Somersault Software Fault-Tolerance

Paul Murray, Roger Fleming, Paul Harry, Paul Vickers
Internet Communication Services Department
HP Laboratories Bristol
HPL-98-06
January, 1998

software fault-tolerance, process replication failure masking, continuous availability, topology

The ambition of fault-tolerant systems is to provide application transparent fault-tolerance at the same performance as a non-fault-tolerant system. Somersault is a library for developing distributed fault-tolerant software systems that comes close to achieving both goals.

We describe Somersault and its properties, including:

1. Fault-tolerance – Somersault implements "process mirroring" within a group of processes called a *recovery unit*. Failure of individual group members is completely masked.
2. Abstraction – Somersault provides loss-less messaging between units. Recovery units and single processes are addressed uniformly as single entities. Recovery unit application code is unaware of replication.
3. High performance – The simple protocol provides throughput comparable to non-fault-tolerant processes at a low latency overhead. There is also sub-second failover time.
4. Compositionality – The same protocol is used to communicate between recovery units as between single processes, so any topology can be formed.
5. Scalability – Failure detection, failure recovery and general system performance are independent of the number of recovery units in a software system.

Somersault has been developed at HP Laboratories. At the time of writing it is undergoing industrial trials.

Internal Accession Date Only

Somersault Software Fault-Tolerance

The ambition of fault-tolerant systems is to provide application transparent fault-tolerance at the same performance as a non-fault-tolerant system. Somersault is a library for developing distributed fault-tolerant software systems that comes close to achieving both goals.

We describe Somersault and its properties, including:

- Fault-tolerance – Somersault implements "process mirroring" within a group of processes called a *recovery unit*. Failure of individual group members is completely masked.
- Abstraction – Somersault provides loss-less messaging between units. Recovery units and single processes are addressed uniformly as single entities. Recovery unit application code is unaware of replication.
- High performance – The simple protocol provides throughput comparable to non-fault-tolerant processes at a low latency overhead. There is also sub-second failover time.
- Compositionality – the same protocol is used to communicate between recovery units as between single processes, so any topology can be formed.
- Scalability – failure detection, failure recovery and general system performance are independent of the number of recovery units in a software system.

Somersault has been developed at HP laboratories. At the time of writing it is undergoing industrial trials.

1 Introduction

Highly available software systems for business applications have been available for some years now. There are cluster products that manage a collection of computers to automatically restart failed applications, e.g Hewlett-Packard's MC/ServiceGuard [13], and Microsoft's Wolfpack NT Server Cluster. There are also middleware software systems that support messaging paradigms from which fault-tolerant software can be built. Perhaps the most notable of these is the ISIS toolkit [3].

When choosing a technology base for a highly available application there are several factors to consider: the required level of availability, the required performance, and the cost (both development and operational). Different components will have more or less impact on the overall system when they fail. Replacing standard components with

custom-built components incurs a higher development cost. We believe that by considering each component of a distributed system individually a different trade-off will be discovered and a different technology is appropriate.

This paper describes Somersault: a library for developing distributed fault-tolerant software systems. Somersault is aimed at replicating software components to achieve continuous availability without introducing complexity for the programmer or altering the overall system design. Somersault introduces fault-tolerance in a component-wise manner, only where it is needed. Like micro-kernel approaches such as [2], we encapsulate fault-tolerance mechanisms within a middleware layer below the application.

One of the key aspects of Somersault is the simplicity of its protocols, permitting high throughput and low latency in replicated processing. Failover can be achieved in sub-second timing.

Communication is asynchronous and uses an identical protocol for a replicated group of processes (called a recovery unit) as for a single process, isolating the rest of the system from the effects of making one component fault-tolerant.

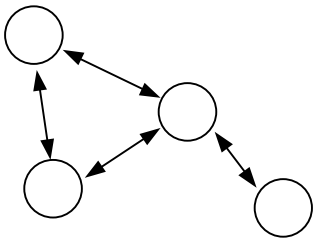
Replication and communication are also independent of system size, allowing Somersault to scale to any number of units without affecting performance.

Somersault is an ideal complement to other technologies, such as clustering for high availability. Developed at Hewlett-Packard Laboratories, it has been implemented and tested on both HP-UX and Windows NT. At the time of writing Somersault is undergoing industrial trials.

