	DSoS ***IST-1999-11585*** ***Dependable Systems of Systems***

Failure analysis of an ORB in presence of faults

Report Version: Deliverable IC3

Report Preparation Date: 1 October 2001

Classification: Public Circulation

Contract Start Date: 1 April 2000 **Duration:** 36m

Project Co-ordinator: Newcastle University

Partners: DERA, Malvern – UK; INRIA – France; CNRS-LAAS – France; TU Wien – Austria; Universität Ulm – Germany; LRI Paris-Sud - France

Project funded by the European Community under the "Information Society Technology" Programme (1998-2002)

Failure analysis of an ORB in presence of faults

Eric Marsden and Jean-Charles Fabre
LAAS-CNRS, Toulouse, France
{emarsden,fabre}@laas.fr

Abstract

This document describes a method and experimental results for the dependability characterization of middleware implementations, and in particular failure mode analysis of CORBA ORB implementations. The aim of the work is to provide an overall approach for identifying and quantifying failure modes using various fault injection techniques and fault models.

Related work in dependability characterization of executive software layers is discussed. An analysis of the architecture of middleware-based systems and their error confinement regions motivates the development of a fault model. A number of fault injection approaches are discussed, and results from network-based corruption experiments targeting four CORBA service implementations are presented.

Table of Contents

1 Introduction

This document describes a method for characterizing the dependability of middleware implementations, in particular, the failure modes of CORBA ORB implementations. The aim of the work is to provide an overall method to identify and quantify failure modes using various fault injection techniques and fault models.

The method is based on existing work in dependability characterization. Although a large amount of work has been carried out for the characterization of executive software using fault injection (e.g., kernel and standard operating system), very little work has targeted the middleware layers. The current work in the field, in particular software-implemented fault injection techniques and tools, is reviewed in this document because of its interest for the characterization of CORBA-based systems.

The conventional software engineering view of a middleware is not sufficient when considering the assessment of dependability-related properties. We present a more detailed architectural view of a CORBA-based system, concentrating on the identification of error confinement regions and a failure modes classification. Based on this structural analysis, we discuss the classes of faults that can affect such systems.

A number of fault injection techniques that simulate these fault classes are described. We present results from one of these techniques, which measures the impact of corrupt method invocations on a middleware implementation. These experiments are particularly relevant to the DSoS project, since they provide insights into the ways in which errors may propagate between component systems, over the interconnection infrastructure. Experimental results we have obtained on a number of CORBA implementations reveal that this impact can be significant.

The document is organized as follows: In Section 2, we discuss the objectives of our work within DSoS, and give an overview of dependability assessment techniques and of CORBA-based middleware.

In Section 3, we discuss some previous work in fault injection that is related to the failure mode analysis of CORBA systems. Some results of experiments targeting CORBA are also reported in this section.

In Section 4, we address the failure modes in middleware-based systems. This involves discussing the architecture of a middleware system from different viewpoints and identifying possible targets, defining fault assumptions and models, classification of failure modes and the identification of possible fault injection strategies.

Section 5 is devoted to the description of our method for experimentally characterizing the dependability of CORBA ORB implementations. A number of fault injection techniques that are well suited to this target are described.

Section 6 reports on the experimental results we obtained when targeting CORBA service implementations using network corruption techniques. In the last section, we present some first lessons that have been learned from our work, from the viewpoint of a DSoS system integrator, and draw some conclusions.

2 Objectives and enabling technologies

In this section we present the aim of our work on middleware assessment with respect to the DSoS project, and give an overview of dependability assessment techniques. We conclude with a brief introduction to CORBA middleware.

2.1 Middleware assessment for DSoS

An increasingly large class of dependable systems of systems will be built on some form of middleware infrastructure. A likely candidate for this middleware infrastructure is the CORBA platform, a standard that is well suited to the interconnection of heterogeneous systems, and is widely used in industry. Figure 1 illustrates possible roles for CORBA-compliant middleware in a system of systems, both as a technology for the implementation of linking interfaces, possibly using wrapping techniques, and as a means of interconnecting component systems. Evidently, the dependability of this middleware layer is crucial to the dependability of the DSoS built above it.

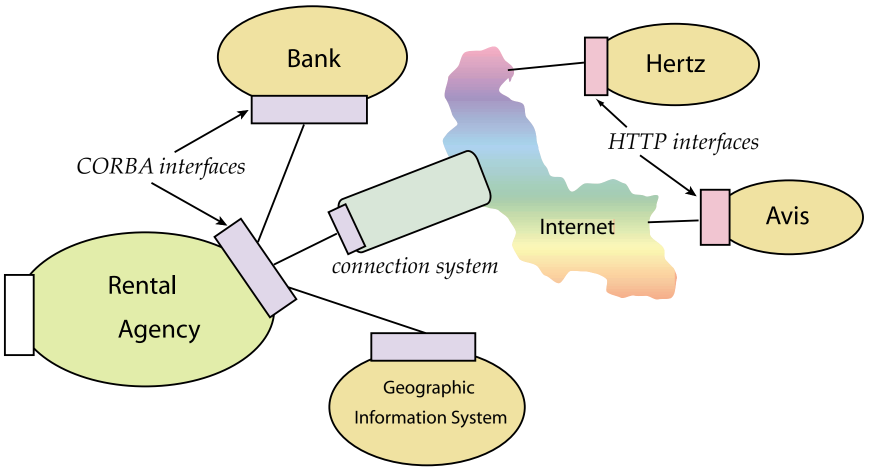

Figure 1: CORBA infrastructure for a system of systems

There has been little published research on the dependability of middleware-based systems. The DSoS project aims to contribute to this issue in two ways:

- categorize and study the types of faults which can be experienced by a middleware-based system, and the ways in which errors propagate through the system to cause failure.
- design and implement methods to improve a middleware implementation's behaviour in the presence of these faults: improve error detection

mechanisms, fail silence and robustness characteristics, using techniques such as wrapping.

This report contributes to the first point. We present a number of approaches for characterizing the behaviour of a middleware-based system (and in particular a CORBA-compliant ORB implementation) in the presence of faults, and present the results of fault injection experiments targeting several implementations of the CORBA Name Service.

The second point is addressed in the *Architecture and Design* workpackage, which includes work on the use of wrapping techniques to improve the robustness and fail silence of component systems, and of the interconnection infrastructure. The aim is to ensure that individual component systems have a well-defined behaviour in the presence of faults. The *Architecture and Design* work is closely related to the present deliverable; indeed, the failure mode characterization is an essential input to the design and development of fault containment wrappers.

Our work on fault injection also provides insights into the manners in which errors may propagate between component systems, via the connection systems. It provides a mechanism for evaluating the probability of such error propagation, and for characterizing their effects. It also investigates the impact of faults affecting the communication subsystem itself. This work helps the integrator of a system of systems answer two important questions:

- what type of dependability properties can be assumed of the interconnection infrastructure?
- how might the dependability of a component system be affected by the addition of DSoS-related software, implementing a linking interface?

2.2 Dependability assessment techniques

The dependability of computer systems can be assessed using either model-based or measurement-based techniques. Modelling work allows system designers to obtain predictions of the dependability attributes of a system, based on probabilistic measures of the behaviour of its subsystems. These measures are useful during the design phase, since they enable the pertinent dependability attributes of different system configurations to be estimated, even before they are built. There is work in the *Validation* workpackage that addresses this problem from a DSoS point of view.

However, modelling techniques can only provide predictions of the dependability attributes of a system. Once a system has been implemented and deployed, measurement-based techniques can be applied to obtain more specific insights and measures. There are two main measurement-based approaches to obtaining information on a system's dependability:

- the observation of a large set of systems in operation, as in[Kalyanakrishnam *et al.*, 1999]. This approach relies on error information obtained either from logs maintained by system administrators or from automatic monitoring mechanisms provided by the system. By analysing the data, one can obtain information on the nature and frequency of failures, and on the type of usage that led to the failure of the component system.

  A disadvantage of this approach is that failures are rare, which makes it necessary to collect information on a large population of identical systems over a large time span before being able to make statistically significant analyses. It is thus poorly suited to short development cycles.

- the deliberate insertion of faults into the target system, so as to accelerate the characterization of its behaviour in the presence of faults. These *fault injection* experiments allow the system's error detection mechanisms to be triggered more frequently than in normal operation. They also allow evaluation of the system's behaviour when error detection coverage is not perfect, as is usually the case for complex systems.

Field-based observations are complementary to fault injection experiments, since they provide data on types of failures that can be experienced by a system, in given operational conditions. The analysis of failure reports can be used to derive a fault model, which is then used to develop fault injection campaigns. This increases the likelihood that fault injection experiments are representative of real faults experienced by the target system.

Unfortunately, there has been no reported work on field observation of failures in middleware-based systems. This makes it difficult to validate the degree of representivity of a given fault model for these targets.

2.3 Fault injection

Fault injection is a well-known dependability characterization technique [Arlat *et al.*, 1993], which studies a system's reaction to abnormal conditions. It is a testing approach that is complementary to analytical approaches, and which allows the examination of system states which would not be reached by conventional functional testing. The aim of fault injection is to simulate the effect of real faults impacting a target system, namely the error due to the activation of a fault.

Fault injection experiments provide a number of useful results:

- an understanding of the system's failure modes, or its behaviour in the presence of faults;

- information on the fault tolerance mechanisms in the target system, in particular a measurement of their coverage (the conditional probability that, given a fault in the system, the system can tolerate it).

A number of fault injection techniques have been developed. Most early work concerned the injection of physical faults [Karlsson *et al.*, 1998], radiating electronic circuits with heavy ions, to simulate the effect of electromagnetic radiation, or acting directly on the pins of a microprocessor to modify voltages.

Due to the complexity and the speed of modern integrated circuits, recent research has concentrated on software-implemented fault injection (SWIFI). In this technique, the corruption is performed by software, and can target different components or different layers in a system (operating system kernel, system services, middleware components, application code, system memory and registers). This approach is very generic and flexible, since a large variety of fault models can be used. Several studies have shown that a single bit-flip leads to similar errors to those produced by physical fault injection techniques (e.g.,[Rimén *et al.*, 1994, Fuchs, 1998]), and also that they simulate errors produced by software faults fairly faithfully [Madeira *et al.*, 2000].

