neighbouring nodes.

Voltan applications conform to the state machine model [26] consisting of processes which communicate with each other through message exchanges. Here is a typical state machine process that picks up a pending message, processes the message, and sends a result message (output message):

```
process S:
cycle
  receive(msg);
  process msg;
  send(result.msg);
end
end S
```

To allow such a process to be replicated, it is required that the computation performed on any selected message is deterministic. Furthermore, it is required that all non-faulty replicas of a process receive identical input messages in identical order. The output messages of the process replicas are voted upon to mask the failure of at most one processor. In Voltan nodes, this voting is performed at the node which generates the message.

The ordering protocol used in the Voltan TMR node requires that the clocks of the processors of the node be synchronized. This task is carried out by the clock synchronization module. So the system software of Voltan TMR node has three major modules: voting module, clock synchronization module and ordering module. A copy of the system software runs on each processor of a node. Figure 7 shows how these modules relate to a given application process replica S.

Figure 6. A Voltan node.

Figure 7. Voltan node software structure.

When messages destined for the application process S arrive at a processor, they are ordered by the ordering module. The ordering module makes use of the local clock (Clock) which is kept in synchronization with clocks on other correct processors by the clock synchronization module. The ordered messages are made available to S via the DMQ (delivery message queue). When the application process generates an output message, it is deposited in the PMQ (processed message queue). These deposited messages are voted by the voting module before being sent to their destinations.

The three modules of Voltan software all require the use of a message authentication mechanism—both for creating digital signatures and authenticating them. A message authentication mechanism allows a message to be signed and the signature of a received message to be verified. As a result, any alteration to a signed message can be detected by a recipient. The simplest form of digital signature is a checksum; checksums are adequate if it can be assumed that a processor would not deliberately forge signatures. More sophisticated forms of digital signature could be developed based on the techniques proposed in [24]. In Voltan nodes, a simple checksum based authentication mechanism is employed.

4.1. Voting module

The voting module (figure 8) consists of two processes: the diffuser process and the voter process. The diffuser picks up a message from the PMQ, signs the message, and puts a copy of it in the IMQ (internal message queue) and sends one copy to each of its two neighbouring processors. Each message contains a sequence number assigned to it by the application process. The sequence numbers are unique to each application process. Non-faulty replicas of a given application process will assign identical sequence numbers to message replicas. At the neighbouring processor, the authenticity of the incoming signed message is verified; if found authentic, the message is deposited at the local EMQ (external message queue). The job of the voter is to vote the matching messages in the IMQ and EMQ by comparing the contents of the messages (signatures of the messages are different and are not compared). Messages from IMQ and EMQ are matched by using their sequence numbers. When a message comparison is successful, the message from the EMQ is counter-signed (the local processor signature is added to the message). Such a double-signed message is then sent to its destination node. If a message comparison is not successful, the voter will look for the next matching message from the EMQ for comparison (for each message in the IMQ, there are two matching messages in the EMQ). At a destination node, only double-signed and authentic messages will be accepted for processing (such messages will be termed valid).
4.2. Clock synchronization module

The ordering protocol used in the Voltan TMR node requires that the clocks of the non-faulty processors of the node be kept in known and bounded synchronism, i.e., the absolute difference between the readings of non-faulty processors’ clocks at any given instant is within a known constant $\epsilon$. Clocks do not have identical running rates, so their readings will progressively drift apart from each other. (For crystal clocks, the drift rate is known to be within $2 \text{ us/yr}$.) The clock synchronization module ensures that processors’ clock readings are periodically adjusted to compensate for the effects of drift and are thus kept in the required $\epsilon$-bounded synchronism. It computes the necessary adjustments using the algorithm of [16].

As a processor’s physical clock is a read-only device, its readings cannot be adjusted directly. So, the reading of a processor’s synchronized clock (called simply clock hereafter) at any given moment is the physical clock reading plus the necessary adjustment. Thus, a processor’s (synchronized) clock can be modelled as the ordered sequence of clocks $C^1, C^2, \ldots C^k, C^{k+1}, \ldots$ and so on, such that $C^k = \text{physical clock reading} + \text{adj}^k$, where $C^k$ is the processor’s synchronized clock during the interval that follows the $k$th synchronization in which adj$^k$ is computed to be the necessary adjustment. The algorithm of [16] ensures that every non-faulty processor has exactly one $C^k$ as its current clock at any given moment, that the readings of the current clocks of any two non-faulty processors at a given moment are within $\epsilon$ of each other, and that the adjustments are never negative. The order protocol has mechanisms to deal with events that were scheduled to occur in the time intervals that would be lost due to clock jumps.