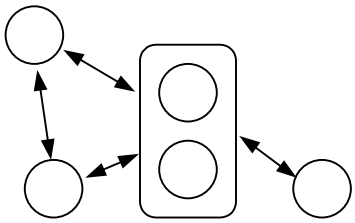
In the following section we describe the approach we have taken to modelling a fault-tolerant system. In section 3 we describe the operation and performance of Somersault. Section 4 describes features of replication that the application programmer needs to be prepared for. The final section briefly reviews the main features of Somersault and how it relates to crash-restart cluster technologies .

2 Model

2.1 System Model



A regular distributed system has communicating processes



A Somersault distributed system has communicating units – units are single processes or groups of replicated processes

Figure 1 Somersault System Model

Somersault uses a model of processing units that communicate via messages. There are two types of unit: the simple unit, which contains a single process; and the recovery unit, which contains a collection of processes. A unit is always viewed as a single entity and all communication is between units.

The replicated processes in a recovery unit are redundant copies of one another. A process is a member of exactly one unit[1].

Although a simple unit can be viewed as the special case of a recovery unit that contains only one process, we prefer to consider it separately.

2.2 Replication

We model an application process as a state machine driven by non-deterministic events. We assume that the only visible behaviour of a process is its message communication. Two processes are replicas of one another if they input and output the same messages in the same order.

From a process' point of view, input messages are non-deterministic events, but there may also be other non-deterministic events – reading a real-time system clock, for example.

---

[1] We avoid the term "group" to emphasise that fact that a unit is strictly a collection of replicated processes. Group communication systems often use a model in which a process can be a member of more than one group, can receive messages broadcast to any of its groups, and can communicate as an individual. ISIS is an example of a group communication system [3].

Output messages are deterministic events. As a process executes it may input messages and output messages in some sequence. Some non-deterministic events will determine that sequence and hence determine the observable behaviour of the process. Some non-deterministic events will not influence the observable behaviour at all.

Two copies of the same process will be replicas, if they:

1. Receive the same input messages.
2. Experience the same non-deterministic events that determine observable behaviour.

A recovery unit contains replica processes.

2.3 Failure

We assume processes have fail-stop semantics (fail by halting [8], [15], [16]).

A recovery unit will replace a failed process by making a copy of a surviving process.

If all processes in a unit fail the unit will cease to exist. This will be apparent to other units only due to an inability to communicate with it. In particular, a unit does not fail due to an inability to communicate with other units.

2.4 Reliable Messaging

Messaging between units is defined in by properties that relate the send and receive of messages to and from a unit, to the send and receive of the unit's replicas. The way we present these properties is biased by our implementation described later.

1. If unit A sends a message to unit B then every replica in unit A is certain to generate the message.
2. Unit B has received a message from unit A when every replica in unit B has received it.
3. If unit A receives two messages then every replica in unit A receives them in the same order.

Notice that these properties allow one replica to receive and generate a sequence of messages before other replicas receive any messages.

3 Inside Somersault

In the following we will use two sets of terms for talking about message passing. We will use "send" and "receive" to refer to the actual transfer of a message using a transport protocol such as TCP. Send and receive will always be performed by the Somersault layer.

We will use “consume” and “generate” to refer to the transfer of an application message to and from the application code within a process.

Somersault uses three mechanisms: failure detection, replication, and reliable messaging. Failure detection is distinct from the other two, whereas replication and reliable messaging interact.

A unit is a number of processes that together monitor each other for failures. Some of these processes, but not necessarily all, contain application level replicas. The failure detection mechanism decides a consistent view of which processes are alive. It performs no other function.

Replicated application processing is achieved by process mirroring. Process mirroring enforces replication by managing non-deterministic events. However, delivery of messages is handled by the reliable messaging protocol. Hence process mirroring is closely tied in to the messaging protocol.

Our reliable messaging protocol is called Secondary sender. Secondary sender supports the guarantees required for process mirroring and failure recovery. Together, these two provide failure masking.

In describing Somersault we will consider the minimum recovery unit, with three processes. All of these processes are involved in failure detection, only two in application replication.

At the end we comment on how this generalises to more than two replica processes.

3.1 Failure Detection

We want processes to be fail-stop [8]. It has been shown in [15] that fail-stop processes can be simulated on asynchronous communications using group membership consensus algorithms.

We implement fail-stop processes in two steps. The first step is based on heartbeats. The processes pass heartbeat messages to one another at regular intervals. If one process misses its deadline for delivering heartbeat messages, another process will suspect it has failed. This step is particularly sensitive to the throughput, latency and load on the network. Delay in message delivery can lead to “false positive” failure detection ([6], [14]) so the heartbeat interval has to be adjusted to reflect the profile of network traffic.