The target for the fault injection can either be the interface of a software component, or its internal address space. Targeting the interface assesses the component's *robustness*, its ability to function correctly in the presence of invalid inputs and stressful environmental conditions. It is a useful way of evaluating the probability of error propagation from one system component to another, due to their interactions. Targeting the address space assesses the impact on the component's behaviour of internal corruptions, resulting either from physical faults or software faults.

2.4 Targeting CORBA middleware

Middleware is software that mediates between an application program and the network. It manages the interaction between disparate applications across heterogeneous computing platforms, abstracting from the programming language, operating system and hardware, and often providing convenient access to services such as naming, transactional processing and concurrency management.

CORBA [OMG, 2001a] is a middleware platform that focuses on interactions between distributed objects. The standard is defined by the *Object Management Group* (OMG), an industry consortium. A key principle of CORBA is its separation of interface and implementation. Interfaces are used to specify the operations and data types that allow access to a service; they are described in an *Interface Definition Language* (OMG IDL). An interface is independent of the programming language and operating system that is used to implement the service it describes, and of the location where the service is provided. The software that transports service requests between the client and the server is called the *Object Request Broker* (ORB).

Figure 2 shows a client invoking a method named `foo` on an object hosted on a remote computing node. The object's semantics are implemented by a programming-language dependent entity called a *servant*. The figure shows the different functional elements composing a CORBA middleware implementation:

- the core of the ORB, which handles marshalling of information to and from the CORBA wire format, and communication with ORBs on remote nodes, including request demultiplexing and concurrency management;
- a set of CORBAservices, including naming, trading, event propagation, access control and persistency;
- on a server, an object adapter (OA) that is responsible for dispatching incoming requests to the appropriate servant, controlling security issues and the lifecycle of servants;
- client *stubs* that provide an interface to the ORB core, and implementation *skeletons* that connect the object adapter to the servant (these elements are programming language dependent, and are generated automatically from the IDL interface);
- modules that handle dynamic invocation, providing an invocation mechanism that can be used when the interface of a service was not known at compile-time (DII[1] and DSI[2] modules).
- an optional *Interface Repository* service, which allows runtime introspection of the IDL interfaces available in the system.

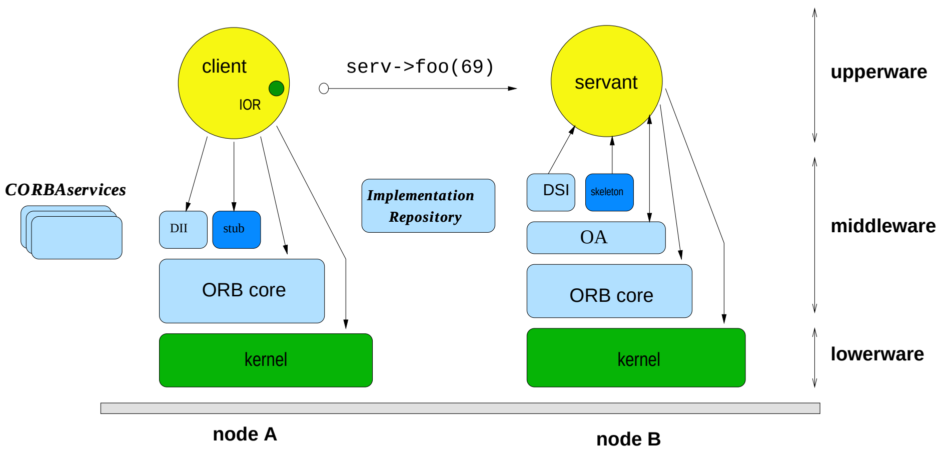

Figure 2: High-level view of a CORBA method invocation

[1]DII: Dynamic Invocation Interface, that allows clients to construct method invocations without passing through a stub

[2]DSI: Dynamic Skeleton Interface, that allows servers to handle incoming dynamically constructed requests.

ORB implementations from different vendors, running on different platforms, are able to interoperate by exchanging messages adhering to the General Inter-ORB Protocol (GIOP). This specification describes the data representations, message types and message formats to be used for communication between ORBs. GIOP assumes that the underlying transport protocol is connection-oriented, reliable, and can be viewed as a byte stream. The mapping of GIOP onto TCP/IP is called the Internet Inter-ORB Protocol (IIOP).

3 Related work

In this section, we describe previous research on fault injection for dependability characterization. We concentrate on work that has targeted middleware implementations, as well as executive software such as operating system kernels and network protocol stacks. We classify fault injection techniques according to whether they simulate faults that originate internally to a component system, or whether they originate via its linking interface or its local interfaces [Jones *et al.*, 2001].

3.1 Faults internal to a component system

A fault is said to be *internal* when the corruption of the system's state that it causes originated inside the system. This encompasses programming errors that may affect the internal data of the system, and hardware faults that may corrupt both data and code memory segments. The most common fault model used is the single bit-flip.

A large number of tools have been developed to automate the execution of experiments using this fault model. Two significant examples are *Xception*, developed at the University of Coimbra, Portugal [Carreira *et al.*, 1998], and *MAFALDA*, developed at LAAS-CNRS, France [Fabre *et al.*, 2000].

There has been some work [Chung *et al.*, 1999] investigating the impact of high-level faults on CORBA and DCOM applications. The faults simulated are hangs and crashes of threads, processes and computing nodes. The authors found a significant proportion of application hangs, which led them to recommend the use of application-level watchdog mechanisms.

We are not aware of any work on CORBA ORBs simulating finer-grained faults, such as bitflips. However, the technique has been extensively applied for the failure analysis of other executive software, including operating systems and language runtimes. To illustrate the information that can be obtained from these types of experiments, we present results extracted from campaigns applying the *MAFALDA* tool to a number of COTS[3] real-time microkernels, including Chorus and LynxOS.

Figure 3 illustrates results obtained by subjecting an instance of the Chorus/ClassiX microkernel (composed of basic functional components implementing basic services such as synchronization, memory, and scheduling) to a series of SWIFI experiments using *MAFALDA*.

[3]COTS: Commercial Off-The Shelf

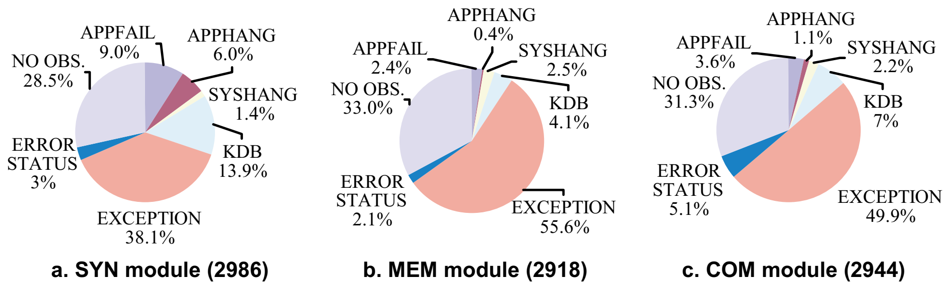

Figure 3: Results from *MAFALDA* applied to Chorus (internal bitflips)

These experiments consist in selecting randomly a location in the kernel address space and randomly flipping a bit into this memory cell. The bit is restored as soon as the cell is read, irrespective of whether the cell contains instructions or data. The pie diagram in Figure 3a shows the failure modes observed when about 3000 faults (transient single bit flips) were injected in the code segment of the standard synchronization component. Regarding the failure modes, about 50% of the errors were successfully detected by the microkernel error detection mechanisms ("error status", "exception", "kernel debugger [KDB]"), while a hang ("system hang [SYSHANG]", "application hang [APPHANG]") occurred in 7.4% of the cases. Nevertheless, 9% of the errors led to an incorrect service ("application failure [APPFAIL]"). Finally, the "no observation [NO OBS]" category (29%) corresponds to errors that had no observable consequences although the injected faults were actually activated. Similar results can be obtained on different kernel components such as the memory management module (Figure 3b) and the communication management module (Figure 3c).

Section 5.1 discusses how a tool like *MAFALDA* could be used to characterize the failure modes of a CORBA-based middleware implementation.

3.2 External faults at a linking interface

Another source of faults that can affect a component system is the linking interface, through which it is connected with other systems. This form of fault injection provides a means for measuring a system's *robustness*, its ability to function correctly in the presence of invalid inputs and stressful environmental conditions. Robustness testing involves injecting corrupted data at the linking interface of the system, and observing its behaviour.

In [Miller *et al.*, 1990], the robustness of different implementations of standard UNIX utilities was measured, by submitting them to randomly generated input. Despite using an extremely simple failure mode classification (crash or not-crash), this *fuzz testing* showed that most implementations had quite high failure rates.

The technique has also been applied to the robustness testing of POSIX-compliant operating systems in the context of the Ballista project [Koopman and DeVale, 1999]. This work consists of using invalid parameters in system calls, such as null pointers, or using an incorrect sequence of system calls. It is based on a library of corruption test cases, specialized for each data type. Figure 4 shows that all of the 15 targeted operating systems exhibited a large proportion of non-robust behaviours (e.g., between 18 and 34% of the tests lead to an *abort* failure mode).

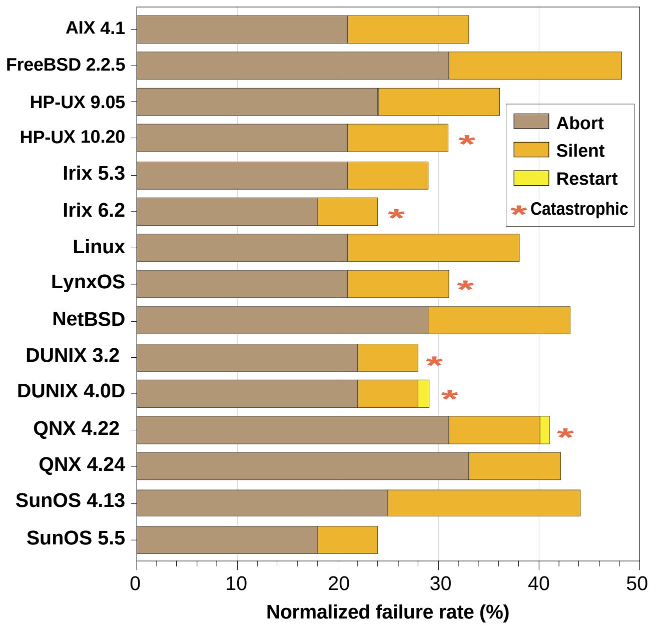

Figure 4: Comparison of 15 POSIX-compliant operating systems

The Ballista approach has also been applied to the robustness testing of a number of CORBA implementations [Pan *et al.*, 2001], with respect to corrupted invocations of a portion of the client-side interface exposed by an ORB. For example, the `object_to_string` operation, which converts an object reference into a textual representation, is invoked with an invalid object reference, to see whether the ORB crashes or hangs or signals an exception.