The clock module consists of two processes: time monitor (TM) and message manager (MSG) and any one of these two processes will perform the $k$th synchronization of a given processor’s clock (see figure 9). The TM process sleeps until the expected time when the next synchronization, say, $k$th synchronization, has to be done. Upon waking up, it will first check whether the $C^k$ has already been formed by MSG. If the clock has been formed, TM will do nothing and go to sleep until the next expected synchronization time. If the clock has not been formed, TM will broadcast a signed clock synchronization message to other processors saying “It is time to start $C^k$, form $C^k$ by setting $\text{adj}^k = \text{adj}^{k-1}$, compute the time for $(k + 1)$th synchronization, and go to sleep again.

When an authentic clock synchronization message arrives from a neighbouring processor, the MSG process picks it up from the CMQ (clock message queue). MSG will check whether the clock number ($k$) carried by the message matches the number it expects and whether the message arrives within an acceptable time frame (for details see [16]). If one of the conditions is not satisfied, MSG will do nothing and wait for the next synchronization message, if any. If both conditions are satisfied, MSG will relay the clock synchronization message (with its own signature added) to the other processor saying “It is time to start $C^k$, form $C^k$ by setting $\text{adj}^k = \text{expected time for $k$th synchronization} - \text{physical clock reading}$, compute the time for $(k + 1)$th synchronization, and wait for any $(k + 1)$th synchronization message to be received.

4.3. Ordering module

The ordering module employs the atomic broadcast protocol of [8] adapted for a fully and directly connected three-processor system. It ensures that all non-faulty replicas of an application process receive identical input messages in an identical order. It contains four processes (figure 10): broadcaster, relayer, transferrer and deliverer. The synchronized clock service required by the protocol is provided by the clock synchronization module described in section 4.2.

When a valid (i.e., double-signed and authentic) message is received at a processor, the broadcaster appends the message with the current reading of the local clock as message timestamp, signs the message (this third signature is needed because a timestamp has been added to the message), broadcasts the message to its two neighbouring processors, and also inserts a copy of it in the local OMQ (ordered message queue) where messages are queued in increasing timestamp order. When a broadcast message arrives at a processor, the relayer will receive it. Note that the message received by the relayer will have three signatures and would have been received from the processor that is the creator of the third signature. The relayer verifies the authenticity and timeliness of the received message (as specified by the atomic broadcast protocol). If the message is authentic and timely, it is relayed to the other (non-signatory) processor and a copy
The transferrer process picks up relayed messages, and inserts them in the local OMQ if the messages are found to be valid and timely. The message picked up by the transferrer will also have three signatures, but would have been received directly from the processor who is not the owner of the third signature. This simple way of distinguishing the broadcast messages from the relayed messages eliminates the need (as required by [SI) to sign a message by the relayer.

The deliverer process will be checking the messages in the OMQ regularly to see whether a message has become stable; a message with timestamp \( t \) becomes stable at local clock time \( t + \Delta \), where \( \Delta \) is the fixed ordering delay of the protocol described in [SI. The deliverer moves stable messages to the DMQ for consumption by the application process. The deliverer queues messages in the DMQ in increasing timestamp order, while duplicates are discarded.

5. Fault injection implementation

The Voltan software has been implemented on top of the Helios operating system [22] which runs on each transputer to provide essential operating system services. All of the Voltan software is written in C++, as are the fault injection objects. Each Voltan system service is provided by a system module consisting of a set of active objects. Messages are instances of a class called Message_Block. Queues are instances of a class called Message_Block_Queue. These passive (data) objects are used for communications between the active objects which represent processes. Active objects are also instances of C++ classes. The overall Voltan software system with application processes has the following form:

```c
/* active objects implementing voting module */
    Diffuser diffuser(&pmq, &imq, &mo0, &mo1);
    Voter voter(&imq, mp[1], &vmp);
/* active objects representing application processes */
    Application application(&dmq, &pmq);
```

With a particular fault injection object, the above program would change slightly to the following form:

```c
/* one of the parameters of the following object is changed */
    Diffuser diffuser(&pmq, &imq, &fq, &mo1);
/* fault injection object */
    Fault_Object fo(&fq, &moo);
    Voter voter(&imq, mp[1], &vmp);
    Application application(&dmq, &pmq);
```

This program will be capable of injecting faults in those output messages which are produced by the Diffuser and sent to one of the neighbouring processors, and hence can be used to test the effectiveness of the voting mechanism. Note that there are only two small differences: (1) application object is defined (started) with a different parameter; (2) an extra active object `fo` (of object class Fault_Object) and a queue it uses are added to the system. The Voltan system software does not need to be changed. Thus, the efforts involved in each fault injection experiment are kept to a minimum.