When one process suspects another of failing it invokes a group membership consensus protocol. There are various group membership protocols that could be

used, including those reported in [4], [5], [9], [11] and [18]. We do not claim to be original here.

During the membership protocol each process identifies other processes that it can communicate with and form a group. If a group of processes find they form a majority of the unit, they declare themselves to be the new unit membership.

The requirement for a majority decision on unit membership is what determines the minimum unit of three processes. If the unit reduces to two processes failure detection can not be achieved reliably.

When consensus has been reached the unit membership is made available to the secondary sender protocol, which will mask the failure.

Generally processes in a recovery unit contain replicas of the application code, but some may not. Those that do not are present purely for failure detection and are called "witnesses". Replication is covered in the next section.

3.2 Replication – Process Mirroring

In process mirroring, two copies of the same application code are forced to act as replicas by allowing one (called the "primary") to perform non-deterministic events arbitrarily and then making the other (called the "secondary") perform the same events. Hence the secondary reflects the actions of the primary.

Process mirroring is achieved by placing a logging channel between the primary and secondary. Non-deterministic events are logged on the channel and fed to the secondary process in order.

Exactly how to feed a non-deterministic event depends on the type of event. There are basically two types to consider:

1. Those that are initiated from outside the process, e.g. input messages, timers.
2. Those that are initiated by the process, e.g. system calls (we refer to this type as a non-deterministic choice.)

Somersault controls event scheduling in the application process. It determines the delivery of timers and messages, and it controls thread execution. Event scheduling is a matter of managing the occurrence of events to which the application responds.

Non-deterministic choices are more difficult because they occur as part of the execution of application code. Somersault uses a mechanism to capture the result of a non-deterministic choice in the primary and inject it at the secondary. This requires the co-operation of the application programmer as described in section 4.

As we mentioned earlier, it is not necessary to replicate all non-deterministic choices for the two processes to behave as replicas. If it is possible to identify which choices do not affect process mirroring then the programmer may decide not to log the choice. Minimising the replication in a recovery unit can improve its robustness. Memory leaks in heap allocation are good example. If only the content of a heap is replicated, rather than its actual structure, then the primary and secondary can allocate memory differently and run out at different times. This is particularly true if one replica has been running longer than the other has (i.e. the secondary was created later).

3.3 Secondary Sender

The secondary sender protocol manages reliable FIFO messaging between units. Previous work by Alsberg and Day, described in [1], promotes the use of a primary site server for performing updates in a replicated resilient server group. The secondary sender algorithm is similar to the approach taken by Alsberg and Day. There are two main differences between our work and theirs. Principally we have applied our work to a peer-to-peer asynchronous message-passing environment, whereas they use a synchronous message-passing client-sever environment in which only servers are resilient. Secondly, the protocol we describe here has been designed to support process mirroring.

In the following we describe the secondary sender protocol in regular operation, actions during failover, and actions during recovery.

3.3.1 Messaging

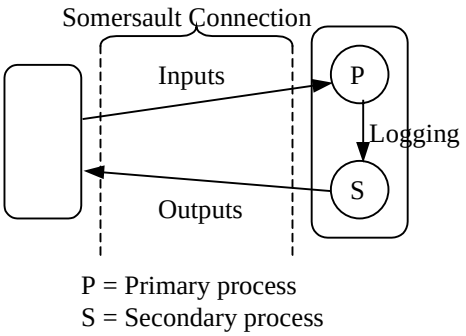


Figure 2 Somersault connection

A connection between two units is implemented using two reliable FIFO connections: one for receiving messages and one for sending them, as shown in Figure 2. The application code at both primary and secondary consumes the same input messages and generates the same output messages.

For the sake of descussion we describe messaging situations that occur in request/response scenarios. We do not wish to imply that Somersault adopts a request response paradigm. Somersault uses peer-to-peer asynchronous messaging between units.

The Roles

The secondary sender protocol defines two roles: sender and receiver. The primary process always adopts the receiver role, because message receipt is a non-deterministic event. If there is a secondary it adopts the sender role (hence the name secondary sender), otherwise the primary is also the sender.

The receiver and sender roles differ in how they obtain input messages and what they do with output messages.

