Somersault: Enabling Fault-Tolerant Distributed Software Systems

Paul T. Murray, Roger A. Fleming, Paul D. Harry, Paul A. Vickers

Hewlett-Packard Laboratories

Filton Road, Bristol BS34 8QZ, U.K.

{ptm,raf,pdh,pav}@hplb.hpl.hp.com
Tel: (+44) 117 979 9910
Fax: (+44) 117 922 8003
http://www-uk.hpl.hp.com/

Abstract

Somersault is a platform for developing distributed fault-tolerant software components and integrating these critical components with other components into distributed system solutions. Critical application processes are mirrored across a network, with each critical process being replicated in a primary and secondary. Replication of processes and recovery from the failure of a replica are handled transparently. Somersault provides a fault-tolerant communication transport protocol, which can be plugged into an Object Request Broker, a combination which achieves replication transparency.

Somersault has been developed at Hewlett-Packard Laboratories, and runs on standard operating systems such as Unix and NT. Somersault has been through two iterations of algorithm design, implementation and optimisation in order to achieve high message throughput and fast recovery from failure. At the time of writing it is undergoing industrial trials and detailed performance testing.

1 INTRODUCTION

The demand for highly available systems, operating 24 hours a day and 7 days a week, is extending beyond the traditional areas of telecommunications and banking. Information Technology systems, for both internal operations and customer-facing services, are becoming mission-critical to more and more enterprises.

High availability software technology for *centralised* systems, typically based on clusters of computers, is well understood and in widespread deployment, eg [5], [6].

Increasingly, *decentralised* approaches, i.e. distributed systems, are being used in mission-critical systems. However one of the main challenges to the uptake of distributed systems is high availability. Distributed applications are generally perceived to be less reliable than centralised applications. Industry efforts to apply distributed computing, such as OMG [11] and TINA-C [12], have identified the requirement for fault-tolerant distributed computing software technology.

In theory, one of the benefits promised by distributed software systems is higher availability. System components can be replicated, and the replicas can be geographically distributed, to enable continued system operation in the event of component failure.

This paper describes a middleware technology which aims to realise this promised benefit of distributed systems. Our aim is to support the distributed application programmer by offering powerful high availability abstractions, enabling the programmer to concentrate on the application.

1.1 High Availability Approaches

High Availability (HA) obeys one of the truisms of distributed systems: “One size does not fit all”. Different HA solutions fit different application classes. Our requirements and design goals are motivated by very demanding applications such as telecoms, finance, electronic business and air traffic control :-

- Continuous Availability i.e. service interruption on failure is measured in seconds, and recovery is fully automated
- High message throughput i.e. up to thousands of messages per second
- Need to maintain consistent state and guarantee message delivery.

1.2 The Somersault Approach: Process Mirroring

Our approach is to replicate at the process level by maintaining a hot standby replica i.e. a primary process and secondary process. We characterise this as a roll-forward approach (hence the name “Somersault”), as compared to the roll-back approach of restarting from disk. We use the term “Process Mirroring”, by analogy to Data Mirroring. Process mirroring allows applications to survive hardware failures and operating system failures. Application failures that are not replicated can also be survived (replicated code bugs such as divide by zero cannot be masked).

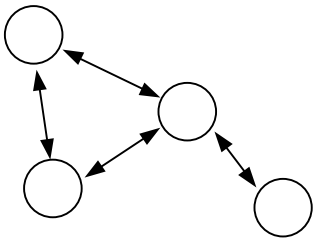
Process Mirroring implies the need to manage each pair of replica processes, including detecting failures and recovering from failures, and to maintain identical computation and consistent state in each pair of processes. Somersault provides the infrastructure to do all of these tasks automatically in the form of a C++ middleware library. The library is fully functional in its own right, or can be integrated with CORBA to offer a simpler programming model.

Section 2 describes the “insides” of the Somersault approach to replication and reliable messaging. Section 3 outlines how the system is used i.e. how it looks from the “outside”. Section 4 describes the use of Somersault with CORBA. Section 5 discusses the performance and scalability characteristics of the approach. Section 6 makes comparisons with related work and section 7 concludes.

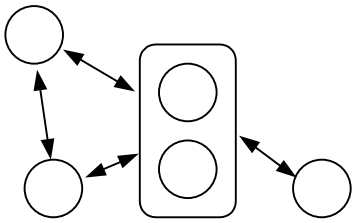
2 SOMERSAULT ON THE INSIDE

This section describes the approach, to the depth of understanding needed by fault-tolerant system designers using the Somersault platform. A full description of the algorithms and the implementation is outside the scope of this paper.