6. Experiments and results

According to the design, a Voltan TMR node should continue to function correctly even if one of its three constituent processors has failed. Before starting fault injection experiments, we had tested the software...
Our experiments concentrate on injecting faults to test the three fault tolerant modules, namely voting, clock synchronization and ordering modules. In particular, we wish to ascertain that a single processor failure does not cause the node to fail, even if the faulty processor behaves in a two-faced manner.

Faults are injected into the software of one of the three TMR node processors and the behaviour of the modules under test is observed in various ways. How faults are injected on the selected processor depends on which software module is being tested and the nature of that module.

6.1. Voting module

In the experiments, the three replicas of the server (S) running on a TMR node provide a reliable service, with clients (C1, C2) running on a separate processor sending requests and receiving replies. The system configuration is shown in figure 11.

The application server S running on the TMR node provides a positioning service. It holds two sets of coordinates for two graphical objects. Each client manoeuvres an object; for this purpose, it needs the positioning service provided by the server S. A client sends a request to the server giving its identity and the next position number. The corresponding reply from the server will contain the coordinates for the next position.

To test the voting module, we injected faults to emulate the behaviour of a faulty processor generating erroneous output messages. The correct functioning of the voting module can be observed by the clients from the fact that double-signed and authentic reply messages are still being sent by the TMR node despite the 'failure' of one processor.

The Voting module is a relatively simple module, it consists of two active objects (see figure 8). However, even such a simple module has been known to contain software bugs [34]. We inserted two injection objects (FO1 and FO2) each with its own queue (FQ1 or FQ2) between the diffuser object and the link handling objects in the software of one processor (see figure 12). The link handling objects which actually send the messages down the links is not shown in the figure.

Fault injection in diffuser.

This created the effects of a faulty processor producing incorrect output (reply) messages. It is the job of the voting modules on the other two correct processors to weed out wrong reply messages and so mask the failure of one processor.

6.1.1. Omission faults

In the experiment, we first injected consistent omission faults by having the injection objects delete messages. This simulates a faulty processor which is not producing any message for voting. Despite the silence of the faulty processor, the other two correct processors could still vote and manage to send double-signed replies to the clients. We then generalized the case whereby the processor sometimes appeared silent to just one of the remaining two processors. No bugs were discovered.

6.1.2. Value faults

In the experiment, value faults were injected by replacing one byte of application data with a randomly generated byte or by replacing the sequence number of the message with a random number. A new signature was also generated to replace the one on the intercepted message, otherwise the injected fault will be easily picked up by the message authentication mechanism. The two injection objects operated independently of each other. This creates the effects that the processor concerned is sending messages with wrong contents and correct signatures. The voting modules on the other two processors (where byte-by-byte comparison is performed) successfully detected and discarded all incorrect messages from the faulty processor.

During this experiment, a software bug regarding the data structure of a message was discovered. It was not in the voting module, but in the passive object class Message_Block. This was not expected, and so shows the value of fault injection testing.

6.1.3. Timing and arbitrary faults

Timing and arbitrary faults of a single failed processor should also not affect voting at the voters of the correct processors. This was the case when we injected late timing faults at the selected processor. Random and independent delays were injected by the two fault injection objects. The experiment was repeated for the case of arbitrary faults by injecting both timing and value faults. No bugs were discovered.

6.2. Clock synchronization module

A precise testing of any clock synchronization module is impossible unless special hardware support, such as the one used in [21], is available for correctly measuring clock differences. The impossibility arises from the fact that a processor cannot 'instantly' read another processor's
clock to check whether the clock difference at a given instant of time is within the bound $\epsilon$. The error or imprecision involved in reading a remote clock is influenced by variation in message transmission and processing delays. The special hardware support of [21] provides each processor with access to a global reference clock. With such a facility, a processor can then indicate to another processor its own time with reference to this globally accessible time base. This enables processors to compute accurately their relative differences at a given instant of the reference time.