The receiver (primary process):

1. Receives input messages on the input link from another unit.
2. Consumes input messages at the application level.
3. Generates output messages at the application level.
4. Maintains a re-send queue for output messages.
5. Passes input messages on the logging channel (if there is a secondary process).

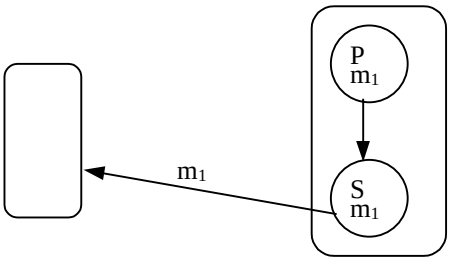
The sender (primary or secondary process):

1. Receives input messages on the logging channel (if the sender is a secondary process).
2. Consumes input messages at the application level.
3. Generates output messages at the application level.
4. Maintains a re-send queue for output messages.
5. Sends output messages on the output link to another unit.

When the secondary sender protocol sends an application message it adds a header to include its own control information. This includes message acknowledgements, which are used to garbage-collect re-send queues.

If there is only one process in a unit it takes both receiver and sender roles; there is no logging channel and there is only one re-send queue. This is the case for simple units and recovery units that are reduced to one replica.

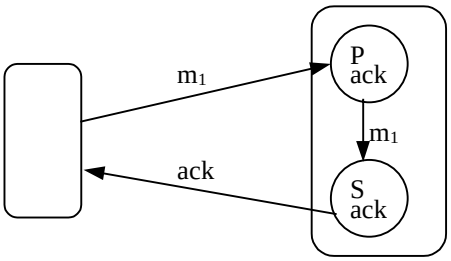
The properties



The secondary sends messages, so both the primary (P) and the secondary (S) have a copy

Figure 3 Message Sent Property

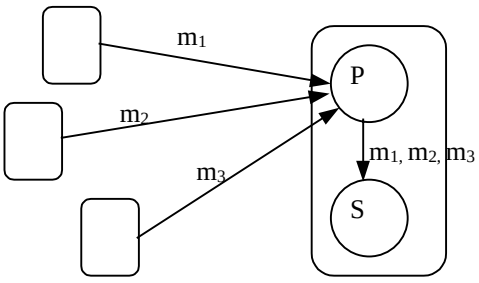
If a unit sends a message it is sent by the secondary. Process mirroring ensures that the secondary reflects the primary, so both the primary and secondary will generate the message. Hence the first property in section 2.4 is satisfied.



The secondary sends the ack, so both the primary (P) and the secondary (S) have received m_1

Figure 4 Message Received Property

When a message is received by a primary, it is passed to a secondary. At some point the secondary will send an acknowledgement for the message (typically in the header of an output message). So, if a unit receives an acknowledgement it can be sure that each process in the sending unit has received the message that is being acknowledged. Hence the second property in section 2.4 is satisfied.



The primary (P) logs m1, m2, m3 to the secondary (S) in the order that they were consumed

Figure 5 Message Order Property

All input messages are received by the primary and logged to the secondary. Message consumption is a non-deterministic event, so the secondary will consume messages in the same order as the primary. The last property in section 2.4 is satisfied.

3.3.2 Failover

When a process fails the failure detection mechanism will establish the new unit membership. The secondary sender protocol reacts by reallocating roles. If a sender or receiver role has to be moved then a connection has been lost. The failover procedure reconnects units and re-sends messages that may have been lost due to the failure. By reconnecting the units and recovering lost messages, the secondary sender protocol preserves its messaging properties across a failover, masking the failure.

To look at failover in more detail we will examine what happens when different processes fail.

Primary fails

The primary process has the receiver role. The primary is an endpoint for the input link of each connection to other units. The loss of the primary results in the loss of the input links. Messages that the primary was in the process of receiving or had not yet passed down the logging channel will also be lost.

The secondary process will respond by promoting itself to primary, retaining the sender role and adopting the receiver role. A new input link is established for each connection and the far end re-sends messages. This will replace any messages that the secondary had not received; any duplicate messages can be ignored.

At this point the unit has completed failover. This procedure regains all connections and recovers any lost messages.

Secondary fails

The secondary process has the sender role and is therefore the endpoint for the output link of each connection to other units. The loss of the secondary will result in the loss of the output connections. The secondary may have failed in the process of sending a message or before it sent a message that the primary thinks has been sent. So output messages may have been lost.