A regular distributed system has processes that communicate using a process-to-process protocol. A Somersault system has units that communicate using a unit-to-unit protocol. The Somersault units are of two types: a single non-fault-tolerant process is called a “simple unit”; and a collection of replicated processes is called a “recovery unit” (see figure 1). Processes within a recovery unit will be distributed across a network to achieve tolerance of faults.



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

Both simple units and recovery units are addressed as single entities and are indistinguishable from a messaging standpoint. The Somersault messaging protocol is a general n-to-m connection-oriented transport allowing an n-process unit to connect to and communicate with an m- process unit.

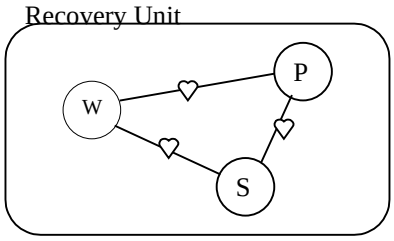
The minimum recovery unit has three processes: the Primary, Secondary and Witness. Only the primary and secondary replicate the application. All of these processes are involved in failure detection, the witness acting as a tie-breaker.

Whilst the Somersault messaging protocol supports the general case of n application replicas, there are diminishing returns from using more than 2 replicas (and growing costs in increased usage of computer cycles). This paper will describe the 2-replica case, which is the one our implementation supports. Our examples will illustrate a simple unit, typically a non-fault-tolerant client, communicating with a recovery unit, typically a fault-tolerant replicated server.

The following subsections describe how Somersault :-

1. Detects failures within a recovery unit by the use of heartbeats and a unit membership protocol
2. Replicates processes within a recovery unit, ensuring computation is mirrored
3. Handles communication between units, via a reliable messaging protocol with strong semantic guarantees
4. Handles failover i.e. deals with messages and connections at the time of a failure
5. Handles recovery i.e. rebuilds the unit after a failure and continues processing.

2.1 Failure Detection



Processes in a recovery unit use heartbeats for failure detection. Group membership consensus is enabled by a Witness (W)

Figure 2 Failure Detection in a Recovery Unit

The processes of a unit pass heartbeat messages to one another. Suspicion is raised over the health of a process when its heartbeat is missing. In this case a unit membership consensus protocol is invoked to decide a new unit membership. There are various unit membership protocols that could be used e.g. those reported in [3] and [14]. We use a simple quorum protocol, with the witness avoiding the "split-brain problem" because an odd number of processes take part in the voting. For the rest of this paper we will ignore the witness process.

2.2 Replication : Process Mirroring

We model an application process as a state machine driven by non-deterministic events. We assume that the only observable behaviour of a process is its message communication. Two copies of the same process will be replicas, if they:

1. Receive the same input messages in the same sequence.
2. Experience in the same sequence the same non-deterministic events e.g. reading a real-time system clock, e.g. calling a random number function.

Somersault ensures that the process pair act as replicas by allowing the primary to perform non-deterministic events and then making the secondary perform identically. This is achieved by a logging channel between the primary and secondary, implemented as a messaging connection, shown in Figure 3. Non-deterministic events are logged on the channel and fed to the secondary process in the order they occurred at the primary.

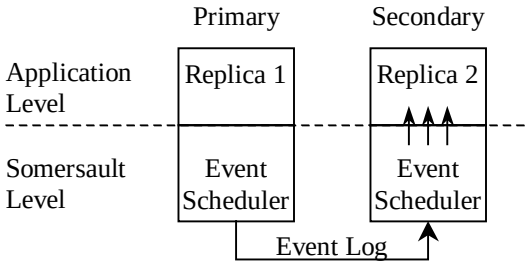


Figure 3 Non-Deterministic Event Logging

There are two types of non-deterministic event :

1. Those that can occur at any time, e.g. a timer expiring or an input message. The Somersault event scheduler controls the delivery of timeouts and messages to the application, and it controls thread execution.

2. Those that occur at a specific point in the execution of the application code, e.g. system calls, e.g. a random number function. We call this type Non-Deterministic Choice. Somersault captures the result of a non-deterministic choice in the primary and injects it at the secondary. This requires the application programmer to make non-deterministic choices explicit.