In our testing of the clock synchronization module, no special hardware is used. We however circumvent the impossibility of instant access by exploring the minimum requirement imposed by the ordering module on the clock synchronization module. This requirement (see below) is weaker than requiring that correct processors’ clocks be synchronized within some known bound $\epsilon$. Thus the experiments reported here only check whether the clock synchronization module provides what is required from it by the ordering module, rather than whether processor clocks are synchronized within $\epsilon$. This is enough for our purpose which is to test Voltan TMR node software in implementing failure masking strategies.

We will first describe the mechanism we have set up to measure the difference in clock readings of two processors. This mechanism involves two processes (reader and checker), each running on a processor. The reader process on one processor reads its local clock and sends a message containing the clock reading to the checker process on the other processor. Upon receiving the message, the checker process reads its own clock and works out the difference by subtracting the clock reading contained in the message from the local clock reading. The actual message transmission and clocking delay involved in taking a measurement varies and is bounded by the known constant $\delta$.

Using this measurement mechanism, we will not test whether the actual difference between two clocks is within the bound $\epsilon$, but will ascertain whether the measured clock difference, $\delta$, is within the range: $-\epsilon < \delta < \epsilon + \delta$. A careful analysis of the correctness reasoning in [8] will indicate that the ordering protocol presented there will be correct so long as $-\epsilon < \delta < \epsilon + \delta$ holds; in fact any ordering protocol that assumes $\epsilon$-synchronized clocks will only require $-\epsilon < \delta < \epsilon + \delta$. (Note that $\epsilon$-synchronization implies that $-\epsilon < \delta < \epsilon + \delta$, but not vice versa.)

The testing of the clock synchronization module does not require the running of an application. The experimental set-up only involves a TMR node. Let the three processors of a TMR node be designated as P1, P2 and P3. P3 is selected for fault injection while the clock differences between P1 and P2 are measured. We put the reader process of the measurement mechanism on P2 and the checker process on P1.

It is assumed that there is no fault when the clocks of the processors are initialized. A simple non-fault tolerant program is used to initialize the clocks. Due to the way the clocks are initialized, we know that P1 is running ahead of P2 and P3.

As shown in figure 9, the clock synchronization module consists of two active objects (TM and MSG), either one of them can synchronize the local clock and send synchronization messages to other processors. We first fault-injected TM of the selected processor P3. Two fault injection objects were used as shown in figure 13.

6.2.1. Omission faults In the experiment, we first injected consistent omission faults by having the injection objects delete all clock synchronization messages from TM. The measurements taken on P1 and P2 indicated that the two non-faulty processors remained in synchronization. We then generalized the case whereby TM appeared silent to just one of the two processors. No bugs were discovered.

6.2.2. Value faults In the experiment, value faults were injected by adding a random value to the synchronization round number $k$ carried by the messages. A new signature was also generated to replace the old one on the intercepted message. This creates the scenario where a faulty processor sends clock synchronization messages with incorrect round numbers which should be rejected by non-faulty processors. The measurements taken on P1 and P2 indicated that the two non-faulty processors remained in synchronization.

6.2.3. Early timing faults The injection of early timing faults requires an additional fault injection object. The two existing fault injection objects now delete messages as if omission faults were being injected. The third injection object which is not shown in figure 13 will generate and send a clock synchronization message before the next round of synchronization is due. The message is only sent to P1, the processor with a fast clock. The aim here is to create a malicious fault scenario in which the faulty processor tries to push the correct processor with a fast clock even faster so as to cause a violation of the clock difference bound.

The experimental results were quite interesting. The measurements taken on P1 and P2 showed that on four occasions the recorded difference of clock readings of the two correct processors exceeded the bound of $\epsilon + \delta$, other figures were all within the bound. These violations happened during the first four rounds of fault injection. The experiment was repeated several times and this phenomenon recurred. This indicated a bug in the program.

Our subsequent analysis of the source code revealed a subtle bug. The clock initialization program makes use of the clock synchronization program. To allow the synchronization program to be used in this manner, the message timeliness check (which detects synchronization messages that arrive too early) is disabled during the initialization period. It takes two rounds of synchronization during the initialization period to get the clocks initialized within the required initial bound; at each round a processor
is expected to receive at most two messages. The timeliness check at each processor is restored after receiving exactly four messages. This is incorrect because according to the clock synchronization protocol, a processor with the fastest running clock does not receive any message from any other non-faulty processor. Due to this bug, P1—the processor with fastest running clock—thought it was still in the initialization period and did not do the timeliness check when the first four erroneous messages arrived, and allowed itself to be pushed exceeding the bound. This bug was corrected.

6.2.4. Late timing and arbitrary faults These faults were also injected in TM. The measurements taken indicated no further bugs.