The primary will respond to the failure by adopting the role of sender as well as receiver. A new output link is established for each connection and output messages in the re-send queue are sent again. The far-end receiver ignores duplicate messages.

At this point the unit has completed failover. This procedure regains all connections and recovers any lost messages.

Witness fails

The witness is not a replica and is not involved in the secondary sender protocol. Its failure is of no consequence to application processing, but the unit may be unable to perform failure detection without it. It should be replaced as soon as possible.

Additional notes

In the case of primary failure it is possible that the primary has performed some work that the secondary never found out about. After failover the secondary will receive the same input messages and repeat the work, possibly with a different outcome due to unreplicated non-deterministic events, including the message receipt order.

This is not a problem because of the reliable messaging properties. If input messages have not been logged, the unit has not received them and they can not affect messages output by the unit.

If two units experience simultaneous failures they can lose both links of a connection between them. The recovery will involve re-establishing the connection and both sides re-sending messages. Complete loss of a connection due to simultaneous failures at each end does not affect the messaging properties.

3.3.3 Recovery

Once a failover has occurred, the number of processes in a recovery unit is reduced and the unit's ability to survive further failures is compromised. It is necessary to replace the lost process to recover the desired level of fault-tolerance. Somersault handles automated replacement of process.

Failure of primary or secondary process results in the need to replace a secondary (if the primary failed the secondary will have become a primary). So there are only two cases to consider: replacing a secondary or a witness.

The witness is the easiest case because it is not involved in the secondary sender protocol. A new witness process is started and its first action is to contact the unit and invoke a change in the unit membership. If this step is not successful after a few attempts the witness will give up. If the step is successful the unit is back to full strength.

The first step of creating a new secondary is the same as for the witness. Once it has joined the unit it must become a replica. This involves copying runtime state from the

primary to the secondary and then synchronising to join the secondary sender protocol.

A logging channel is opened from the primary to the new secondary. The primary serialises its state and passes it down the logging channel to the new secondary, which rebuilds the state.

When all the state has been passed, the secondary adopts the sender role. A new output link is established for each connection. This time there is no need to re-send any messages because the role hand-over can be performed gracefully without messages being lost.

The primary proceeds with logging all input messages and non-deterministic choices to the secondary. The unit is now back to full strength.

The whole procedure preserves the reliable messaging properties.

The above description assumes that the state of the primary is so small that it can be passed to the new secondary without compromising availability of the unit. Sometimes this is not the case.

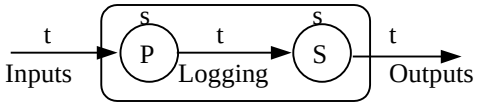
The state transfer can be fragmented and interleaved with regular application processing. This is called a fuzzy state transfer (or fuzzy checkpoint).

Fuzzy state transfers introduce two requirements:

1. The state can be transferred more quickly than it changes.
2. The secondary eventually has an exact copy of the primary's state.

These requirements demonstrate a race condition that can occur in fuzzy state transfers. The condition represents a trade-off between regaining fault-tolerance and service availability. Several approaches to fuzzy state transfers have been developed; a small selection can be found in [2], [10] and [12].

3.4 Performance - Early message logging verses late message logging



The processes in a recovery unit exhibit pipeline parallelism. t = message transfer time, s = processing time.

Figure 6 Breakdown of Recovery Unit Processing

In the last section we stated that the secondary sender protocol passes input messages from the primary to the secondary process via a logging channel. This establishes a pipeline, allowing the primary and the secondary to process input requests in parallel.

The logging channel also passes over other non-deterministic events to implement process mirroring.

The order in which events are logged with respect to one another and the deterministic processing affects the performance characteristics of the pipeline. Here we consider two obvious cases that we refer to as late message logging and early message logging.

Late message logging

When a message arrives at a primary it will be consumed at the application level and then logged. As a consequence, all processing invoked by the message is performed before it is logged to the secondary. The secondary will not be able to process it in parallel with the primary, although it will process it at the same time as the primary processes subsequent messages.

Let us say that the time taken to transfer a message across a link is uniform and denoted by t and that the time taken to consume a message and perform the subsequent processing is denoted by s (as shown in Figure 6). Then a request-response message pair will take 3t+2s. Any non-deterministic event logs at the primary can be buffered and sent together with the input message, so this figure is a worst case.

The throughput of late logging is theoretically as high for a recovery unit as for a simple unit.