2.3 Unit Communication: Secondary Sender Protocol

The Somersault communication protocol provides unit-to-unit asynchronous messaging with the properties that all replicas consume input messages in the same order; and all replicas generate output messages, but only one copy of each output message is sent.[1]

[1] We will use two sets of terms for talking about messaging. “Send” and “receive” refer to the actual transfer of a message between processes across a network. Send and receive will always be performed by Somersault. “Consume” and “generate” refer to the exchange of application messages between Somersault and the application level. Somersault delivers input messages for the application to consume. The application generates output messages for Somersault to send. All of the application replicas in a unit consume and generate messages.

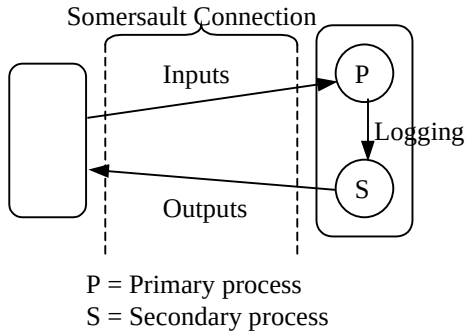


Figure 4 Somersault connection

A connection between two units is implemented using two reliable first-in-first-out network connections: one for receiving messages and one for sending them, as shown in Figure 4. Input messages are sent on the connection to the primary. Non-deterministic events, including the inbound messages, are logged to the secondary. Output messages are sent on the connection from the secondary: it is this characteristic which gives the protocol its name, “secondary sender”.

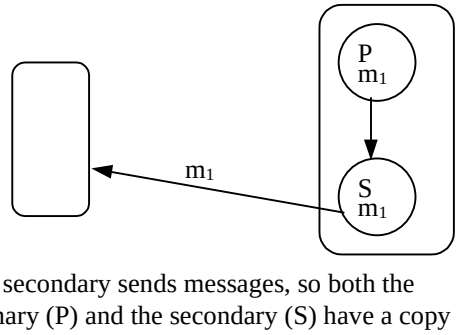
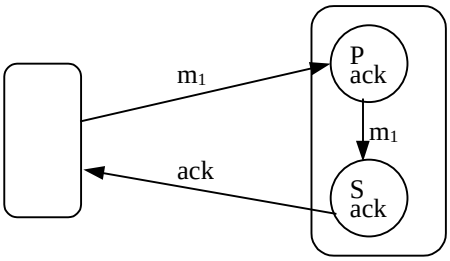


Figure 5 Message Send

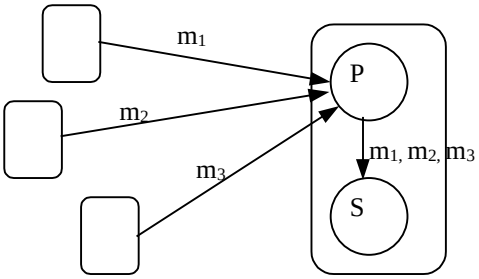
If a recovery unit sends a message it is sent by the secondary, as shown in Figure 5. Process mirroring ensures that the secondary reflects the primary, so both the primary and secondary will generate the message, although not necessarily at the same time.



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

Figure 6 Message Acknowledgement

Figure 6 shows that acknowledgements, as with output messages, are only issued by the secondary. The receipt of an acknowledgement is thus an assurance that the messages acknowledged have been received at both replicas.



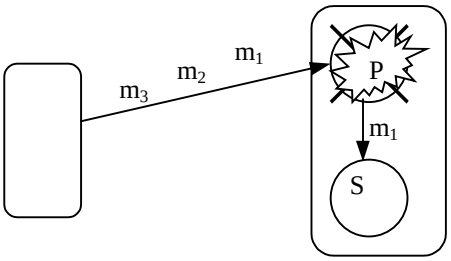
The primary (P) logs m1, m2, m3 to the secondary (S) in the order that they were consumed

Figure 7 Message Order

All input messages are received by the primary and logged to the secondary, see Figure 7. Message consumption is a non-deterministic event, but process mirroring ensures that the secondary will consume messages in the same order as the primary.

2.4 Failover

The secondary sender message protocol ensures that connections are re-established as required and that messages are not lost or re-ordered in the event of the primary or secondary failing. It does this using a windowing protocol that works at the level of unit-to-unit communication. Messages are buffered at the primary and secondary and resent as required when a process is lost.



The primary (P) fails after receiving messages m_1, m_2, and m_3, but only managed to log m_1

Figure 8 Lost State On Primary Failure

In the case of primary failure it is possible that some non-deterministic events never make it across the logging channel to the secondary, see Figure 8. The primary may have performed some work and generated some outputs that the secondary never did. After failover the secondary will receive the same input messages and repeat the work, possibly with a different outcome due to non-deterministic events that have not been replicated, including the message receipt order. This is not a problem because only messages sent by a unit (output by the secondary) are externally visible.