We also fault-injected MSG of P3, using software structures similar to those used for injecting TM. The clocks of P1 and P2 remained synchronized.

6.3. Ordering module

With the voting module and clock synchronization module tested, we then went on to the testing of the ordering module. The testing of the ordering module relies on the correct functioning of both the voting module and clock synchronization module. If the ordering module of a non-faulty processor does not work correctly and as a result replicas of application processes on non-faulty processors end up processing different input messages and producing different output messages, the voting module will be unable to form a majority. So the failure of the ordering module, manifested by the lack of double-signed and authentic reply messages, can be observed by the clients.

The experimental set-up required for testing the ordering module is similar to that used for testing the voting module (see figure 11). The only difference is that the operations of the two clients C1 and C2 need to be coordinated. In order to put the ordering module through its paces, we would need a scenario like this: a processor receives C1’s request followed by C2’s request while another processor of the node receives C2’s request followed by C1’s request. To guarantee this scenario, a single process is used to simulate two clients sending independent requests.

Message ordering in the TMR node is achieved by the use of an atomic broadcast protocol [8] as mentioned earlier in the paper. The protocol achieves the required ordering properties in two stages: (1) broadcast, (2) relay. A message sent to the TMR node would first be timestamped and broadcast by the processor to its two neighbours, and the other processors would then relay the message to each other. The idea behind all this is that every one of the non-faulty processors should receive identical messages while timestamps are used to achieve identical message ordering.

As we have seen, the ordering module handles two types of messages received from other processors: broadcast messages at the broadcast stage and relayed messages at the relay stage. In other words, a faulty processor could only produce two types of message to 'confuse' non-faulty processors. In the experiments we fault-injected the software of one processor so it produced erroneous broadcast and relayed messages.

We first fault-injected the broadcaster of the selected processor by inserting an injection object as shown in figure 14. The injection object has two input and two output channels. The reason we used a single injection object instead of two separate ones is that we need to co-ordinate the injection to emulate a ‘two-faced general’ [20]. The effect of this fault injection is that erroneous broadcast messages will be generated by the processor selected for fault injection.

When injecting value faults we injected faults at the timestamp field of the messages. This is because the timestamp is the only piece of data appended to the message when a message is broadcast and it is the value of the timestamp that decides message order. Value faults injected in other parts of a message would be detected by the authentication mechanism which had been assumed to work correctly as stated before. The experimental results are described below.

6.3.1. Omission faults at the broadcast stage The fault injection object deleted broadcast messages. For the case of consistent omission faults, no broadcast messages were sent to neighbouring processors; while for the case of inconsistent omission faults, only one of the two neighbouring processors received broadcast messages. The ordering module and the TMR node as a whole were observed to work correctly.

6.3.2. Value faults at the broadcast stage The injection object adds a random number to the timestamp of a message and generates a new signature for the message to replace the one generated by broadcaster. When we had the injection object add the same random number, to the timestamps of the two messages (emulating a consistent value fault), experimental results showed that the ordering modules of the two non-faulty processors worked as expected. However, when different random numbers were used by the injection object (emulating an inconsistent value fault), creating the scenario of a ‘two-faced general’, identical message ordering at the server replicas running on the two non-faulty processors was not always achieved. The cause of the problem was eventually traced to an incorrect optimization of the broadcast protocol. A non-faulty processor must perform the following check to detect a ‘two-facing’ failure: the message received directly from a given processor and a copy signed and relayed by the other processor must have identical timestamps. If they are not identical, the original sender must be concluded to have sent the same message with different timestamps to
different processors. Note that this conclusion is valid only when the direct and the relayed messages are received; if one of them is not received (within a finite time) then no conclusion can be drawn as to whether the sender or the relayer is faulty. So when only one message is received, no check can be, and therefore needs to be, done. In our implementation, in an (over-zealous) attempt to minimize the storage requirement, the relayed message was simply discarded whenever it arrived after the original message was received; because of this, a 'two-facing' failure was observed. For creating the required fault scenarios, the injection of faults in more than one processor at a time. A very useful generalization of an Arjuna fault tolerant distributed system as described in this paper, we have injected faults in only one processor at a time. A very useful generalization would be the capability of injecting faults in multiple processors, as many fault tolerant systems are designed to tolerate multiple failures. For creating the required fault scenarios, the injection of faults in more than one processor will require a certain amount of coordination among the fault injection objects running on the processors. We are currently developing such coordination mechanisms. Secondly, we note that in many systems, it is impractical (or not possible) to insert fault injection objects in the target system software for intercepting and manipulating output messages. In order to use the focused fault injection method under these circumstances, we will need the ability to intercept communication messages within the underlying communications software. Once communication messages have been intercepted, it then becomes possible to manipulate them in a manner discussed in this paper. A tool, Delayline [17], developed by our colleagues has such message interception capability for messages exchanged using UNIX sockets. Although Delayline was originally developed for emulating wide area network characteristics over a local area network for UNIX based distributed systems (essentially by artificially increasing the transmission times of the intercepted messages), it can be applied quite well for fault injection testing. We are planning to use the Delayline tool for fault injection testing of an Arjuna fault tolerant distributed system [29], and expect to report our findings in a future publication.