Figure 5 presents results from this paper. It shows the breakdown of experimental outcomes for the different targets. ORB implementations from three different vendors were tested, with different versions and on different operating systems. Their experiments show a high proportion of non-robust behaviour such as thread hangs and crashes.

The fault model considered in this work only targets client-side operations. In particular, activity that involves interaction between a client and a server is not covered. Furthermore, the functionality exposed through ORB's client-side interface, which was targeted in this research, is mainly used during the initialization of an application. Most of the functionality provided by an ORB

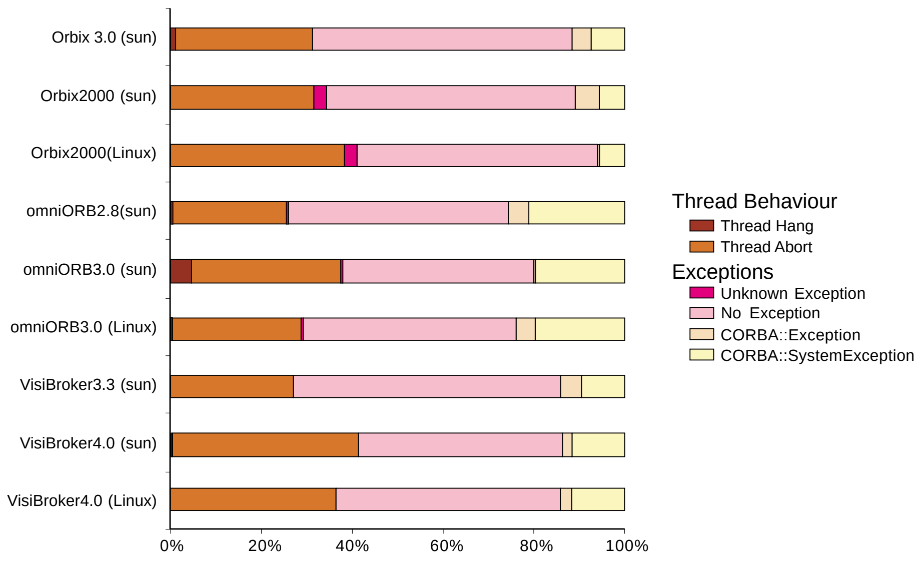

Figure 5: Ballista project applied to ORB characterization

is in fact implicit, in the sense that it is activated without any explicit calls made by the application level, and is thus difficult to target using this approach.

We are not aware of other characterization work on CORBA using fault injection. Other validation efforts have used a functional testing approach (such as the *CORVAL* project, which aims to test the functional correctness and the interoperability of ORB implementations) or concentrated on performance evaluation (e.g. [Nimmagadda *et al.*, 1999]), without considering the presence of faults.

3.3 External faults at a local interface

A DSoS component system [Jones *et al.*, 2001] may also communicate with its environment through one or more *local interfaces*, and may be subjected to faults arriving through them. Examples of local interfaces are a system's network stack, and interfaces over which it may be providing legacy services. The types of faults that may arrive over a local interface are protocol errors in a communication with a remote system.

There has been work on fault injection for the characterization of the behaviour of the networking stacks in UNIX operating systems [Dawson *et al.*, 1997]. In this work, faults such as message loss, delays and reordering were injected, to assess the robustness of the protocol implementations. In [Labovitz *et al.*, 1998], the stability of routing protocols used on the Internet is studied.

4 Fault pathology of CORBA-based systems

In this section we describe the architecture of a middleware-based system, identifying the places in the system where faults may occur, grouping these faults into classes, and classifying the types of failures that it may exhibit.

4.1 Error confinement regions in a CORBA-based system

Middleware is generally seen as a layer of software that lies between the operating system and the application layer, as shown in Figure 2. This high-level view of an ORB is sufficient for most CORBA development. Indeed, the CORBA specifications are implementation-agnostic, and do not mandate any specific representation for CORBA objects, or require any particular form of interaction with the underlying operating system. For dependability analysis, however, more detailed knowledge of the architecture of a CORBA-based system is necessary, particularly with respect to the error confinement regions implied by the architecture.

CORBAservices such as naming and the interface repository are generally implemented as daemons[4]. A service may run on a single computing node, or may involve the collaboration of multiple computing nodes (federation of name servers, for example).

A CORBA ORB can be implemented in several different ways:

- kernel-based strategy, where the ORB is provided as a service of the operating system. This strategy can allow certain performance optimizations, since the operating system knows the location of object, and can facilitate authentication of requests.
- daemon-based strategy, where ORB functionality is provided by one or more daemon processes, which mediate between clients and object implementations. For example, each computing node may run an activation daemon that is responsible for activating server processes and dispatching incoming requests, and for routing outgoing requests to the appropriate computing node. This implementation strategy facilitates centralized administration, since all CORBA processes are known to the activation daemons.
- application-resident strategy, where code implementing the ORB functionality runs in the same execution context as the client and the object implementations. The ORB is typically provided as a shared library that is linked with CORBA applications. This is the most common implementation strategy currently used on POSIX-like systems. In fact, even in the two

[4]daemon: standalone operating system process that runs in the background, providing some form of service.

previous implementation strategies, a certain amount of ORB functionality is hosted in each CORBA application, to deal with the language mapping.

The kernel-resident implementation strategy protects the ORB from modification by faulty application programs, due to the operating system's memory protection mechanisms. However, the ORB service constitutes an error propagation channel for all applications on the same computing node. The daemon-based strategy introduces a single point of failure per computing node, since failure of the activation daemon will impact all application processes on that node. The degree of error confinement offered by the application-resident implementation strategy depends on the way in which CORBA objects are mapped onto the processes and threads provided by the underlying operating system. This mapping is (deliberately) left unspecified by the CORBA standards, and different deployment configurations are possible:

- a dedicated computing node for each CORBA object. In this case the only error propagation channel is through the network (and through calls to CORBAservices). However, it is unsuited to a system comprising a large number of lightweight objects.
- each CORBA object in a separate operating system process. This is a heavyweight solution when large numbers of objects are required, but provides good error confinement, since the crash of one object does not mechanically cause the crash of other objects running on the same computing node.
- multiple CORBA objects per operating system process. This technique, which is called *collocation*, leads to several objects sharing the same address space. The crash of one object may cause the crash of all the collocated objects, so this choice clearly provides the least error confinement.

4.2 A fault model for CORBA-based systems

In general, information on the types of failures experienced by a class of systems, and the rates at which they tend to occur, are obtained from field measurements. However, we are not aware of any such study for middleware-based systems. Consequently, we can only derive a list of the classes of faults that affect these systems through structural analysis, by studying the architecture of a typical system, and examining the points where faults may arise, and how they can propagate through the system.

Figure 6 provides a more detailed view of the path taken by a remote method invocation, from the invoking object to the servant implementing the service. The figure shows that many different layers of software and hardware are traversed by

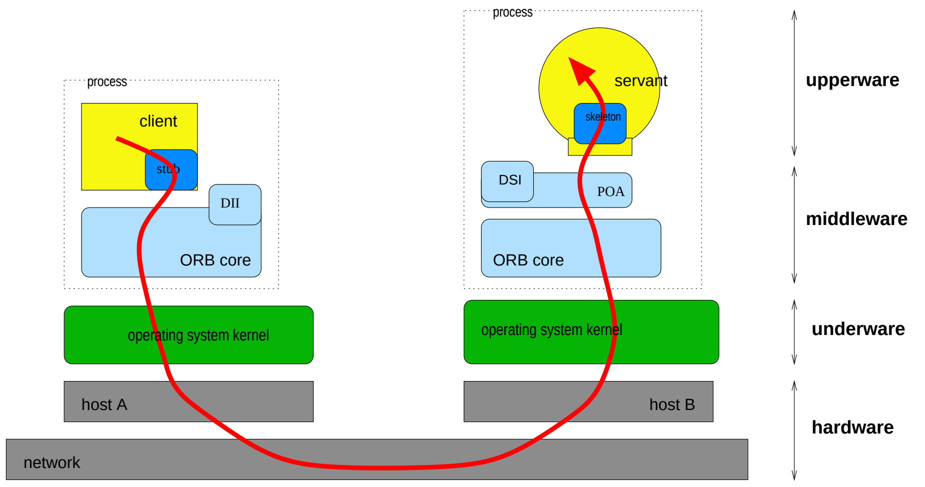

Figure 6: Details of the path of a CORBA request

the request; clearly, the impact of faults affecting each layer must be taken into account when considering the failure modes of a CORBA-based system.

The types of faults that can affect a CORBA-based distributed system can be classified as follows:

- physical faults affecting RAM or the processor's registers (so-called Single Event Upsets or soft errors [Ziegler and Srinivasan, 1996]). For example, a hardware fault may cause a bit to be flipped at one or more addresses in memory.
- software faults (design or programming errors) at the application, middleware and operating system levels. For instance, an application may pass a NULL pointer to the middleware, or the middleware may omit checking of error codes returned by the operating system.
- "environmental" faults, such as the interruption of network connections and disk-full conditions.
- resource-management faults: "process aging" produces effects such as leaking of memory (particularly common in CORBA applications), fragmentation effects, exhaustion of resources such as file descriptors.
- communication faults, such as message loss, duplication, reordering or corruption. While this class of faults is widely assumed not to affect middleware that builds on a reliable network transport protocol, as is the case of CORBA's IIOP, recent research discussed below suggests that they deserve attention.

While a system of systems is subject to each of these fault classes, physical faults and software faults are less specific to this DSoS context than the last three fault classes. Consequently, our work has concentrated on studying environmental, resource-management and communication faults.

4.3 Failure modes of an ORB

In this section, we briefly analyze the ways in which an ORB may fail, and the impact of these types of failures on the system that builds on the middleware. We classify the failure modes of an ORB as follows:

- crash of a process or of a thread;
- hang of a process or of a thread;
- corruption of incoming and outgoing data;
- omission and duplication of messages;
- incorrect signaling of exceptions.

The impact of these failure modes depends on the capacity of the system to detect the failure, and on the degree to which it can mask or recover from the failure. The most severe failure modes are those which are not detected by the system, and which therefore allow an error to propagate from the middleware to the application level.