Early message logging

In early message logging the primary logs input messages before consuming them. The objective is to reduce latency by processing the message at the primary and the secondary at the same time.

Using the same denotation as above we find the latency for a request-response message pair to be 3t+s.

In this case the non-deterministic events processed at the primary will not be logged to the secondary until some time after the message that lead to them has been logged. It is possible that the secondary will have to wait for the events to be logged, holding it up. If the events are passed down the logging channel as they occur, it is possible that the time taken waiting for them will be greater than the entire processing time.

For early message logging the latency of 3t+s is a best case. The worst case may be much greater. The logging overhead of early logging may also reduce throughput.

Whether to use early or late message logging depends on the value of s and t and the occurrences of non-deterministic choices in the application code.

In cases where the value of s is insignificant compared to t, early logging will not significantly improve latency; in fact in some cases the latency will increase.

In cases where the value of t is insignificant compared to s, the advantage of early logging will be much greater, but still dependent on the occurrence of non-deterministic events.

3.5 Units with more than two replicas

So far we have described the operation of secondary sender protocol for units with only two replicas. The utility of units with more than two replicas is questionable. When more than one replica fail simultaneously it is either because they have a common failure mode or not. If it is a common failure mode, adding more replicas does not help. If not, then the dual failure is a very rare occurrence – two replicas will typically provide enough protection for most applications.

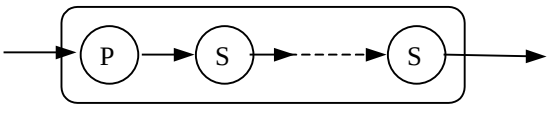
If more protection is required, then we do not want to add any more replicas than we must. For failure detection purposes, to detect k simultaneous failures, a unit must have 2k+1 processes [17]. We can use k+1 replicas and k witness processes. Witness processes incur a low overhead and do not interfere with application processing.

One good reason for more than two replicas is to facilitate deliberate unit migration for hardware/operating system maintenance purposes.

The secondary sender protocol can be generalised for units with more than two replicas. Here we briefly describe two alternatives: chaining and fan-out. In each there is one primary process and a number of secondary processes. The difference between approaches is the logging topology for process mirroring. There are many other alternatives.

Chaining

The chaining approach is a straightforward generalisation from the two-replica unit. In this case there are a number of secondary processes all daisy-chained with logging channels. The primary is the receiver and the last secondary in the chain is the sender. This is shown in Figure 7 below.



Multiple secondary processes can be daisy-chained with logging channels

Figure 7 Chained Secondary Processes

The additional secondary processes consume and generate messages. They also maintain a re-send queue. The main difference is the need to maintain a re-send queue for log messages. The chaining approach requires modifications to the failover actions as follows:

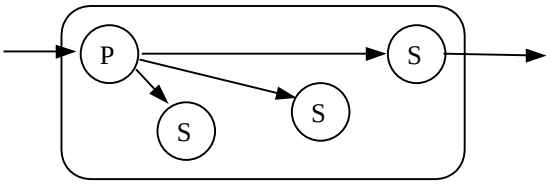
1. Sender fails - previous secondary in the chain adopts sender role and performs failover.
2. Primary fails - first secondary in the chain becomes primary and adopts receiver role.
3. Mid-chain secondary fails - the previous process in the chain links with the next one down and resends log messages that may have been lost.

So long as the sender role belongs to a secondary, there is no need to move it during recovery. When a new secondary is built it can be inserted into the chain, avoiding the need to move the sending link for connections with other units.

A similar topology to the above was described in [1], but there responses were not sent from the end of the chain.

Fan out

In the fan-out approach every secondary receives logs directly from the primary. Only one secondary process is the sender. This is shown in Figure 8 below.



Multiple secondary processes can each receive logs directly from the primary

Figure 8 Fan-out Secondary Processes

As for chaining, the additional secondary processes consume and generate messages, and maintain a re-send queue. In fan-out, all the processes must maintain a re-send queue for log messages, even though only the primary sends them.

The fan-out approach requires modifications to the failover actions as follows:

- Sender fails - another secondary is elected to take sender role.
- Primary fails – a secondary is elected to take over as primary. On promotion the new primary must obtain the most recent log messages from the other secondaries before accepting re-sent messages from other units.

- If a secondary that does not have a sender or receiver role fails it has no impact and no actions needs to be taken.