2.5 Recovery

If the primary failed, the secondary will be promoted to be the primary. The first step of creating a new secondary is to create a process that joins the recovery unit. It must now become a replica. This involves copying runtime state from the primary 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 becomes the sender. The unit is now back to full strength. The whole procedure preserves the unit messaging properties.

Note that the secondary can be rebuilt in the background i.e. the primary will continue to provide service.

3 SOMERSAULT FROM THE OUTSIDE

3.1 Programming Model

To use Somersault, the system must be analysed to identify which units need to be fault-tolerant, and designed as a set of communicating units. Somersault takes care of replication and failure recovery if the following programming rules are followed for recovery units (e.g. a fault-tolerant server):

1. Use Somersault messaging for communication.
2. Use the Somersault mechanism for making non-deterministic choices explicit.
3. Provide a state transfer procedure for initialisation of a new process (at failover time).

When programming a simple unit (e.g. a non-fault-tolerant client), only programming rule 1 applies:

1. Use Somersault messaging to communicate with recovery units.

Using Somersault messaging is a simple programming task: remember that the replication is transparent to the programmer. In our experience, programming non-deterministic choices has also been easy, and programming state transfer has been the greatest burden on the programmer (although a much lesser burden than building a replication system !).

3.2 Application Experience

Non-deterministic choices tend to relate to the use of time or physical resources, e.g. timestamps and dynamically allocated memory. Many simple applications don't have non-determinism, especially if they use statically allocated resources.

State transfer can be an issue if the state is (a) large e.g. a 500Mb main memory database will take time to transfer; or (b) not neatly stored in data structures or objects that can be easily traversed. However such applications are intrinsically difficult to make fault-tolerant.

Examples where we have found a good fit to Somersault include main memory data stores, lock managers, message queues, stock feeds, name servers etc. At the time of writing, Somersault is undergoing industrial trials.

In some of our application experiments, the solution features a gateway between the Somersault world, within which communications must use the Somersault protocol, and the non-replicated world. The next section will explain how CORBA can be used to provide a transparent solution which does not require a gateway.

4 INTEGRATING CORBA WITH SOMERSAULT

4.1 Somersault as a Fault-Tolerant ORB Transport

CORBA provides a communication abstraction that insulates the programmer from details of distribution – including the transport protocol. Somersault is implemented as a communication protocol and can be integrated under an ORB as a fault-tolerant transport protocol. The programmer then programs to the ORB interface.

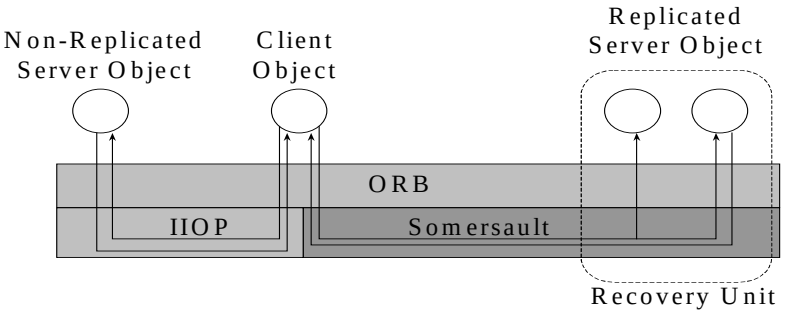


Figure 9 Method invocation on Replicated and non-Replicated Server

We have verified this approach using ORBLite, an experimental ORB developed at Hewlett-Packard Laboratories [10]. ORBLite is able to dynamically select transport protocols at runtime. We used this facility to dynamically select Somersault if the communication involves a replicated object. The client shown in Figure 9 can invoke methods on replicated and non-replicated objects. For the replicated objects Somersault is selected, for the non-replicated objects the default transport, IIOP, is selected. The object is not aware of transport selection.

4.2 Replication Transparency

Programming rule 1 from section 3.1 (Somersault communications) is handled transparently by the ORB. Programming rules 2 and 3 (non-deterministic choices, state transfer) still have to be followed for recovery units, but simple units have complete replication transparency. An existing server which uses CORBA can be replaced by a fault-tolerant version with no change to the client code at all.

Somersault also protects the programmer from as many of the replication details as possible in programming a fault-tolerant CORBA server.