As was noted in the previous section, the effect of a process or thread crash in terms of propagation depends on the choice of mapping between CORBA objects and execution entities. The time taken to detect a process crash or hang also depends on the system's configuration; in certain cases, a remote client may not detect the failure in a reasonable time span.

Concerning exception signaling: during a CORBA method invocation, the ORB on the client side is responsible for propagating exceptions that occurred on the server to the application level. On the server side, the ORB is responsible for propagating any exceptions that occur during the processing of a request to the client. If this signaling is incorrect, either because an ORB does not signal an exception when it should have, or because it has signaled a spurious condition, the system's fault tolerance mechanisms will not be activated correctly.

5 Method and experimental techniques

In this section we describe a method for the experimental characterization of the failure modes of a middleware implementation. This method is derived from the fault model presented in the previous section, and from analysis of the feasibility of different forms of failure observation and fault injection techniques. We present a number of experimental fault injection techniques that can be used to assess the impact of different fault classes on a middleware.

Several factors must be considered before launching a fault injection campaign:

- the fault model: which classes of errors to insert, where to insert them, and when? The injection may be triggered by the occurrence of an event of interest, or occur after a predetermined time period. The fault may be transient in nature (e.g, a single bit-flip), or permanent (e.g., a stuck-at fault).
- the observations: how to monitor the system's behaviour and classify the failure modes? It is important that all significant events be observed, which may be difficult in a distributed system.
- the workload: what operational profile or simulated system activity should be applied during the experiment? The workload is evidently very dependent on the target system. Different workloads may lead to slightly different results, since they cause different system activation patterns.

The rest of this section presents a number of approaches for fault injection in a CORBA environment, which simulate the different fault classes that were identified in Section 4.2. These fault injection techniques are classified according to the origin of the fault they simulate:

- internal faults, arising either from hardware or software faults, simulated respectively using memory bitflips and program mutation techniques;
- faults propagating from the application level, simulated using robustness testing and performance stress-testing;
- faults propagating from the underlying operating system, simulated using system call interposition techniques;
- faults arriving from the network, simulated using message corruption and reordering techniques.

5.1 Corruption of the memory space

This fault model consists of simulating the impact of faults affecting the memory subsystem of the host computer, in regions such as the RAM, the processor's registers, and its I/O controllers.

Two classes of faults can be injected:

- permanent faults, resulting from faulty memory components. These can be stuck-at-1, stuck-at-0.
- transient faults, resulting from Single Event Upsets such as electro-magnetic radiation. These faults are usually more difficult to detect than permanent faults.

The trigger for the fault injection can be temporal, in which case the fault is injected a certain number of seconds after the workload has been initialized, or spatial, in which case the fault is injected once the targeted memory word is accessed by the system.

The memory areas that can be targeted depend on the ORB's implementation strategy. Characterization of a kernel-based ORB requires injections into the kernel's address space, as well as the address spaces of CORBA applications. Characterization of a daemon-based ORB will involve corruption of the memory space of the activation daemon. For an application-resident ORB, different zones of the process' address space can be targeted (see Figure 7):

- the application stack and heap;
- the private code from the stubs and skeletons, that is linked with the process;
- the stack associated with the shared library;
- the code (text zone) of the shared library.

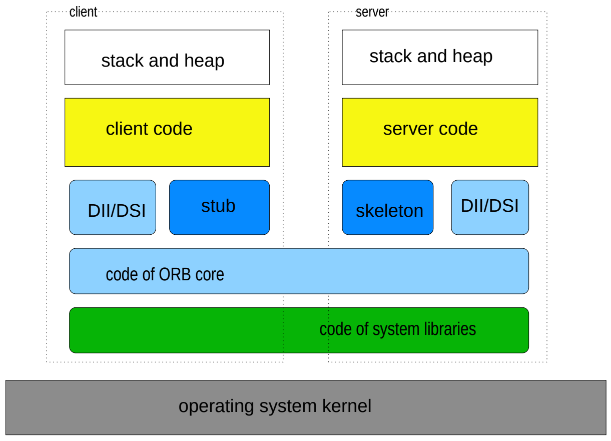

Figure 7: Memory mappings in a CORBA system with an implementation-resident ORB

While the first three sections of the address space are private to each process, the code of the shared library is – on modern operating systems – shared between all processes using the same ORB on that computing node (it is read-only, so it can be shared safely). This means that there is a potential error propagation channel through the ORB via corruption of the shared library's code (indeed, the same is the case of other system libraries, which are shared by all processes).

The information on the address ranges corresponding to each zone can be obtained from the operating system, for example by using the `/proc` filesystem on Linux.

As for the previous fault injection approaches, this technique requires a workload application and failure observers. Once these elements are set up, the experimental campaigns are similar to the targeting of other executive software components, such as operating system kernels. Indeed, existing tools such as *MAFALDA* (see Section 3.1) can be used to conduct the experiments. Measurements of factors such as exception classes and error detection latencies can be obtained.

5.2 Program mutation techniques

This characterization technique investigates the effect of software faults. It consists of artificially inserting bugs into the source code of the program, and observing the behaviour of the modified candidate (called a *mutant*). Previous work[Daran and Thévenod-Fosse, 1996] has shown that program mutation induces errors which are similar in nature to the errors produced by real programming faults.

The early focus of work using this technique was testing, where mutation is used to measure the adequacy of a set of test cases. Some more recent work[Voas and McGraw, 1997] has investigated program mutation as a characterization technique. This is closer to our aim, which is to identify the types of errors and failures which can be caused by software faults, and investigate the degree to which they are detected by the system's error detection mechanisms.

Most work in the literature consists of injecting faults which change the value of a literal constant, or the type of an operator (for example changing a `+` operator into a `-`, or changing the sign of the comparison operator in a conditional statement) [Daran and Thévenod-Fosse, 1996]. Other mutations include replacing the name of a variable or a function by another variable or function. There has been more recent work targeting object-oriented mutation operators [Chevalley and Thévenod-Fosse, 2001], such as changes between deep and shallow equality comparisons.

When applied to a middleware-based system, there are a number of different targets for mutation:

- the IDL interface itself. For example, a parameter that used to be passed using `in` conventions could be changed to `inout`. Mutation could also be applied to the data structure definitions, removing an attribute in a record definition, or replacing an unbounded sequence by a fixed length sequence.

- the stubs and skeletons automatically generated by the IDL compiler. This location simulates faults in the CORBA toolchain. For example, two parameters in a method could be exchanged before being sent over the network. If the types of the parameters are incompatible, this should be detected at compile time.
- the source code of the shared library implementing the ORB. This location evaluates the effect of residual software faults in the middleware itself. Examination of the modifications made in successive releases of a particular implementation, to determine the types of bugs that were corrected, could be an input in the construction of a model of these faults.
- at the application level. This simulates faults made by the application programmer. Classic fault models such as ODC [Sullivan and Chillarege, 1991], which include initialization faults and corruption of pointers, could be used.

In the same way as the use of an object-oriented programming language introduces new classes of software faults, which are simulated by object-oriented mutation operators, it would be interesting to identify a number of CORBA-oriented mutation operators which could simulate software faults specific to the use of a CORBA ORB. For instance, memory management is notoriously tricky in a CORBA context, when using primitive programming languages that do not provide automatic storage management (such as C and C++), so would be a promising target for mutation operators. Other mutation operators could involve the use of object references.

Unfortunately, this work is necessarily programming-language specific. While the most commonly used ORBs are implemented in C++, some are implemented in other programming languages. Applying the same work to these ORBs would require porting of the program mutation toolchain; the effort required for this work would depend on the extent to which the mutation operators are specific to a particular programming language.

5.3 Robustness testing

The *robustness* of a system is a measure of its ability to function correctly in the presence of invalid inputs and stressful environmental conditions. Robustness testing involves injecting corrupted data at the external interface of the system, and observing its behaviour.

Robustness testing requires that the system under test present an explicit interface, which will be targeted by the fault injector. This is problematic in the case of an ORB, since most of the functionality provided by an ORB is implicit. Indeed, the explicit functionality provided by an ORB is limited to the following:

- ORB initialization: passing environment information to the ORB library and obtaining bootstrap references for the ORB and services;
- POA[5] management (on a server object): methods allowing servants to register themselves with the object adapter, and control their lifecycle;
- policy management: dynamic changes to the way aspects such as the concurrency model, or access control, are handled;
- the conversion of object references to and from textual representation;
- utility procedures for the creation of certain data types and lists of values.

The work reported in [Pan *et al.*, 2001] on the robustness testing of ORBs targeted about 20 operations in this interface. These operations constitute only a relatively small portion of the functionality provided by an ORB, and are primarily used during the initialization of an application. Indeed, most of the functionality provided by an ORB is implicit, rather than resulting from explicit calls to a public interface. Consider for example a basic CORBA method invocation in a C++ program:

```
result = theObject->theMethod("argument1", 42);
```

The variable `theObject` is an instance of a class that extends classes provided by the ORB implementation. The ORB identifies the computing node on which the object is running, connects to a given port on that machine, serializes the parameters of the call to a standard format, and sends them over the connection. It then waits for the server's response, and deserializes the reply into the variable `result`, or signals a C++ exception.

All this activity is transparent to the application programmer, since the call is syntactically identical to a standard method invocation on a local, non-CORBA object. Given that this functionality is not exposed via an explicit interface, the standard robustness testing approach cannot be applied.

This implicit functionality provided by the ORB can be broken down into a number of categories:

- interaction with the application programming language: implementing marshalling and demarshalling code, handling object creation and destruction, exception handling;
- network-related processing: resolving the addresses of hosts, establishing network connections, sending and receiving information from remote hosts;

[5]POA: Portable Object Adapter, responsible for dispatching incoming method invocations to the correct servant, and for controlling the lifecycle of servants

- handling concurrency according to requested policy, in cooperation with the operating system;
- resource management: allocating and freeing buffers, etc.

We would like to apply robustness evaluation techniques to these classes of functionality. Since they are not accessible through a standard interface, we propose to generate synthesized interfaces that can serve as targets for fault injection. The purpose of these synthesized interfaces is to provide a way of activating the implicit functionality provided by an ORB. Ideally, we would like to be able to activate each class of implicit functionality individually, to obtain detailed failure mode information. However, it is not possible to isolate certain functional classes from the others – almost all interactions with an ORB will make use of the marshalling and networking functionality, for instance.