The fan-out approach has a lower latency overhead in responding to messages because only the primary and sending secondary fall in the critical path for processing inputs. Also, the failure of a spare secondary has no consequence. However, the failover procedure is more complex and simultaneous failure of sender and receiver breaks the messaging properties. There is also an additional complexity in deciding when messages can be dropped from log re-send queues.

4 Isolation of Application code from FT

The secondary sender protocol provides reliable messaging between units. All units are addressed uniformly as units, providing both location independence and abstraction from the processes that implement a unit. The secondary sender protocol hides replication in one unit from another.

Process mirroring hides replication within a unit. The application programmer uses Somersault as a point-to-point connection-oriented communication transport between units. All replicated events, such as messaging, thread scheduling, flow control, timers etc. are handled by Somersault transparently to the application programmer.

However, there are two features of process mirroring that the application programmer must support:

- Non-deterministic choices must be made visible to Somersault.
- State transfer must be provided.

These facilities are made as syntactically transparent as possible, but it is up to the application programmer to use them.

Non-deterministic choices are performed by effectively telling Somersault to call a function with a non-deterministic result on behalf of the application. If the application code is in a primary process, Somersault calls the function, logs the result and returns it to the application. If the application code is in a secondary process, Somersault does not call the function; instead it looks up the result logged by the primary, and returns that to the application. At no point is the application code aware that the result of a non-deterministic function is being captured or substituted. Hence the same application code is used in both the primary and secondary and the same path through the code is followed in all cases.

State transfer is basically a checkpoint of application state. The programmer is best placed to decide what the application state is and how to copy it. The transfer involves serialising the application state at the primary, passing it to the secondary and reconstructing it there.

When a process is started Somersault decides if it is a completely new unit or a new process for an existing unit. If the later is the case the first event that Somersault schedules is a state transfer.

State transfer functions are provided by the programmer and registered with Somersault. When a new process is started, Somersault will invoke the functions to send a state transfer in an existing process and those to receive a state transfer in the new process.

The application code does not direct the transfer, it does not determine when it is time to transfer, and it does not know about the process that is receiving the state.

State transfer and non-deterministic choice logging are the only replication facilities that the application programmer needs to support. Logging and state transfer only occur in recovery units, so simple units do not require them. The only requirement for simple units is to use the Somersault transport when communicating with recovery units.

5 Discussion

The secondary sender protocol uses a synchronous logging algorithm. The secondary sends output messages, so logging from primary to secondary occurs before messages are sent. As a consequence the properties described in section 2.4 can be maintained with minimal messaging. The protocol is very simple and efficient.

The primary and secondary processes of a recovery unit exhibit pipeline parallelism. The time taken for a recovery unit to process a message is slightly greater for a unit than for a single process, but the throughput is roughly the same in each case because of the pipeline processing.

The time taken for a recovery unit to process a message is longer because the secondary will process it slightly behind the primary. In section 3.4 we showed that late logging provided a round trip time of 3t+2s for a request-response pair of messages, as compared 2t+s for a single process performing the same task. An appropriate early logging recovery unit could achieve a round trip latency of 3t+s. A

recovery unit can achieve the same throughput as a simple unit using either late-logging and early-logging.

A benefit of the secondary sender protocol is the uniformity of connections between units having any number of processes. The protocol and the overhead of communicating between units are the same in all cases. This makes it easy for a single process to be replaced by a recovery unit in a distributed system.

Somersault presents units to the application programmer as single entities. Also, great effort has been made in isolating fault-tolerance mechanisms from application code, especially in the case of recovery units talking to other recovery units. These factors make fault-tolerant application programming relatively simple.

Clusters

The crash restart approach to high availability is used in cluster systems such as Hewlett-Packard's MC/ServiceGuard and Microsoft's Windows NT Server Cluster. The objective of these systems is to reduce downtime by automatically restarting applications when software or hardware failures occur.

Some distributed system components have greater consequence if they fail. For example, in a distributed transaction processing system:

- Loss of a process executing a single user transaction may be noticed by a single user and require a transaction rollback;
- Loss of a naming/location service may result in processes being unable to find one another, but may not effect those that are already communicating;
- Loss of a centralised lock manager may require rollback of all running transactions and total system unavailability until recovery is completed.

Somersault increases system availability by avoiding failure of software components. A system designer may determine that the Somersault approach is overkill in general, but may be useful in the case of critical system components, such as the lock manager case above.