Acknowledgment

The work reported here has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC).

References

detection mechanisms efficiency: results Proc. 12th Int.
Symp. on Fault-Tolerant Computing (FTCS-12) (Santa
Monica, CA) (Silver Spring, MD: IEEE Computer Society) pp 375–6
tolerance testing in the advanced automation system.
Proc. 21st Int. Symp. on Fault-Tolerant
Computing (FTCS-21) (Montreal) (Washington, DC:
IEEE Computer Society) pp 18–25
fault injection for fault-tolerant protocol testing Proc.
21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)
fault-tolerant distributed system testing Proc. 1992 IEEE
Workshop on Fault-Tolerant Parallel and Distributed
Systems (Amherst, MA) (Washington, DC: IEEE
Computer Society) pp 28–35
[14] Ezhilchelvan P D and Shrivastava S K 1986 A
classification of faults in systems Proc. 5th Symp. on
Reliability in Distributed Software and Database Systems
Los Angeles, CA) (Washington, DC: IEEE Computer
Society) pp 215–22
error detection schemes using fault injection by
heavy-ion radiation Proc. 19th Int. Symp. on
Fault-Tolerant Computing (FTCS-19) (Chicago, IL)
Fault-tolerant clock synchronization Proc. 3rd ACM
Symp. on Principles of Distributed Computing
(Vancouver, B.C.) (New York: ACM) pp 89–102
wide-area network emulation tool USENIX Comput.
Syst. 7 313–32
(Englewood Cliffs, NJ: Prentice Hall)
FERRARI: a tool for the validation of system
dependability properties Proc. 22nd Int. Symp. on
Fault-Tolerant Computing (FTCS-22) (Boston, MA)
generals problem ACM Trans. Program. Lang. Syst. 4
382–401
[21] Palumbo D L 1994 The derivation and experimental
verification of clock synchronisation theory IEEE Trans.
Comput. 43 676–86
[22] Perihelion Software Ltd 1989 The Helios Operating System
(Englewood Cliffs, NJ: Prentice Hall)
[23] Powell D 1992 Failure mode assumptions and assumption
coverage Proc. 22nd Int. Symp. on Fault-Tolerant
Computing (FTCS-22) (Boston, MA) (Washington, DC:
IEEE Computer Society) pp 386–95
obtaining digital signature and public-key cryptosystems
Commum. ACM 21 120–6
injection and its application in distributed systems Proc.
23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23)
[26] Schneider F 1990 Implementing fault-tolerant services
using the state machine approach: a tutorial ACM
Comput. Surv. 22 299–319
Experimental evaluation of two concurrent error
detection schemes Proc. 16th Int. Symp. on
Fault-Tolerant Computing (FTCS-16) (Vienna)
testing environment Proc. 18th Int. Symp. on
Fault-Tolerant Computing (FTCS-18) (Tokyo)
overview of the Arjuna distributed programming system
IEEE Software 8 66–73
Tully A 1992 Principal features of the VOLTAN family of
reliable node architectures for distributed systems
Understanding component failures and replication in
distributed systems ISA Project Report (UNITR1)
Department of Computing Science, University of
Newcastle upon Tyne
[32] Speirs N A, Tao S, Brasiliero F V, Ezhilchelvan P D and
Shrivastava S K 1993 The design and implementation of
VOLTAN fault-tolerant nodes for distributed systems
Transput. Commun. 1 93–109
[33] Tao S 1995 Fault injection based testing of software
implemented fail-silent node Technical Report
Department of Computing Science, University of
Newcastle upon Tyne
[34] Yang X-Z and York G 1985 Fault recovery of triplicated
software on the Intel IAPX 432 Proc. 5th Int. Conf. on
Distributed Computing Systems (Denver, CO) (Silver
Spring, MD: IEEE Computer Society) pp 438–43