The following requirements should be satisfied by the synthesized interfaces and the corresponding service implementation:

- they should use all the different data types that can be defined in OMG IDL, including compound data types such as structures;
- they should include operations with arguments and return values that cover the possible combinations of these data types, including the different argument passing conventions (`in`, `out` and `inout`);
- given the large number of test cases implied by the two preceding requirements, the injection code targeting these synthesized interfaces should be generated automatically. Likewise, it should be possible automatically to generate a workload application for a given interface (clearly, this will severely limit the semantic level of the services which we can target);
- the service implementation should be deterministic, so that failure of the service can be detected automatically at the application level;
- the service should be dependent on the history of previous invocations (i.e., it should not be stateless). If the service depends on some internal state, there is a greater probability of faults propagating to the interface than if the service were stateless.

These requirements can be met by a delayed echo service, consisting of operations that take any number and type of arguments, and return the arguments supplied by the previous call to the service. This service can be implemented for arbitrary method signatures, is deterministic, and is not stateless.

A fault injection campaign using this approach consists of the following steps:

- generate an interface with some combination of data type definitions and operation signature. Since the set of possible interfaces is infinite, the generation process is probably random, possibly weighted.
- generate corresponding implementations for the service, the workload, the fault injector, and the fault observer.
- invoke the service with corrupted parameter values, and observe the service's behaviour.

The generation of parameters for the invocations of the service is a well-known problem in functional testing. There are a number of possible techniques, including statistical generation [Thévenod-Fosse *et al.*, 1991]. The corruption of these parameters is necessarily dependent on their type, and on the programming language mapping. Many OMG IDL types are "incorruptible", in the sense that all the bit sequences that can be represented in memory have a valid representation in the given type. Certain types, however, have a restricted domain, and can thus be subjected to out-of-range corruption.

The interfaces, service implementations and workload described above can be reused for a number of the fault injection techniques described in the following sections.

5.3.1 Performance stress-testing

Another form of robustness testing is performance stress testing, where the unexpected inputs to the system consist of an unusually intense activity of the workload. These performance tests evaluate the scalability of the service, in terms of the average response time and jitter, as a function of the number of incoming requests per second, and also as a function of the complexity of the request.

This approach is particularly well suited to the characterization of CORBAservice implementations, since their level of performance can affect the whole system of systems, and timely responses may be critical for services such as Notification[6].

5.4 Syscall interposition techniques

This fault model investigates fault propagation to the middleware from the operating system kernel and system libraries. The failure at the operating system level can have resulted from various types of faults, both hardware and software.

The middleware layer depends on services provided by the operating system kernel, such as networking, scheduling of threads, and stable storage provided

[6]The CORBA Notification Service provides a publish/subscribe infrastructure that mediates between event producers and event consumers, according to certain Quality of Service policies.

through the file system. There are a number of ways for an error to propagate from the operating system to the middleware[7]:

- returning an error code from a system call. Indeed, most system calls return a status code indicating whether the requested operation completed successfully. If not, the application can read an integer code that indicates the reason for the failure.
- signalling an exception: the program's execution is interrupted by the arrival of a signal. If the application has registered a handler for this signal, it has the opportunity to run the related code; otherwise the application is aborted by the operating system. For example, the `ALRM` signal is used to notify an application that a timeout has expired.
- taking too long to complete a certain system call (for hard real time applications).
- corrupting data during input/output operations, for example while reading and writing to stable storage.
- failing to inform the application that some event has occurred. For instance, applications can use the `select` system call to sleep until activity is detected on a set of file descriptors. If the operating system does not wake up the application, it won't handle incoming messages.

A robust middleware implementation should be able to handle (certain classes of) failures of the operating system gracefully. In many cases, this would involve signalling an exception to the application level, to allow any error recovery mechanisms to be executed. The response to certain types of exceptional conditions is specified by the CORBA standard. For example, a `NO_MEMORY` CORBA exception must be used to signal a problem with dynamic memory allocation, and a `PERSIST_STORE` exception to signal a problem with persistent storage on the server.

A campaign using this fault injection technique consists of observing the effects of these unexpected operating system behaviours at the middleware level. The experimental testbed includes a system call interposition layer, which is able to intercept a given system call made by the middleware. Instead of propagating this call to the operating system kernel, the interposition layer returns – possibly after a certain delay– an error code to the middleware. The behaviour of the middleware is then observed, from both the application level – is an exception raised, or is the fault masked – and from the operating system level, to see whether the system call is repeated (providing information on the middleware's error recovery mechanisms).

[7]In the following we use terminology from the POSIX standard, though similar concepts exist in most modern operating systems.

The fault injection campaign could be run randomly, by arbitrarily selecting the targeted system call for each run. However, more interesting analysis of the experimental outcomes can be obtained by targeting specific system calls, when the middleware is in a known state. Using this approach, it is possible to determine what activity the middleware was involved in when the fault was injected, and examine the corresponding source code to isolate portions of code that could be made more robust.

A particularly common activity in CORBA middleware is the exchange of a number of messages with a remote host. This activity results in a certain trace of system calls, which is represented in Figure 8.

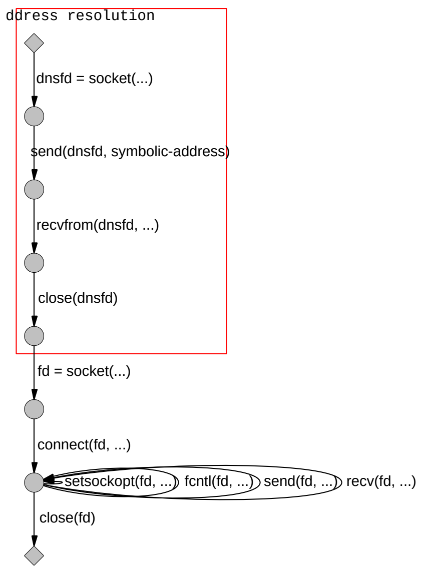

Figure 8: System call graph for a network communication

The initial part of the activity resolves the target host's symbolic name into a numerical network address. The middleware then creates a communication socket to this address, which it accesses via a numerical descriptor, and optionally sets various flags on the socket. It then sends and receives messages using this descriptor, and finally closes it.

In order to use the syscall interposition technique to target specific middleware activities, the following information must be available:

- a description of the operating system's service interface, listing the signature of each system call (including the types of its parameters and the return value) and the meaning of each of the error codes which can be generated by that system call. This information is available in the operating system's programming manual.

- a list of activity graphs for the target ORB implementation. The system call traces can be generated using a tool such as *truss*[8].
- a workload and failure observer. Those developed for the robustness testing approach (see Section 5.3) can be reused in this context.

A fault injection campaign for this fault model involves targeting each activity graph. For each activity graph, a given sequence of system calls is selected. Then one system call within this sequence is selected for corruption. For this system call, one of the possible failure modes is selected. A syscall interposition layer is generated for that syscall and fault activation sequence. The interposition layer will detect the targeted sequence of syscalls, which becomes the trigger for the injection.

5.5 Network-level faults

This fault model consists of simulating the effect of faults affecting the communication subsystem. This approach is particularly interesting in a DSoS context, since it provides information on the way in which errors may propagate between component systems, through the communication infrastructure.

This approach investigates the impact of corrupt method invocations arriving over the network. It consists of sending a corrupted request to the target, and observing its behaviour. This fault model simulates three different classes of faults:

- transient physical faults in the communication subsystem, resulting for example from faulty memory banks in routers, or faulty DMA transfers with the network interface card. Network corruption, even over reliable transport protocols such as TCP (on which IIOP is based), is more frequent than is commonly assumed. Based on analysis of traffic traces on a LAN and the Internet, [Stone and Partridge, 2000] reports that approximately one packet in 32000 fails the TCP checksum, and that between one in a few millions and one in 10 billion packets are delivered corrupted to the application level. This is because the 16-bit checksum used in TCP is not able to detect certain errors. While this proportion is very small, it is non-negligible given the high capacity of modern LANs.
- propagation to the target of a fault that occurred on a remote computing node interacting with the target. The fault may have affected the remote operating system kernel, its protocol stack implementation, or the remote ORB, leading to the emission of a corrupted request.
- malicious faults, such as denial of service attacks against the target. Given the pivotal role of the name service in most CORBA-based systems, an

[8] See Figure 14 for an example of the type of information provided by this tool.

attacker who can crash the service may be able to cause the entire system to fail. We note, however, that most CORBA systems will be deployed on private networks where all parties can be assumed to be trustworthy.

The types of errors that could be investigated include single bitflips, and the zeroing of successive bytes in a message. These are among the most common patterns of corruption identified in [Stone and Partridge, 2000], and we assume that they are representative of error propagation from remote nodes.

There are several possible means of injecting these faults. We could use dedicated network hardware, but this is cumbersome and expensive. Using software-implemented fault injection, faults could be injected at the protocol transport layer (for example by instrumenting the operating system's network stack, as in[Dawson and Jahanian, 1995]). However, there is a very high probability that this form of corruption is detected by the remote host's network stack, and therefore not delivered to the middleware. Consequently, it would be more efficient to inject the fault at the application level (see Figure 9), before the data is encapsulated by the transport layer. These experiments simulate the proportion of corrupt packets that TCP incorrectly delivers as being valid.

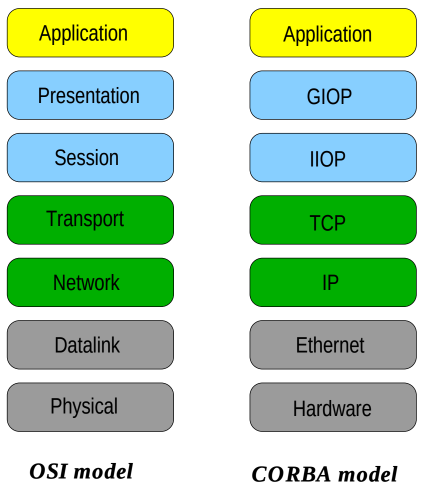

Figure 9: Protocol levels in CORBA

5.5.1 Network protocol faults

As well as considering faults that corrupt the data contained in incoming messages, it would be interesting to consider the impact of higher-level faults, which affect the semantics of the message rather than its syntactical information.