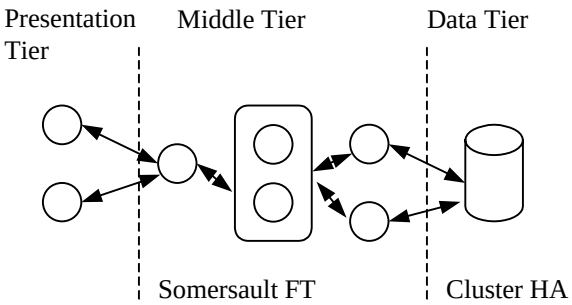


Figure 9 Highly Available Three Tier Architecture

The combination of cluster-based crash-restart for general protection and Somersault fault-tolerance for system-critical components creates a useful environment for highly available systems. This is particularly the case in two or three tier architectures, where presentation tier elements are permitted to fail, the middle tier contains critical services suited to Somersault, and the data tier contains databases suited to cluster approaches. This is shown in Figure 9.

6 References

[1] P. A. Alsberg, J. D. Day, A Principle for Resilient Sharing of Distributed Resources, Second International Conference on Software Engineering, October 1976, pp562-570

[2] T. Becker, Application-Transparent Fault Tolerance in Distributed Systems, Proceedings Second International Workshop on Configurable Distributed Systems, March 1994, pp36-45

[3] K. Birmen, T. Joseph: Reliable Communications in the Presence of Failures, ACM Transactions on Computer Systems, 5:1, 1987, pp47-76

[4] R. Blanchini, R. Buskens, An Adaptive Distributed System-Level Diagnosis Algorithm and its Implementation, Fault-Tolerant Computing: Twenty-First International Symposium, June 1991, pp222-229

[5] F. Cristian, Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System, Eighteenth International Symposium on Fault-Tolerant Computing, FTCS-18, June 1988, pp206-211

[6] M. Davidson, Failure Detection for Somersault Distributed systems, Masters thesis, MIT

[7] E. N. Elnozahy, D. B. Johnson, W. Zwaenepoel, The Performance of Consistent Checkpointing, Proceedings 11th Symposium on Reliable Distributed Systems, October 1992, 39-47

[8] M. Fischer, N. Lynch, M. Paterson, Impossibility of Distributed Consensus With One Faulty Process, Journal of the ACM, Vol. 32, No. 2, April 1985, 374-382

[9] S. H. Hosseini, J. G. Kuhl, S. M. Reddy, On Self-Fault Diagnosis of the Distributed Systems, Fifteenth Annual International Symposium on Fault-Tolerant Computing FTCS 15, June 1984, pp30-35

[10] K. L. Jeffrey, P. Naughton, J. S. Plank, Low-Latency, Concurrent Checkpointing for Parallel Programs, IEEE Transactions on Parallel Distributed Systems, 5:8, August 1994, pp874-879

[11] K. H. Kim, H. Kopetz, K. Mori, E. H. Shokri, G. Gruensteidl, An Efficient Decentralised Approach to Processor-Group Membership Maintenance in Real-Time LAN Systems: The PRHB/ED Scheme, Proceedings 11th Symposium on Reliable Distributed Systems, October 1992, pp74-83

[12] J. Lyon, Tandem's Remote Data Facility, Thirty-fifth IEEE Computer Society International Conference. Intellectual Leverage, Feb. 1990, pp562-567

[13] MC ServiceGuard Product Breif, http://www.hp.com/gsy/high_availability/mcsg_brief.html

[14] A. Ricciardi, A. Schiper, K. Birman, Understanding Partitions and the "No Partition" Assumption, Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems, September 1993, pp354-360

[15] L. S. Sabel, K. Marzullo, Simulating Fail-Stop in Asynchronous Distributed Systems, 13th Symposium on Reliable Distributed Systems, October 1994, 138-147

[16] R. D. Schlichting, F. B. Schneider, Fail-stop Processes: An Approach to Designing Fault-tolerant Computing Systems, ACM Transactions on Computer Systems, 1:3, August 1983, pp222-238

[17] F. B. Schneider, Byzantine Generals in Action: Implementing Fail-Stop Processors, ACM Transactions on Computer Systems, 2:2, May 1984, pp145-154

[18] C.-L. Yang, G. M. Masson, Hybrid Fault Diagnosability with Unreliable Communication Links, FTCS: 16th Annual International Symposium on Fault-Tolerant Computing Systems, July 1986, pp226-231