4.3 Server Example

We will use a simplified example to illustrate the usage of Somersault. The example is a password server which has two functions: it can assign a randomly generated password to a user, and it can validate passwords. The server maintains a mapping of user identifiers to passwords in a Map data structure. A replicated password server will have to keep the map data consistent and generate consistent random numbers. We choose the password example because it contains non-determinism (the random number function) and vital state (the password map). A C++ definition of the password server class may look like the following:

```
class Password
{
public:
        int generate( int user_id );
        int validate( int user_id, int pw );
private:
        Map password_map;
}

int Password::generate( int user_id ) {
        int pw = rand();
        password_map[user] = pw;
        return pw;
}

int Password::validate( int user_id, int pw ) {
        return ( password_map[user] == pw );
}
```

4.3.1 Non-Deterministic Choice

The programmer creates an object for performing non-deterministic choices on behalf of the replicated object. The choice object is present at both the primary and secondary replica and uses a special non-deterministic choice object adapter provided by Somersault. Our password example uses a random number generator called rand(). We could define the following implementation for the non-deterministic choice object.

```
class GenerateChoiceImpl:
        public virtual _BOA_GenerateChoice,     // created from idl
        public NonDet                           // special object adapter
{
public:
        int generate();
};

int GenerateChoiceImpl::generate() { return rand(); }
```

The primary and secondary make the same distributed call to the generate() method of GenerateChoiceImpl using its object handle. The NonDet object adapter at the primary will call the method, return the result and send the result to the NonDet object adapter at the secondary (via the Somersault logging channel). The NonDet object adapter thus returns the same result to both replicas.

4.3.2 State Transfer

State transfer is handled similarly to non-deterministic choices. The programmer creates an object for performing state transfer. The state transfer object is present at both the primary and secondary replica and uses a special state transfer object adapter provided by Somersault. The object must have a do_transfer() method which will be invoked by the Somersault transport when it is time to do the transfer. The programmer can then define methods for building state from parameters.

In the password server example, the state is held in a map structure. We could define the following implementation for the state transfer object.

```
class Password
{
friend class PasswordStateXferImpl;
…
}

class PasswordStateXferImpl:
        public virtual _BOA_PasswordStateXfer,// created from idl
        public StateXfer                      // special object adapter
{
public:
        PasswordStateXferImpl( Password& password_object );
        void do_transfer();
        void build_map( Map pw_map );
private:
        Password& password;
};

void PasswordStateXferImpl::build_map( Map pw_map ) {
        password.password_map = pw_map;
}

void PasswordStateXferImpl::do_transfer() {
        PasswordStateXfer remote_handle = _self();
        remote_handle.build_map( password.password_map );
}
```

In this code, password is the local copy of the password. We have defined PasswordStateXferImpl as a friend of Password so it can access all its data members. When do_transfer() is executed at the primary it does a distributed call to its own build_map() method. Instead of invoking the call locally, the modified object adapter invokes the call on the corresponding object at the secondary. So invoking do_transfer() at the primary calls build_map() at the secondary, passing the state as a parameter.

5 PERFORMANCE AND SCALABILITY

5.1 Failover Time

An important measurement is the time that a recovery unit is out of action when one of its replicas fails i.e. the time until the unit resumes application processing. We have measured failover times of less than one second. As discussed in [4] the failover figure is limited by a realistic heartbeat interval. If the heartbeat is set too small, unnecessary failovers will occur because the processes or network are not timely enough to stick to the deadline.

5.2 Message Throughput

We have measured performance up to 3000 requests per second at a replicated server in fail-free operation. The figures were obtained using HP9000 C class 160Mhz workstations running HP-UX 10.20, with a dedicated 100Mbit Ethernet network. This was using vanilla Somersault without the ORB.

The secondary sender protocol establishes a pipeline in the recovery unit, allowing the primary and the secondary to process method invocations in parallel. Theoretical analysis shows that we can expect the replicated server to achieve 2/3 the throughput of the non-replicated server when network bound. This is due to the increase in messages. This was confirmed in our work with ORBlite, where the performance was approximately 2/3 that of the ORB. When not network bound, we have achieved even better throughput.

5.3 Configuration and Scalability

Fault-tolerant systems survive failures by means of redundancy, which implies an additional cost. The main cost of using the Somersault approach is a doubling of the number of CPU cycles for those critical processes that are replicated. In any one system using the Somersault platform, there will be a number of critical components (these will typically require double the number of machines that would be used otherwise) and a number of non-critical components.