We do not investigate protocol faults that occur at the transport level of the network protocol stack, since these will be handled by the operating system's networking stack rather than by the middleware. Rather, we concentrate on protocol faults at the level of GIOP, and more specifically IIOP, its mapping onto TCP (see Figure 9). The types of unexpected conditions to which we can expose a target implementation include:

- The reception of unexpected GIOP messages. For example, GIOP specifies a LocateReply message type, which is sent in response to a LocateRequest message. An injected fault could consist of sending LocateReply message to an object, without it having emitted a corresponding LocateRequest message.
- The reception of GIOP messages containing strange request-ids. Each GIOP message contains a request-id, which is a numerical identifier for the request. This request-id is then used in the response, to identify a response with a request. The target could be sent dummy responses containing request-ids that it didn't send. Additionally, the effect of request-id duplication could be studied (the receiving ORB should drop messages containing request-ids that it has already handled).
- The reception of GIOP messages whose IORs contain unusual service contexts. The service context is used by the ORB to contain connection-related information (such as a session key, or the identifier of the character set negotiated upon establishing the network connection).

6 Experimental framework and results

In this section, we present the results of work carried out at LAAS using the network-level corruption fault model described in Section 5.5. Our motivation for selecting this fault model from the list of techniques presented in the previous section, for our initial experimental work, is its relevance in the context of the DSoS project. Our experiments aid a system integrator in the selection of a middleware implementation to be used in wrapping and as a support for the interconnection infrastructure, by assessing the robustness of different candidate implementations. Furthermore, the method provides a means of characterizing the nature and the likelihood of error propagation between component systems, through a CORBA-based interconnection infrastructure.

These experiments targeted different implementations of the CORBA Name Service [OMG, 2001b]. This service provides a hierarchical directory for object references, allowing server applications to register a service under a symbolic name, and clients to obtain references by resolving a name.

We chose this target since its standardized interface makes it easy to compare different implementations of the service. Furthermore, the Name Service may constitute a single point of failure in a CORBA-based system: while it is possible to deploy applications without using a naming or trading service, by allocating object references statically, most systems require the dynamism provided by this service.

The same failure mode characterization techniques could be applied to other CORBA services, as well as to user services implemented on CORBA. We also believe that the failure modes exhibited by a vendor's implementation of the name service will also be present, to a significant extent, in other applications built using the vendor's CORBA ORB. Indeed, a vendor's name service implementation is typically composed of a certain amount of application code implementing the service-specific functionality, which is linked with the vendor's shared library implementing its ORB. A significant proportion of the robustness failings we have observed are relatively low level, and thus more likely to come from the ORB library than from the application code; we would therefore expect that they will also be present in other applications using the ORB.

6.1 Failure modes

Although the classification of failure modes may depend on the target component, the various possible outcomes of a component's behaviour in the presence of faults are similar. Roughly speaking, either the fault is successfully detected by various error detection mechanisms (behavioural checks, executable assertions, hardware mechanisms, etc.) and signalled by different means (error status, exceptions, interrupts, etc.) to the interacting components, or it is not.

The latter case is the more difficult to classify. The first possible situation is the crash or the hang of the target component. Observing this situation involves external mechanisms that control the liveness of the component under test. When no crash or hang are observed, then more subtle mechanisms must be used to distinguish the correct outcomes of the target. In testing, this is known as the notion of Oracle. This Oracle must be defined beforehand and is part of both the activation profile of the component under test and the fault injection campaign at runtime. Indeed, during a test experiment, the outputs of the component must be obtained to be compared (at the end) to the Oracle. This is the only way to detect incorrect behaviour of the target component during the test phase, when built-in error detection mechanisms fail.

We classify the experimental outcomes for injections targeting the name service as follows:

- kernel crash: the computing node (or nodes) hosting the service becomes inaccessible from the network. We test for this condition by attempting to execute a command from a remote machine.
- service crash: attempts to establish a network connection to the service are refused. Typically this means that the process implementing the service has died.
- service hang: the service accepts the incoming connection, but does not reply within a given time span. Note that this does not necessarily mean that other clients of the service are blocked, since processing may continue in other threads.
- application failure (error propagation to the application level): the service starts returning erroneous results to clients. We assume conservatively that error propagation to the application causes an application failure.
- Exception: an invocation of the service results in a CORBA exception being raised. We distinguish between System Exceptions (which come from the ORB) and User Exceptions (which are raised at the application level).

The observation of these failure modes is a crucial issue in a fault injection campaign. It is difficult to achieve 100% coverage of the error detection mechanisms, so some failures may be undetected. In particular, since all fault injection experiments are finite in time, it is possible for an injected fault not to lead to any observable effect during the duration of the experiment. This does not necessarily mean that the fault has no effect, since its effect may be postponed after the end of the observation period (notion of error latency).

These failure modes are not equivalent from a dependability point of view. Signalling an exception is the “best” experimental outcome, since the service remains available to other users, and the application can decide on the most appropriate recovery action, such as retrying the operation (in the case of a `TRANSIENT` exception) or deciding to use an alternative service (for `COMM_FAILURE`). It is important that the exception provide as much information as possible; `COMM_FAILURE` is more useful than `UNKNOWN`, since in the latter case the application has less information on which to base its recovery strategy.

The most serious failure mode is error propagation to the application level; indeed, any fault tolerance mechanisms implemented at the application level will not be activated, and the error is free to propagate to the system's service interface. The kernel and service crash and hang failure modes, while not positive outcomes, are considered less serious, since they can be detected by system-dependent mechanisms such as watchdog timers.

6.2 Experimental setup

The infrastructure we use to support our fault injection experiments is shown in Figure 10. It consists of the following components:

- the workload application, which activates the target service's functionality (the workload runs on a different computing node from the service);
- the fault injector, which sends a corrupted request to the target once the workload has been running for a certain time span;
- monitoring components, which observe the behaviour of the target and log their observations to an SQL database;
- offline data analysis tools, to identify the various failure modes by examining the data collected by the monitoring components.

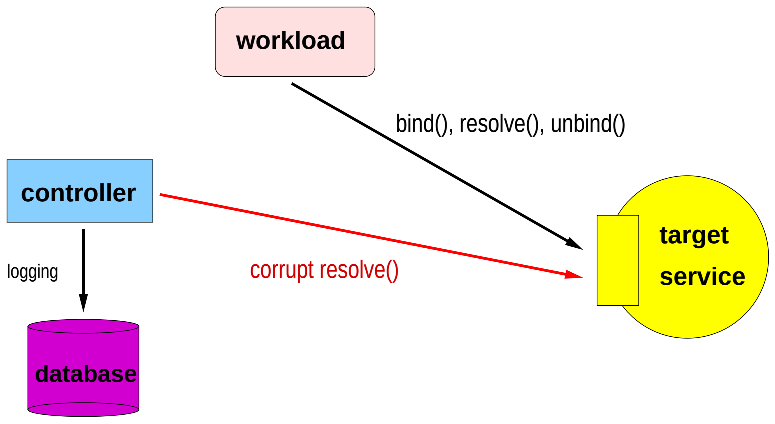

Figure 10: Experimental configuration of our testbed

Our workload application repeatedly constructs a naming graph, resolves names against this graph, and then destroys the graph. Since the graph is constructed in a deterministic way, the workload is able to check that the results returned by the service are correct (it plays the rôle of oracle with respect to the functional specification of the service). If the workload detects an anomaly in the operation of the target service, such as an incorrect result, this is signalled as an application failure. If it receives an exception from the target, it signals the appropriate exception outcome.

Each experiment corresponds to a single injected fault. A controller process launches the target service and obtains its object reference (in the implementations which we have targeted, the name service is implemented as a Unix dæmon). It then starts the workload application, passing it the service's reference. After 20 seconds, the fault injector sends a corrupted `resolve` request to the target service (for a name which has not been given a binding) and waits for the reply. The expected reply is a `NotFound` exception raised by the naming service. If no reply arrives within 20 seconds, a ServiceHang failure mode is signalled. At the end of the experiment, the monitoring components check for the presence of the different failure modes by trying to launch a command on the target host, checking for returned exceptions, etc.

For each targeted implementation, a fault injection campaign involves running an experiment for each bit or byte position in the `resolve` request. A campaign lasts around 48 hours per target for the bitflip fault model.

This fault injection technique is very portable, since the only implementation-specific component in our testbed is the code responsible for launching the target implementation. The technique is also non intrusive, and does not require any instrumentation of the targeted service.

6.3 Target implementations

We have carried out our experiments on four implementations of the CORBA Name Service:

- *omniORB* 2.8, by AT&T Laboratories, Cambridge. Freely available under the GNU General Public Licence, and implemented in C++;
- *ORBit* 0.5.0, also available under the GNU General Public Licence, and implemented in C;
- *ORBacus* 4.0.4, a commercial product from *Object Oriented Concepts*, implemented in C++;
- the `tnameserv` bundled with version 1.3 of Sun's Java SDK.

All experiments were carried out on workstations running the Solaris 2.7 operating system, connected by a 100Mb/s Ethernet LAN. While we tried to make the experimental conditions as similar as possible across implementations, a number of factors require particular attention:

- persistence: the *omniORB* implementation maintains log files so as to provide persistence across service shutdowns. To ensure a fresh environment for each experiment, we erase these log files before starting the service. The *ORBacus* implementation can be configured to use log files, but we do not enable them in our experiments. The two other tested implementations do not support persistence.
- number of experiments: we perform experiments for each bit or byte position in the corrupted method invocation. CORBA method invocations contain an ORB-dependent parameter called the service context (which can be used to propagate implementation-specific data and implicitly propagate transactions). The size of this parameter differs slightly between ORB implementations, so the exact number of experiments changes slightly from target to target.
- the *ORBit* implementation defaults to using non-interoperable object references. We configured it to use standard IIOP profiles.

In certain experiments, we observe several failure modes: for example a service crash will generally result in clients of the service receiving an exception indicating that a communication error has occurred. In the figures presented below, the failure modes are classified according to gravity, and for each experiment the most serious mode observed by the testbed is selected.

6.4 Analysis of results

In this section we present the results of our fault injection experiments, for both the double-zero and bitflip fault models. More general analysis from a dependable system integrator's perspective is presented in Section 6.5.

Figure 11 compares the experimental outcomes for each target implementation, for the double-zero fault model (where two successive octets of the message are set to zero). The three outcomes to the left of the legend are "bad", whereas those on the right indicate robust behaviour. The outcomes whose names in the legend are in capital letters correspond to CORBA SystemExceptions. The `NotFound` outcome is a CORBA application-level exception raised by the naming service when it cannot resolve a name; this is the expected behaviour of the service for our experiments. The sum of the vertical bars for each target is 100%.

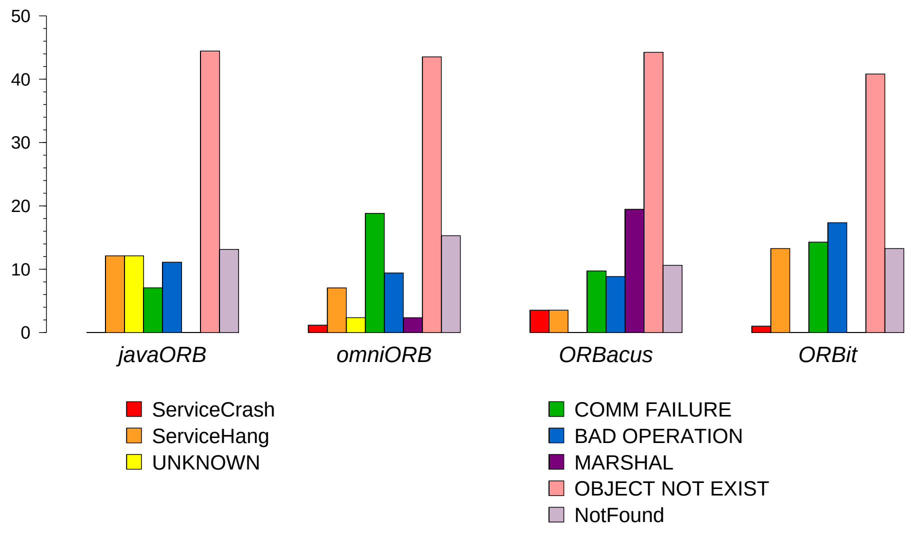

Figure 11: Experimental outcomes for double-octet zeroing fault model

A first remark is that we have not observed any cases of error propagation to the application level, which is a positive point. However, there are a relatively large proportion of service hangs and crashes.

As stated earlier, our service hang failure mode does not imply that other clients of the naming service are blocked; we only consider the time taken to reply to the corrupted invocation. However, given its relative frequency, it is one of the most serious dependability problems we have identified. The upcoming CORBA 2.4 specification allows clients to specify timeouts on their requests, which would be helpful for detecting this type of situation without resorting to application-level watchdog mechanisms. Some of the implementations tested already support these interfaces or provide similar mechanisms (but they were not activated in our experiments).

Examining the details of the breakdown of CORBA exceptions, we observe that the Java implementation raises very few `COMM_FAILURE` exceptions, but a larger proportion of `UNKNOWN` exceptions (this exception is raised by an ORB when it detects an error in the server execution whose cause it cannot determine – for example, in Java, an attempt to dereference a null pointer). `UNKNOWN` is a less useful exception to signal to the application layer, since it conveys no information on the cause of the exception, so from this point of view the Java ORB can be considered less robust. The *ORBacus* service raises a greater proportion of `MARSHAL` exceptions, which indicates that its marshalling code does more error checking than other implementations (a positive point from a robustness point of view); *ORBit* does not raise `MARSHAL` exceptions.

The proportion of `OBJECT_NOT_EXIST` exceptions, which the ORB uses to signal that the object reference against which the method was invoked does not exist, is very similar between implementations. This is to be expected, since an ORB is required to check the validity of this value before dispatching the method invocation to the appropriate servant. A similar remark can be made for the `BAD_OPERATION` exception.

6.4.1 Differences between fault models

Figure 12 shows the experimental outcomes for each target implementation for the bitflip fault model. The results differ slightly from those for the double-zero fault model. The first difference between the results from the two fault models is the appearance of a `InvalidName` exception which is not provoked by the double-zero fault model. This exception is raised by the naming service either when the name it is asked to resolve is empty, or –more likely in our case– when the name contains an invalid character.

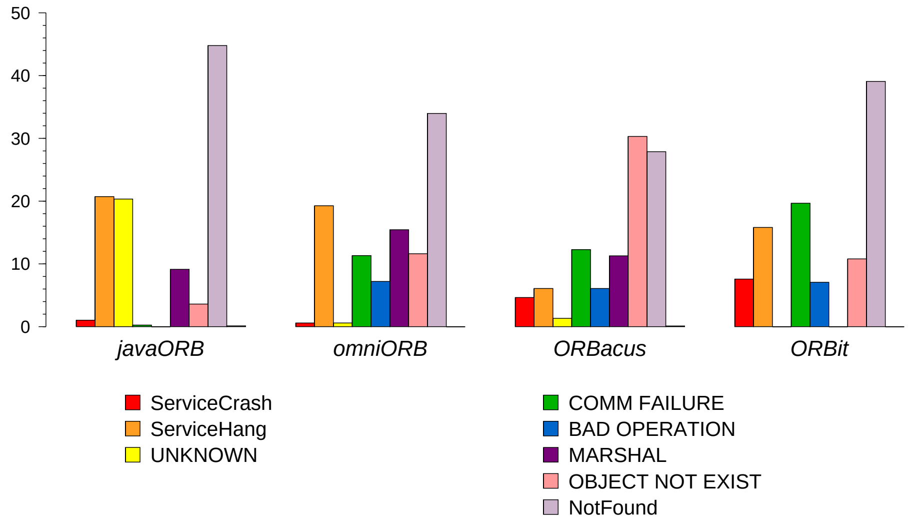

Figure 12: Experimental outcomes for bitflip fault model

A second observation is that the bitflip fault model results in a greater proportion of `MARSHAL` and `NotFound` exceptions. In the latter case, the difference is likely to be due to the service masking certain errors. Indeed, certain bits in an IIOP message are unused. For example, the byte order of a message is represented by a zero or a one marshalled into an octet; seven of these bits are not significant, and so their corruption may not be detected by the ORB. In contrast, a double-zero error is unlikely to escape the notice of the ORB.

Certain other phenomena, such as the small proportion of `COMM_FAILURE` and

`BAD_OPERATION` exceptions raised by *JavaORB* for the bitflip fault model, would require deep analysis of the source code to explain.

6.4.2 Influence of the error position

Figures 11 and 12 aggregate the results of faults injected at each possible position in the message. It is also interesting to examine the failure modes as a function of the position in the message where the fault was injected. Figure 13 shows the most common experimental outcomes for certain regions of the message[9].

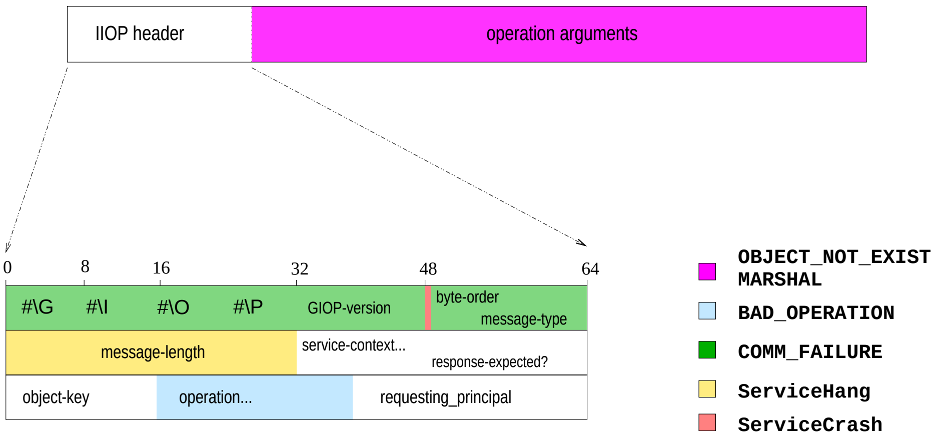

Figure 13: Format of a GIOP message

When the fault affects the part of the message which identifies the invoked operation, primarily `BAD_OPERATION` exceptions are signalled, as would be expected. Similarly, faults injected in the first few bytes of the IIOP request (which contain a special signature which identifies the message type) result mainly in `COMM_FAILURE` exceptions.

When the fault affects the header bits encoding the message's length, we mostly observe service hangs. Given that there are 32 bits to encode the message length, and that our messages are relatively short (around 900 bits), a bitflip in this zone (normally set to zero) is likely to increase the announced message length, so the service waits to read more data than will actually arrive.

6.4.3 Internal error checking mechanisms

The *ORBacus* service was compiled in its default configuration, without deactivating internal "can't happen" assertions. When these assertions fail, the program voluntarily exits using the `abort` procedure. This leads to *ORBacus*

[9]The data in the figure is valid for all the targeted implementations

showing a relatively high proportion of service failures, some of which could be avoided by using a different configuration. The *omniORB* implementation can be configured at runtime to abort when it detects an internal error, but we did not enable this feature.

6.4.4 System call trace analysis

Our testbed allows us to obtain system call traces and execution stack backtraces of the target process. These show that different middleware implementations activate the operating system in different ways. For instance, the *ORBacus* implementation makes a large number of `lwp_mutex` and `lwp_sema` calls, which enable the synchronization of threads, whereas the *omniORB* implementation uses a much narrower range of system calls, primarily for reading and writing to the network and to its log file.

The system call traces also illustrate differences in the level of internal error checking between ORB implementations. For example, when faults are injected into certain bit positions, the *ORBit* implementation causes a segmentation violation while decoding the corrupted message, and is forcibly aborted by the operating system. In contrast, the *ORBacus* implementation sometimes detects the corruption internally, and is able to print a warning message indicating the position in the program where the error was detected, before voluntarily aborting. This lack of internal error checking is an implementation decision for *ORBit*, whose primary design goals are high performance and a small footprint.