The Somersault infrastructure facilities are themselves distributed, indeed designed to allow the system to scale to a large number of recovery units distributed over a wide area network. Somersault can be used in systems where some components are not replicated, where some components are replicated in local clusters of machines, where some components are replicated across geographically distributed machines.

Configuration is a design issue for fault-tolerant systems, and the designer has complete flexibility in choosing how to distribute replicas. At any one instant, any machine can be running any number of primary and secondary processes from different recovery units, although it would be a poor design to have both primary and secondary from any one recovery unit running on the same machine. The Somersault infrastructure will make sensible default choices from a list of machines to launch a secondary.

Somersault recovery units can survive network partitions to the optimal degree (it is not possible to continue communications across a network partition). Recovery units in which the primary and secondary were on opposite sides of the network partition will handle this situation in the same way as a hardware or software failure.

6 RELATED WORK

Maffeis and Schmidt [9] identify alternative HA approaches in distributed systems, viz. Message Queues, Transaction Processing Monitors and Virtual Synchrony. Some ORB vendors provide Transaction Processing Monitors e.g. [7].

Somersault falls into the Virtual Synchrony category, of which a well-known example is the Isis toolkit [2], which has been a major influence on Somersault. Isis is a general-purpose group communication toolkit from which fault-tolerant applications can be built. A selection of protocols provide the primitive properties along with failure detection and group membership, but it is up to the programmer to use those protocols to build a fault-tolerant system. Somersault is designed and optimised to offer the programmer a library which supports process replication, and enables the programmer to compose fault-tolerant units.

Some ORB and middleware vendors support replication of object services and client rebinding on failure e.g. [13]. These approaches typically do not provide the strong guarantees offered by Somersault, but are well-suited to applications where maintaining consistent state is not an issue.

Somersault makes a contribution to the study of replication transparency in distributed systems which was identified as an architectural goal by ANSA [1] and ODP [8], and is also a goal of an OMG Request for Proposals for Fault-Tolerant CORBA [11].

7 CONCLUSIONS

High Availability is very much an application-dependent issue. We have described one approach which is a candidate for distributed message-based systems with very demanding requirements, typically encountered in telecommunications, banking, electronic business, and air traffic control.

Somersault automates process replication, failure detection and failure recovery. It can be plugged into an ORB to realise the goal of replication transparency, at least on the client side. Somersault enables programmers to realise the promised high availability benefit of distributed systems.

Somersault has been through two major iterations of algorithm design, implementation and optimisation. Throughput has been measured up to several thousand messages per second in fail-free operation, and in test applications service interruption times on failure of less than one second have been achieved. More detailed performance results will be available in the middle of 1998.

8 REFERENCES

[1] An ANSA Analysis of Open Dependable Distributed Computing, N.J. Edwards, ICL Technical Journal 9(2), Nov 1994, pp218-240.

[2] K.P.Birman and R.van Renesse, eds, Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, 1994.

[3] 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, Jun 1988, pp206-211

[4] M. Davidson, Failure Detection for Somersault Distributed systems, Masters thesis, Massachusetts Institute of Technology, May 1995

[5] HP Multi-Computer/ServiceGuard, http://www.hp.com/esy/go/ha.html

[6] IBM Clustering Solutions for RS/6000, http://www.austin.ibm.com/software/Appfinder/clustering.html

[7] Iona Technologies Orbix Object Transaction Monitor, http://www.iona.ie/otm.html

[8] ITU-T Recommendation X.902, Basic Reference Model of Open Distributed Processing – Part 2 – Descriptive Model, Jun 1993.

[9] S. Maffeis and D.Schmidt, Constructing Reliable Distributed Communication Systems with CORBA, IEEE Communications Magazine, Feb 1997.

[10] K. Moore and E. Kirshenbaum, Building Evolvable Systems: The ORBLite Project, Hewlett-Packard Journal, Feb 1997

[11] Object Management Group, Fault Tolerant CORBA, Request For Proposals, http://www.omg.org/docs/orbos/98-03-05.pdf, Feb 1998

[12] TINA Consortium, Distributed Processing Environment Architecture, Dec 1994, http://www.tinac.com/95/dpe/emc94-public.pdf

[13] VisiBroker information, http://www.visigenic.com/prod/vbrok/vb30DS.html

[14] C.-L. Yang, and G. M. Masson, Hybrid Fault Diagnosability with Unreliable Communication Links, IEEE Transactions on Computing, vol. 37, no. 2, Feb. 1988, pp175-181