Figure 14 shows output from the `truss` tool on Solaris, for the *ORBit* segmentation violation described above. The tool generates a trace of the interaction between the operating system and a process, showing the system calls performed by the process with their arguments, machine faults incurred by the process and the signals delivered to it by the operating system kernel. The trace shows that after having read a GIOP request from the network, the name service attempts to access memory outside of its address space, receives a segmentation violation signal, and is aborted by the operating system.

```
   fcntl(7, F_SETFL, 0x00000082) = 0
   poll(0xFFBEF300, 3, -1) = 1
   read(7, " G I O P01\001\0 U\0\0\0", 12) = 12
   read(7, "\0\0\0\001\0\0\001\0\0\0"..., 85) = 85
       Incurred fault #6, FLTBOUNDS  %pc = 0x00014274
         siginfo: SIGSEGV SEGV_MAPERR addr=0x6E7A1124
       Received signal #11, SIGSEGV [caught]
         siginfo: SIGSEGV SEGV_MAPERR addr=0x6E7A1124
      siginfo: SIGSEGV SEGV_MAPERR addr=0x6E7A1124
[...]
   getpid() = 26780 [26779]
   kill(26780, SIGABRT) = 0
       Received signal #6, SIGABRT [caught]
         siginfo: SIGABRT pid=26780 uid=3905
   fstat(3, 0xFFBED8E0) = 0
[...]
   llseek(0, 0, SEEK_CUR) = 0
   _exit(1)
```

Figure 14: `truss` output showing an *ORBit* segmentation violation

6.5 Analysis from an integrator's point of view

The experimental results presented in Section 6 show a relatively large variability of behaviour of the target candidates in the presence of faults. This demonstrates that, although the service's interface is standardized, a particular candidate's behaviour depends on the design and implementation decisions made by the vendor. In this section, we adopt the viewpoint of a system integrator who must select a candidate implementation for a safety critical system.

As such, we rank first candidates that deliver relevant error reporting information, i.e., those which exhibit fewer service hangs and `UNKNOWN` exceptions. These are the most problematic failure modes when deciding on fault tolerance strategies and error recovery mechanisms that can meet the system's dependability requirements.

By grouping all the exceptions except for `UNKNOWN` together, we obtain the percentages for the bitflip fault model shown in Table 1.

Table 1: Ranking of service implementations

Implementation	**Exception**	`UNKNOWN`	**Service Hang**	**Service Crash**
ORBacus	88.0	1.3	6.1	4.6
omniORB	79.5	0.6	19.3	0.6
ORBit	76.6	0.0	15.8	7.6
Java SDK	58.0	20.3	20.7	1.0

From this viewpoint, the *ORBacus* and *omniORB* implementations exhibit the safest behaviour: more significant exceptions are reported, i.e., fewer `UNKNOWN` exceptions, and there is a smaller proportion of service hangs. *ORBacus* has a relatively high rate of service failure, which (as discussed in Section 6.4.3) is partly due to the configuration we chose. This type of reaction to abnormal situations is not necessarily a negative point from a dependability viewpoint. Many fault tolerance strategies, particularly in a distributed computing context, make a *fail silence* assumption, which requires components to produce either correct results, or none. Silent failures can successfully be handled by replication, either by using identical copies located on different sites, to deal with physical or environmental faults [Powell, 1991], or by using diversified copies to protect against software faults [Avizienis, 1975, Randell, 1975, Laprie *et al.*, 1990].

We also observed in the experiments that the behaviour depends on the fault model. The results obtained with double zeroing and bitflips lead to a different statistical distribution of the failure modes. However, the resulting numbers do not disturb the ranking given in Table 1. Many issues can influence the observed results. Nevertheless both types of experiments leading to the same conclusions reinforce the confidence one can have in the ranking.

Clearly, many other aspects of middleware dependability must be taken into account in the final selection of a candidate. In particular, the effects of other classes of faults need to be investigated. From this viewpoint, the work done by the Ballista project [Pan *et al.*, 2001], which uses a different fault model and targets a different part of the middleware, is complementary to ours.

7 Conclusions and future work

This document proposes a method for the failure mode analysis of CORBA-based systems. This method relies essentially on conventional fault injection techniques and on a clear identification of possible targets in a middleware implementation. Although the CORBA standard defines the features that must be provided by CORBA ORB implementations, their implementation may vary significantly from one vendor to another. From a dependability viewpoint, the design strategy and the implementation of the standard are of prime importance. Clearly, a middleware such as CORBA includes several facets that make the characterisation quite difficult. We analysed the various components and possible targets in a CORBA middleware and justified the use of a particular fault injection technique. In the context of DSoS, the analysis of failure modes targeting sensitive services using network corruption seemed the most relevant, and was the first to be tackled. Experiments have been carried out to obtain significant results on a number of CORBA implementations. These results show the various possible behaviours that can be observed and their impact in a system of systems, from a dependability viewpoint. The insights revealed by these experiments are novel and useful inputs to develop error confinement wrappers (cf Section 4 of DSoS deliverable IC2).

We have presented an experimental robustness evaluation method for CORBA-based services, and discuss results of experiments targeting four implementations of the CORBA Name Service. These experiments can be carried out on any CORBA service or user-defined service on top of CORBA. The choice of the Naming Service was justified by its essential role in a CORBA distributed system. It is worth noting that these experiments also evaluate the effect of corrupted method invocations at the middleware level.

The implementations we have tested show a non-negligible number of robustness weaknesses, but we have not observed any failures corresponding to the propagation of an error from the middleware to the application level. Our results suggest that the robustness of CORBA-based systems would be enhanced by the addition of an (application-level) checksum to GIOP. The achieved failure mode characterization aids in the selection of a candidate middleware implementation for critical systems, and helps DSoS system integrators decide on the error detection and recovery mechanisms, fault tolerance strategies and architectural solutions that are needed to meet dependability requirements. Our technique is non-intrusive, and (thanks to the transparency provided by CORBA) easy to port, both to new implementations of the service, and to alternative operating environments (operating system, hardware platform). The approach could also be applied for the failure modes characterization of other CORBA services, by modifying the workload and the fault injector.

This method will be used to carry out other experiments and obtain more results regarding the failure modes of a CORBA-based system[10]. First, fault injection will be performed within several target components composing the middleware using bit-flip fault injection techniques targeting memory segments (simulation of hardware faults, as described in Section 5.1). Second, we will address the robustness of implicit functions of an ORB using an *ad hoc* interface to these functions. The robustness of these essential functions will be evaluated using parameter fault injection techniques (cf Section 5.3). Third, we plan to examine the influence of faults propagating from the operating system to the middleware, as described in Section 5.4. The extension of the experiments carried out will provide useful inputs to the definition of error confinement wrappers.

Acknowledgements: The authors would like to thank Jean Arlat for helpful comments on their experiments and on early versions of this document.

References

[Arlat *et al.*, 1993] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, August 1993.

[Avizienis, 1975] A. Avizienis. Fault-tolerance and fault-intolerance: complementary approaches to reliable computing. *ACM SIGPLAN Notices*, 10(6):458–464, June 1975.

[Carreira *et al.*, 1998] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[Chevalley and Thévenod-Fosse, 2001] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for java programs. Technical Report 01356, LAAS-CNRS, September 2001.

[Chung *et al.*, 1999] P. E. Chung, W. Lee, J. Shih, S. Yajnik, and Y. Huang. Fault-injection experiments for distributed objects. In IEEE, editor, *Proceedings of the International Symposium on Distributed Objects and Applications*, 1999.

[Daran and Thévenod-Fosse, 1996] M. Daran and P. Thévenod-Fosse. Software error analysis : a real case study involving real faults and mutations. In S. J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 158–171, New York, January 8–10 1996. ACM Press.

[10]The results obtained from these upcoming experiments will be summarized in the forthcoming DSoS deliverable PCE1, together with the definition of the corresponding robustness-enhancing mechanisms.

[Dawson and Jahanian, 1995] S. Dawson and F. Jahanian. Probing and fault injection of dependable distributed protocols. *The Computer Journal*, 38(4):286–300, 1995.

[Dawson *et al.*, 1997] S. Dawson, F. Jahanian, and T. Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, December 1997.

[Fabre *et al.*, 2000] J.-C. Fabre, M. Rodríguez, J. Arlat, F. Salles, and J.-M. Sizun. Building dependable COTS microkernel-based systems using MAFALDA. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (PRDC-2000)*, pages 85–92. IEEE Computer Society Press, 2000.

[Fuchs, 1998] E. Fuchs. Validating the fail-silence of the MARS architecture. In *Proc. 6th IFIP Int. Working Conference on Dependable Computing for Critical Applications: DCCA-6*, pages 225–247. IEEE Computer Society Press, 1998.

[Jones *et al.*, 2001] C. Jones, K. Kopetz, E. Marsden, M. Paulitsch, D. Powell, B. Randell, and R. Stroud. Revised version of conceptual model. Research report, DSoS, September 2001.

[Kalyanakrishnam *et al.*, 1999] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a LAN of Windows NT based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, pages 178–189, Washington - Brussels - Tokyo, October 1999. IEEE.

[Karlsson *et al.*, 1998] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proc. 5th IFIP Working Conference on Dependable Computing for Critical Applications: DCCA-6*, pages 267–287. IEEE Computer Society Press, 1998.

[Koopman and DeVale, 1999] P. J. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, pages 30–37, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Labovitz *et al.*, 1998] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. *IEEE/ACM Transactions on Networking*, 6(5):515–528, October 1998.

[Laprie *et al.*, 1990] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, July 1990.

[Madeira *et al.*, 2000] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2000)*, pages 417–426. IEEE Computer Society Press, 2000.

[Miller *et al.*, 1990] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[Nimmagadda *et al.*, 1999] S. Nimmagadda, C. Liyanaarachchi, A. Gopinath, D. Niehaus, and A. Kaushal. Performance patterns: Automated scenario based ORB performance evaluation. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 15–28. The USENIX Association, 1999.

[OMG, 2001a] OMG. The Common Object Request Broker: Architecture and Specification. Technical report, September 2001. (formal/2001-09-01).

[OMG, 2001b] OMG. CORBAServices: Common Object Service Specification: Naming Service Specification. Documentation available at www.omg.org, Object Management Group, February 2001.

[Pan *et al.*, 2001] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber, and M. L. Jiang. Robustness testing and hardening of CORBA ORB implementations. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2001)*. IEEE, June 2001.

[Powell, 1991] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, Berlin, Germany, 1991.

[Randell, 1975] B. Randell. System structures for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Rimén *et al.*, 1994] M. Rimén, J. Ohlsson, and J. Torin. On microprocessor error behavior modeling. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, pages 76–85, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[Stone and Partridge, 2000] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 309–319, 2000.

[Sullivan and Chillarege, 1991] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[Thévenod-Fosse *et al.*, 1991] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *Fault Tolerant Computing*, pages 410–417, Los Alamitos, Ca., USA, June 1991. IEEE Computer Society Press.

[Voas and McGraw, 1997] J. Voas and G. McGraw. *Software Fault Injection*. John Wiley and Sons, 1997.

[Ziegler and Srinivasan, 1996] J. F. Ziegler and G. R. Srinivasan. Preface: Terrestrial cosmic rays and soft errors. *IBM Journal of Research and Development*, 40(1):2–2, January 1996.